Linköping University | Department of Computer and Information Science
Bachelors Thesis, 16 hp | Bachelor of Science in Computer Engineering
Spring term 2017 | LIU-IDA/LITH-EX-G—17/051--SE

# Extending a model-based IDE

## Implementing multi-user collaboration

Christopher Doane
Daniel Johannesson

Supervisor: Bernhard Thiele
Examiner: Peter Fritzson

# Abstract

This report proposes an implementation of multi-user support to an existing model-based IDE called Arctic Studio which is created and maintained by ARCCORE AB. Arctic Studio is a single-user embedded software development environment for developers utilizing the AUTOSAR standard. It is based on the Eclipse IDE and the Eclipse Modelling Framework (EMF). The implementation takes the form of an Eclipse plugin using EMFStore.

Users face a challenge to maintain consistency between several versions of a model instance. This is currently solved with the use of version control systems like GIT or SVN or by using tools such as EMF Compare to manually merge models. These traditional version control systems are not well-suited for the structure of model files used in Arctic Studio. By having model-supported version control, developers can synchronize changes in model representation and easily perform merges. To select an appropriate solution, the state of the art in version control techniques for model artifacts was presented. In this case, model artifacts are xml files that define model instances. The version control system called EMFStore was selected as it can be integrated into the Arctic Studio product. The company ARCCORE also expressed a preference for EMFStore over other solutions and named that the ability to integrate the solution with Arctic Studio was desirable. Different methods for integrating EMFStore in Arctic Studio were explored and a prototype was constructed to test the viability of using EMFStore for AUTOSAR models. Limitations of the EMFStore implementation were documented and some of them addressed in the prototype implementation. This study concludes that EMFStore can, with some integration modifications, be setup to handle version control of AUTOSAR EMF model instances in Arctic Studio.

# Acknowledgement

We would like to express our gratitude to developers at ARCCORE AB for putting up with our questions about Arctic Studio and the Eclipse Modelling Framework. Special thanks go to our advisor Tore Risinger at ARCCORE who gave valuable feedback and advice during the thesis work. We would also like to thank our examiner and supervisor at the university for their feedback during the thesis work.

Linköping, May 2017

Christopher Doane & Daniel Johannesson

# Table of Contents

# 1. Introduction

The following section introduces background information necessary for understanding the scope and relevance of this report. Following this, the purpose of the report is explained and the research questions taken up in this report are listed.

## 1.1 Background

Effectively managing a wide range of versions of the same software project among developers is an integral component of successful projects. Organizations want to allow for each individual developer to work on a copy of the code base and later add their changes to a shared repository at a time they see fit. With several versions of the same project, there is a need for means to effectively manage versions within the team, and to handle the merging or combining of all the different versions into one final product. To perform a merge manually is very cumbersome, involving a large communication-overhead between developers and can lead to unexpected bugs or other problems that can delay software from being completed on time. In addition, manually merging serialized model files is especially difficult as the serialized representation is often in a format such as xml that is (albeit subjectively) not very human-readable when files are in the megabyte size.

A common solution to the above problem is to utilize a version control system that manages a master repository or database of changes to the project's code. Version Control Systems (VCS or VC) enable several developers to edit the same program code concurrently [1]. Developers can work on their own copy of the code, called a branch, without modifying the original repository until they are ready to perform a merge. This allows developers or organizations to have a stable release repository of the project while having developers work on improving the software on different branches [1]. Version control enables developers to easily backtrack a project to a previous state. Version control also gives developers and organizations a better overview of both the current status of their project and history of changes. This information can also be used to better determine from which part of the software a problematic bug originated from.

While version control is widely used in all kinds of software projects, it is surprisingly underutilized in the development of modelling artifacts [2]. Modelling artifacts refer to a specification of information used in describing and defining models during software development. Version control or other collaborative solutions for modelling artifacts are currently either underdeveloped, nonexistent or are platform-specific [3].

By not having adequate support for version control of models, developers working on these models must deal with difficult merges where conflicts between models are not handled well. Thus, developing with the use of model frameworks can become a single-user activity as merges between different versions are difficult. This goes against a common premise that software development of complex systems often involves multiple developers working concurrently.

It is worth noting that some limited products exist on the market for managing model version control. Vector's DaVinci Configurator Pro is a commercial solution for model development with support for some model version control [4]. This product is likely to be expensive as Vector requires that one sends in a quote request to get pricing information.

We examine an IDE called Arctic Studio that currently has no model-appropriate version control system handling its models. Arctic Studio utilizes the Eclipse Modelling Framework (EMF) for its models. EMF is a modelling framework and code generator [5]. It is used to build tools and other applications based on structured metamodels. Metamodels define the abstract syntax of model components [6].

A customized version of EMF Compare (a model-based diff and merge tool) exists in Arctic Studio, however it performs poorly on large models. It has been highlighted by ARCCORE AB that the current combination of utilizing arxml files for model instances in Arctic Studio does not play well with traditional version control systems such as GIT. These textual-based version control systems were not designed to handle versioning of EMF models and do not consider the characteristics of the Eclipse Modelling Framework.

## 1.2 Purpose

The purpose of this thesis work is to introduce and integrate multi-user editing support to the Arctic Studio IDE. Currently, multi-user development is supported through the versioning of AUTOSAR model instances using traditional text-based version control systems. This is not an ideal way to handle changes in models, and ARCCORE AB has highlighted the need for a system that can better manage the versioning or synchronization of their extended EMF models. The goal of this report is to implement model-based version control into Arctic Studio to add legitimate support for multiple developers to work concurrently on a single Arctic Studio model instance project.

It is worth noting that the solution eventually proposed will be implemented as a feature into Arctic Studio. EMFStore was identified by ARCCORE AB as most appropriate as it operates on relatively unmodified EObjects and can be easily integrated into ARCCORE's Arctic Studio project. The solution will consider merge handling, error control and offer a user-friendly interface.

### 1.2.1 Research Questions

1. How can one integrate EMFStore model version control into an Eclipse-based IDE such as Arctic Studio?
2. What are the limitations of EMFStore version control when applied to AUTOSAR models? Are model semantics such as containment and non-containment references, as well as model attribute changes in AUTOSAR models handled adequately by EMFStore?

Semantics refer to the relations and dependencies between model components. Semantics are typically ignored when versioning of models are managed in textual-based version control systems, which often leads to developers having to manually handle conflicts.

### 1.2.2 Research Scope

To answer the above research questions in a timely and concise manner, the study will be limited to investigating EMFStore exclusively as a method for versioning of AUTOSAR models in Arctic Studio. Other EMF-specific version control systems will be presented in the theory chapter of the report solely to provide readers an overview of the current state of EMF-model versioning. Alternative methods, such as the extension of a text-based VCS like GIT to handle EMF-semantics are not investigated in this report. Other modelling frameworks other than EMF are also not considered, as Arctic Studio's main support is in handling AUTOSAR EMF models.

# 2. Theory

In this section, we present both theory and existing research on the topic of multi-user collaboration on modelling instances. One means of facilitating multiple developers to work on the same development project includes introducing a version control system that works well with the current development structure of the managed software.

A developer can use an editor or IDE to make changes to the software project, and then afterwards use a version control system to handle the synchronization of different versions of source code among developers. In the case of version control of modelling artifacts, solutions are non-standardized and are often platform-dependent [3]. In other cases, model-level source control has yet to be implemented in modern development environments. An example of this is how Arctic Studio, a development environment used for enterprise AUTOSAR projects, currently lacks adequate version control that can handle merges on a model artifact-level.

What follows is an overview of the Eclipse framework and of plugin development to give an adequate background of how new features can be integrated in Arctic Studio. Following this, a description of the existing version control strategies that exist today is presented. Topics such as difference detection on a model-level are described and methods to implement difference detection are presented. Once there is a sufficient mechanism for detecting differences between models, it is important to be able to handle merges between artifact versions in a structured and well-designed way. Different methods for merging have their own set of advantages and disadvantages and these are described as well.

Finally, we present an overview of existing version control systems for the modelling framework used in Arctic Studio and a basic overview of their underlying methods for difference-detection and merging. Reasoning why other model versioning other than EMFStore were not utilized is provided below.

## 2.1 Eclipse

The Eclipse Framework is a framework used to create different variations of the Eclipse IDE [7]. Eclipse itself is an IDE that supports a wide range of functionality.

### 2.1.1 Eclipse Plugin Development

A large portion of functionality in Eclipse IDEs is implemented through various plugins, and most components of the Eclipse platform are in the form of plugins. With plugins, the Eclipse Framework can be used to create a custom IDE for one's own needs. Plugins provide both integral and optional functionality to the Eclipse platform and can be considered components within Eclipse [7]. Plugins contain all information that is needed to run them in the form of a manifest file, implementation code, visual assets and more. The manifest file is used to define dependencies to other plugins as well as any available points for extension [7]. A plugin integral to Eclipse is the Workspace plugin.

Eclipse plugin development can be performed within Eclipse's Plug-in Development Environment, otherwise known as PDE, which can be obtained through various Eclipse distributions [8]. The distribution utilized within this project was provided by ARCCORE.

***Extensions and Extension Points***
Extensions are usage instances of other plugins' extension points. Extension points offer locations where developers can extend or modify the existing functionality of a specific plugin with their own plugins.

Terminology such as: "registering your class to the given extension point" is used in the Eclipse community to instruct developers to add an entry in their plugin manifest that defines what class extends the extension point. Many times, a graphical interface is provided to aid developers in registering an extension point, however this is not always available.

### 2.1.2 EMF – Eclipse Modelling Framework

EMF is a set of plugins that add support for the Eclipse Modelling Framework to the Eclipse IDE. The modelling framework includes code generation tools that generate java classes from metamodels specified in XMI [5]. The idea behind model-based development is to define the structure of models in metamodels, create model instances in an editor such as the default generated editors by Eclipse EMF or Arctic Studio, then use generators to create standardized code solutions based on the model components used in model instances. Additional tools for generation of visual assets for model components are also included in Eclipse Modelling Framework. Main classes that are used in EMF that are important to understand include the `EditingDomain`, `EObject`, `ResourceSet`, and `Resource`.

In its core, model instances in EMF work around `EditingDomains`, `Resources` and `EObjects`. Each of these are explained in greater detail below.

***Workspace***
There is one workspace in the Eclipse Modelling Edition. Workspaces are used to group projects together. Several editing domains can reside in a workspace. Each workspace contains an `EditingDomain` manager which can be used to fetch editing domains stored in the workspace.

***Editing Domain***
The purpose of an editing domain is to contain related models and functionality to make modifications to them. An editing domain contains a resource set that is used to store resources and later model instances within these resources. Changes to model instances are generally conducted via commands using the `EditingDomain`'s `CommandStack` [9]. `Commands` can be executed, undone and redone [10].

`TransactionalEditingDomain`, which is an extension of the editing domain class, is normally used when working with EMF models, `TransactionalEditingDomain` provides transactional semantics when making modifications to the `ResourceSet` [11]. Transactional semantics means that the system state is kept consistent [12]. Changes to the resource set will either be committed or ignored completely to avoid inconsistencies.

***Resource Set***

A `ResourceSet's` main purpose is to manage related resources and inform listeners of changes [13]. The `ResourceSet` class implements the notifier interface to keep track of different adapters connected to a resource set. When a change occurs and a notifier is called, all adapters connected to the resource set are triggered or notified with a message that enable them to do their functions. `Resources` are stored within a resource set, and no specific resource may reside in more than one resource set. `ResourceSets` also contain a `URIConverter` that is used to normalize URIs and is needed when serializing a resource [13].

***Resource***

`Resource` is a persistent document that is used to store an entire or partial model instance. A resource contains a tree of `EObjects` that represent the module components used in the project. All resources have a `URI` which is used to determine where the resource is stored on disk [13].

***EObject***

`EObject` is the building block of EMF model instances. `EObjects` represent the smallest building block of the tree structure in a resource and can be considered a model component. The `EObject` class defines behavior that all subclasses must implement [13]. It can contain other `EObjects`, creating a tree structure of `EObjects`. The highest level in the tree structure is always a resource. All `EObjects` either have an `EObject` as a parent or a resource as parent. A specific `EObject` instance can only exist in one resource at a time. `EObjects` may contain `EStructuralFeatures` and `EReferences` which are attributes for the object which defines it from other `EObjects`. References can either be contained or labeled as non-containment. Contained means that the `EObject` is stored within the current `EObject`, while non-contained means that the reference is stored elsewhere.

To better explain the relationship between classes, Figure 1 illustrates the hierarchy of the main classes or objects used when working with the Eclipse Modelling Framework.

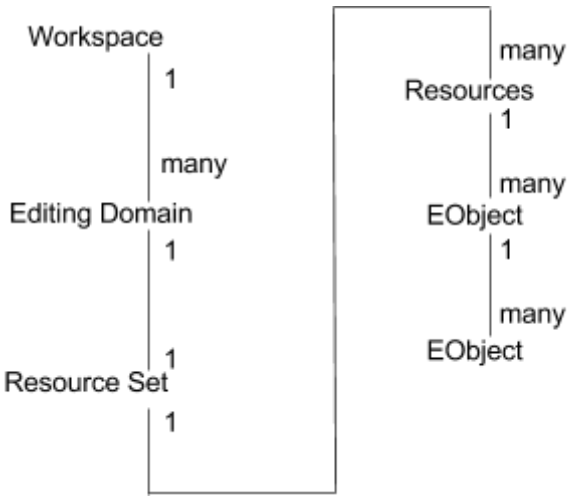


**Figure 1**. Hierarchy of main objects in Eclipse EMF.

To summarize Figure 1, Workspaces can hold several `EditingDomains`, `EditingDomains` can only hold one `ResourceSet`, `ResourceSets` can hold many `Resources`, `Resources` can hold many `EObjects` in a tree-hierarchy and `EObjects` may hold several children

`EObjects`. The focus of this report is to version these `EObjects` and their semantics between each other.

***Metamodels***

Metamodels are models that describe EMF models [7]. The metamodel that is used most often in EMF is the Ecore model. Metamodels describe what model components have for structural and behavioral features. Structural features include attributes and references that a model component may contain, while behavioral features describe available `EOperations` that can be used to modify contents of the model component [7].

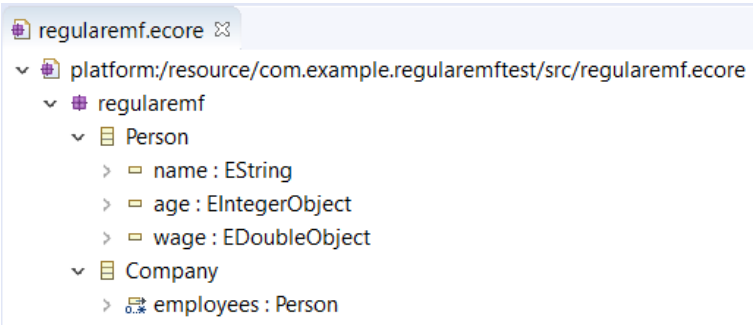


**Figure 2**. Graphical Representation of a Metamodel in Eclipse.

Figure 2 shows how a simple metamodel can be displayed visually within Eclipse. With an Ecore metamodel, the EMF.Edit library can generate editors where users can create instances of the model components described in a metamodel. XMI is used to serialize Ecore metamodels in the Eclipse Modelling Framework [7].

Describing more of the internal workings of metamodels is out of scope for this report as the metamodels for AUTOSAR models are provided by the AUTOSAR Tool Platform User Group, or Artop for short [14], and are meant to be kept unmodified.

***Model Instances***

Model instances or core models can be built from metamodels. A model instance is an actual instance of a model whose components were defined by a metamodel. A model instance or core model can be used to then generate code in Java by default or in another language if generators are extended.

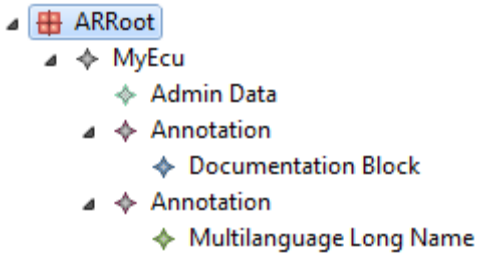


**Figure 3**. Simple AUTOSAR model instance.

Figure 3 shows a tree structure of a simple AUTOSAR model instance in Arctic Studio. AUTOSAR model components are sub classed `EObjects`. Figure 5 shows how the `ARPackage EObject` has one child `EObject` called MyEcu which has children of its own. MyEcu refers to an Electronic Control Unit used within a vehicle. This is visualized in a tree structure using a `TreeViewer` or the implementation provided by Arctic Studio's AUTOSAR

Navigator. The root node, which is not shown in the tree, is a resource and all the nodes in the tree are `EObjects` with either another `EObject` or `Resource` as its parent.

### 2.1.3 Arctic Studio

Arctic Studio is a complete IDE (Integrated Development Environment) built on top of Eclipse.

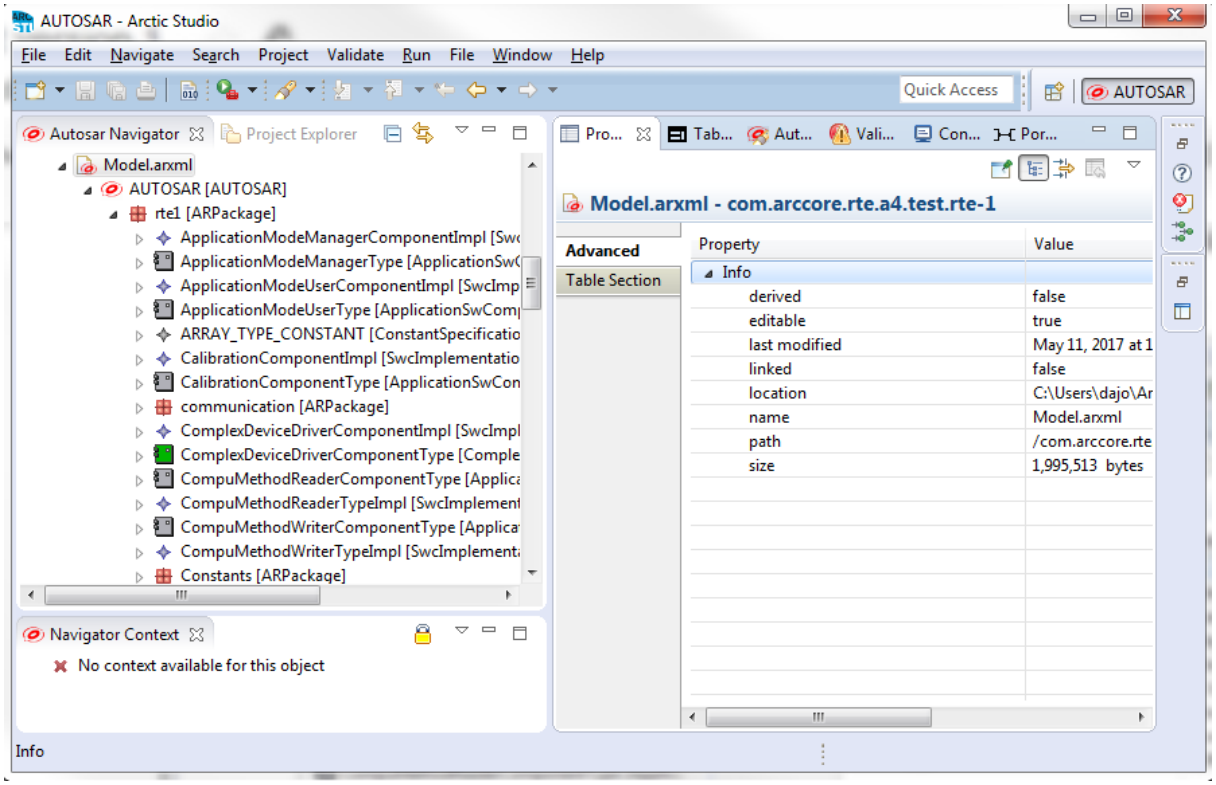


**Figure 4**. Screenshot of Arctic Studio GUI.

Figure 4 shows a screenshot of Arctic Studio Version 16.0.0. Arctic Studio supports C development for automotive embedded software based on the AUTOSAR standard. The AUTOSAR standard is utilized in automotive electrical architecture design. It is used by companies such as BMW, Bosch, Volkswagen and more and was created in 2002 [15]. Support for the C language is granted by Eclipse CDT and a custom-made environment based on msys in Arctic Studio [16]. Msys is intended to be used with MinGW, which is a minimalist development environment for Microsoft windows applications [17]. Msys is a collection of GNU tools that enable developers to build applications and programs that are dependent on UNIX tools [18]. This combination of tools and ARCCORE's philosophy of openness has led to Arctic Studio being a platform where developers can extend the IDE and add custom tools from the Eclipse marketplace or other sources to meet their complex needs.

Arctic Studio offers support for plugins with a built-in update-manager. Today Arctic Studio offers version control through plugins such as EGit that are provided with Eclipse, or lets customers handle version control with other tools such as mercurial and SVN [19]. These are however not designed for managing Arctic Studio's model instance semantics and can lead to errors or conflicts in merging project changes.

Important components of Arctic Studio that are used in this project are the AUTOSAR Navigator and workspace management components provided by Sphinx such as the `WorkspaceEditingDomainUtil` class.

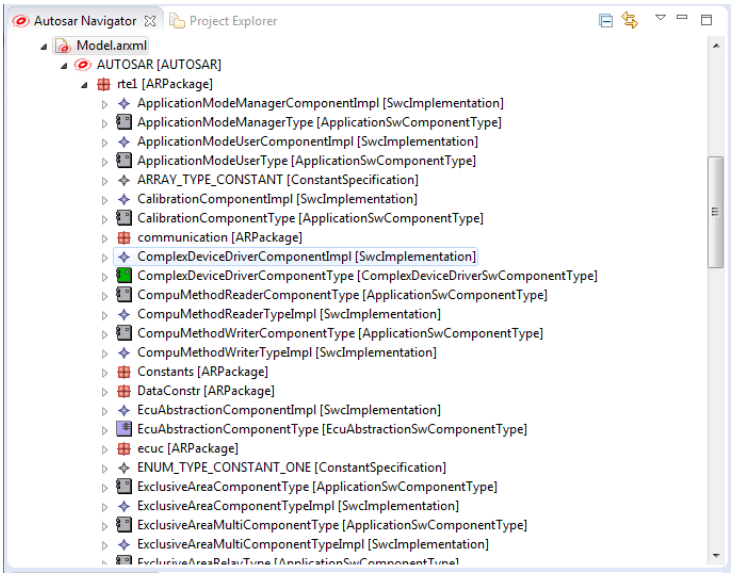


**Figure 5**. Screenshot of AUTOSAR Navigator.

Figure 5 shows a screenshot of the model editor plugin called AUTOSAR Navigator. AUTOSAR Navigator is a plugin that is used by Arctic Studio users to modify model instances and manage resources and files within an Eclipse Workspace. The `WorkspaceEditingDomainUtil` class is a sphinx class used to fetch editing domains from the managed workspace.

## 2.2 Version Control

When implementing or modifying a version control system it is vital to understand the version control strategies that exist today. The purpose of version control is to manage alterations to source code and other information found on computers [20]. Without version control, the communication overhead of handling multiple versions of code among teams grows dramatically and becomes a limiting factor in software development.

***Version Control Terminology***

The following is based on definitions provided by Kotwal et al. 2012 [21]. A repository is a database where version control systems store source change history. A commit refers to the creation of a change in a branch or master repository. A branch or folk refers to a copy of the master repository where commits to it do not modify the parent repository. A pull or update is the process of fetching changes performed by other developers to the branch a developer is working on to keep their copy of the code up to date. This is required if a developer wants to push their changes, which refers to placing local commits on the remote repository. Merging refers to the process of implementing the changes a developer has made into the repository's copy. Merging is often named when one is discussing about conflicts, or when multiple developers edit the same portion of source code and a decision of which change to be kept is made automatically or manually.

Version control can be split up into two broad categories based on their strategy for handling the merging and editing of different versions of code. These two categories include pessimistic and optimistic version control, and vary greatly in their complexity and flexibility.

### 2.2.1 Pessimistic Version Control

Pessimistic version control takes the approach of restricting access and editing of portions of a software project to maintain one version of the software repository. The way this is often done is by using locks.

***Locks***

The theory of using locks to protect versions of documents and data is a concept utilized in other technologies other than version control as well, such as concurrency control in shared databases [22]. The idea is both simple to understand and implement. Locking involves giving exclusive access to a single developer a certain portion of a project and blocking others from making changes to the same portion of code [23]. The level of locking can be chosen to be more or less granular depending on how pessimistic the version control system is. One can lock entire files or lock on a block-level.

***Advantages***

Utilizing a lock-mechanism for version control eliminates the problem of merge conflicts in a textual-context if one ignores dependencies or semantics between files. By only maintaining one version of code between all developers, this approach begins and ends with the locking-mechanism and doesn't support multiple versions of a code base. This is one method for implementing a simple version handling system where multiple developers can work on the same project at the same time if it is not desirable for multiple developers to work on the same portions of the project concurrently.

***Disadvantages***

The first disadvantage of pessimistic version control is the lack of support for multiple copies of a source project and a lack of supporting edits in the same portion of code at the same time. Locking-mechanisms are not flexible. Other disadvantages of a locking-scheme are not always obvious until a set of developers experience the need to handle conflicting code. In comparison to optimistic version control, pessimistic version control handles very little of the conflicts that can occur in the editing of model-based projects. Many conflicts must be addressed and fixed manually after edits have been performed. With a lock, two developers may lock their own model and make changes in their separate models. When these locks are released, it is true that the individual models themselves do not have conflicts, however the semantics between the models or any dependencies may now suddenly conflict [3]. If a locking system does not consider dependencies or semantics between models, then it is likely that conflicts will have to be later handled manually after edits are made.

### 2.2.2 Optimistic Version Control

Optimistic version control takes a different approach to handling versions of code. Developers can work in a project without restriction and the version control system handles conflicts in versions after they are introduced [24]. Optimistic VCS implement difference detection, conflict handling and merging. It is worth noting that difference detection, conflict handling and merging is very tightly coupled. The following overview attempts to organize these three topics separately, but as a reader it is vital to understand that they are often implemented together and are not isolated processes.

***Merging Methods and Conflict Handling***

Merging methods can be categorized based on three criteria [25]. The first one is if a system uses a two-way or three-way merging process. VC systems can also be categorized based on if merges are textual, syntactic, or semantic-based. Finally, a third categorization of methods

include if they are state-based, change-based or operation-based [25]. All three categorization criteria are presented below.

***Two-way and Three-way Merging***

This categorization is based on the number of copies being compared to one another. Two-way merging is the process of merging two versions of software without taking into account a common previous parent version of the code [25]. Two-way merging is limited in the sense that additions and deletions of new versions cannot be detected as this merging technique does not have access to previous versions of the model [3]. Two-way merging has no means to know if a change in the source code is an addition, removal or edit [25]. Three-way merging is an expansion of two-way merging where one takes into account the original parent version of the code that new changes are added to [26]. By knowing the previous state of the source, three-way merging overcomes the shortcomings of two-way merging.

***Textual-, Syntactic-, and Semantic-based Merging***

This categorization is based on how the contents of copies are compared to one another, as well as how software artifacts are represented in a project.

Textual- or line-based merging is the most common way to manage merging in version control systems. In textual-based merging, software is inspected line-by-line and changes such as insertions of new lines, deletions or modifications are found between compared versions of code [25]. Two edits to the same line results in the VCS prompting developers to select a single version of the code. Textual-based merging is appropriate in most software projects, yet it lacks in detecting syntactic or semantic changes.

Syntactic-based merging is an extension of textual-based merging yet is not as complex or powerful as semantic-based merging [27]. It takes into account the syntax of the program code, such as whitespace and language-specific declarations. As such, adding or removing whitespace or comments may not qualify as a conflict in the code. This can allow two edits of the same line to be merged together even if information such as white space was altered. Dependency conflicts can go undetected in syntactic merging while the syntax of these files may be correct [25].

Semantic merging takes into account program syntax as well as aspects such as variable declaration and dependencies between files. Semantic-based merging was still being researched in 1995 [27]. Today, much progress has been done in semantic-based merging and there exists commercial products that utilize this method of merging.

Semantic-based merging is most appropriate for the merging of modelling artifacts as textual and syntactic merging methods miss the dependencies between models. Semantic-based merging can better detect and handle conflicts in software versions of models, and offloads much of the work of managing this manually from the developers' shoulders.

***State- and Operation-based Merging***

This categorization is based on how the VCS handles or takes into account the revision history of the software repository. State-based merging does not take into account the alteration history of the project and only looks at the difference between the origin repository and the copy to be merged. State-based merging is simpler than the other two forms and all two-way merge methods are also state-based [25].

Operation-based merging is a form of change-based merging that takes into account the revision history of a software project when performing merges. As the "operation" name hints to, this form of merging views each change in the source as an operation. As new operations are performed, this form of merging can check if a new operation and an old one change or implement a specific portion of code in incorrect ways [25]. An example of a conflict would be if a new operation were to change the interface to a function that was used somewhere else in the project as documented by an older operation. Operation-based merging stores a history of changes to a project which facilitates the ability to have rollback features as well. It is possible to use operation-based merging to detect syntactic and some semantic conflicts [25]. This explains why the VCS called EMFStore, which is described later in section 2.2.4 utilizes operation-based merging to handle semantic changes between modelling versions.

Based on the above information, the project solution will utilize some form of operation-based merging to better handle semantics between models.

***Artifact-level Difference Detection***

Many traditional version control systems are textual-based and as previously mentioned, are ill-suited for tracking changes in modelling artifacts [28]. Version control systems such as SVN, RCS, GIT, Mercurial and CVS inspect changes on a textual and on a line-basis [3]. If models are represented in XML, then it is possible to serialize these models to files using XMI and thereby make it possible to use traditional versioning tools.

However, the reason why these solutions are ill-suited is because text-based version control does not consider the semantics of models [3]. Semantics in models are what make models related to one another. These include dependencies and graph connections between models. Even if it is possible to serialize models to files, by not taking into account the semantics of models, changes to the relation or communication between models may not be detected by traditional difference tools [29]. In addition, XML files are also very difficult to read and may become entirely regenerated after small model edits, which poses a limitation to their usefulness.

### 2.2.3 Centralized and Decentralized Version Control

Version control systems can either be centralized or decentralized in their repository architecture [3].

***Centralized***

Centralized version control systems are characterized of having a single repository that developers make changes to with the process of a checkout [2]. Developers create branches or copies of the main repository code, where changes can be made to add features or fix bugs without modifying the main repository. Only authorized developers are permitted to commit changes to the main repository.

***Decentralized***

Decentralized version control systems do not require a single master repository. Each branch or fork is considered its own repository and it is fully possible for developers to perform branches to each branch as well [2]. Each branch/fork can be considered its own project. The benefit of this approach is that in the case of a parent repository disappearing due to a network loss or disk failure, each branch of that repository will continue to exist as its own entity.

Decentralized version control is a newer form of version control compared to centralized version control and many software projects have migrated to the newer form [2]. A study conducted by Brian et al. in 2010 explored the reasoning behind migrating to the newer model of version control. One difficulty experienced by some organizations using centralized version control in the study was the difficulty of managing large numbers of branches to a single repository.

The reasons given for migrating projects to distributed version control systems was to give each developer the ability to create their own branch repository and make changes to the code base as seen fit instead of having to checking out upward. Distributed version control systems were also perceived as having better dependency detection during merges than the existing centralized systems. Finally, having the ability to commit changes to a repository locally in the event of a network failure was a desirable feature for developers and organizations.

It may be therefore desirable that the solution implemented in Arctic Studio have support for utilizing a local repository store as well as a remote one.

### 2.2.4 Existing EMF Model-based Version Control Systems

What follows is an overview of existing EMF version control options. This report restricts itself to EMF-specific options as it appears that general-purpose modelling version control systems such as the one proposed by Altmanninger [28] are utilized as research platforms and are not widely available.

***EMFStore***

EMFStore is an existing system for implementing version control for Eclipse Modeling Framework model instances. EMFStore is a VCS that provides change tracking, conflict detection, merging, and versioning of models. Changes are detected and tracked at the model-level. EMFStore is composed of a server and clients, where the server runs as a standalone application and handles conflict detection. Users are responsible for committing changes, updates and merging different versions of the project during a conflict. The sample client is also usually integrated into the Eclipse IDE and provides a user interface to perform these named activities. The EMFStore project offers an API so that developers can integrate various portions of the project into their own version control solutions.

*Change Tracking*

EMFStore uses the operation-based approach to provide change tracking in the project. Operations are tracked on a model-element level. The project categorizes operations into attribute, reference, create/delete, and composite operations [30].

*Version Model*

To store different versions of a project, EMFStore uses a version model that is represented as a tree where previous nodes are tied together based on revision history. Every version in the tree contains a change package and contains all operations that altered the previous version into the current version. Other information is also stored, such as timestamps, log messages and who performed the merge. EMFStore utilizes the optimistic version control approach [30].

*Conflict Handling*

EMFStore uses operation-based conflict detection and resolution. This is a built-in fine-grained conflict detection strategy that may be changed by the user which can implement their own detection strategy into EMFStore [30]. This makes EMFStore especially interesting as it can

potentially be customized to meet the needs of Arctic Studio. Before committing new operations, a user updates their project to the newest version. The new operations have yet to be committed and thus no conflicts can occur in this stage. All operations can be examined by the user and conflicts are handled by marking each operation as either rejected or accepted [30]. By marking a commit as rejected from the server, a developer stops that commit from being added to the result list. Some commits are labeled as non-negotiable, and these are never ignored. Once remote operations are fetched, local ones are also labeled as either rejected or accepted. The accepted operations are also added to the result list. Once the result list has been completed, the system applies all of them onto the project both locally and remotely. The order of these operations is preserved as commits are often built upon one another.

***EMF Compare***

EMF Compare is a merging and comparison facility that support all kinds of EMF models. It is a framework where one can reuse comparison instances of models and operates as a tool that can detect and visualize differences between models. Since EMF Compare is integrated with the Eclipse Team API, it can be used to collaborate on models that use CVS, SVN and GIT as their repository store. EMF Compare is designed with scalability in mind which lets it compare large fragmented models. It does this by only loading the fragments of a project that have been changed and presents them visually for the user. Only necessary fragments of models are loaded when displaying a compare, and this lets EMF Compare to scale well with large models [31]. This statement was contested by ARCCORE developers who have experienced that EMF Compare does not handle the most complex models very well.

EMF Compare is strictly a visual tool for merging and can be combined with some form of repository control to implement model-based version control of EMF-based models. ARCCORE AB verified that a customized version of EMF Compare exists within Arctic Studio, yet its performance was lacking and thus EMF Compare will not be used in this project.

***Eclipse CDO***

Eclipse CDO is a model-based version control repository with support for EMF models. Changes are detected on a model-level and not on a textual level [32]. The project supports object graphs of different sizes and is based on Java. Eclipse CDO requires migrating EMF generator models to CDO enabled generator models [33]. Generator models specify where to save generated code for metamodels and include additional information such as what names packages should be set to and more [7]. As metamodels are provided and managed externally by Artop instead of ARCCORE in Arctic Studio, Eclipse CDO is labeled as not particularly viable for this project.

***Eclipse CHE***

Eclipse CHE was mentioned by ARCCORE early at the start of the project as a possible direction to take to implement multi-user editing in Arctic Studio. Eclipse CHE is a new web-based IDE that uses a web server where developers can connect and write their code without the need to install a local IDE [34]. It currently has limited support for various programming languages and there are plans on implementing support for version control at a later date.

The problem with Eclipse CHE is that its version control system is incomplete, all EMF plugins would need to be rewritten to be made compatible with the new platform, and CHE lacks support for the programming language used in Arctic Studio. Until Eclipse CHE becomes more feature-complete, we do not envision it to be worth testing in this project.

# 3. Method

To effectively answer the research questions, an understanding of EMFStore and Eclipse Plugin Development needed to be acquired. Much of this is documented in the theory section of the report. The limitations of EMFStore need to be identified both in terms of conflict detection as well as its ability to handle AUTOSAR model instances. EMFStore's conflict limitations will be tested with a set of conflict scenarios on an AUTOSAR model instance to either verify that the project does or does not detect conflicts in those scenarios.

First, a method for synchronizing changes to a model instance in Arctic Studio to a versioned project in EMFStore will be created based on reading EMFStore and EMF-specific Javadoc documentation [35].

After constructing a prototype, a controlled confirmatory experiment will be performed and the final solution will be evaluated based on if defined merge conflict inputs are handled. We do our best to follow the guidelines on formal experiments presented by [36]. How the conflict inputs will be defined is described shortly below.

An evaluation of the final solution will be based on how well the solution handles the list of potential merge conflicts on an AUTOSAR model instance. Finally, the solution will be presented to developers at ARCCORE AB and an evaluation interview will be performed based on a semi structured qualitative interview based on suggestions by [37] to assess the final solution.

## 3.1 Identification of EMF Merge Conflicts

There are two primary questions we aim to answer before working on the project implementation.

The first is:
*How are we going to figure out how Arctic Studio EMF models work and look like?*

In order to understand how Arctic Studio EMF models work, we will examine the files used to store the model artifacts, read documentation for both EMF models and Arctic Studio, and look over the source code for how models are built. In addition, we will create a small model in Arctic Studio to verify our understanding of how models are represented in files. This is essential for understanding what semantics we need to address in version control.

If ARCCORE is open to the idea, we will also request a small real-world sample project to base our testing of merge conflicts on. This is so that we can base our tests on a realistic model.

The second question is:
*How are we going to create a list of possible merge conflicts for these existing models?*

First, we will conduct an initial interview to ask ARCCORE developers for specific problems that they face today when they perform a merge of model instances. Specifically, we will ask what kinds of conflicts are detected with their current tools that are not actually conflicts, and result in unnecessary manual merge work as well as changes or conflicts to models that currently go undetected or are difficult to deal with. With an understanding of existing and well-known issues, we can create a list of test scenarios where the sample project is modified to create such merge conflicts. We assume that much of experienced issues have to do with the

existing version control systems not addressing semantic relationships between models and instead creating difficult merge situations where diffs are performed on a line by line basis in text-based VCS.

### 3.1.1 Preliminary Merge Conflict Scenarios

Three-way delta merging is described as desirable or is utilized by [38] and [30] to better detect conflicts. Thus, we assume that the project will either implement three-way merging or extend existing tools that utilize this method. The types of model modifications that need to be supported by the final solution include support for merging insertions, deletions, updates and moves of model elements. This is described as features that are desirable to be supported by related research on the implementation of version control of EMF models [38]. When exporting, or saving the model instance for usage within Arctic Studio, the resulting serialized form must be formatted appropriately.

The following examples are based on details presented in section 2.2.2.

Example of a syntactic conflict that need to be handled:

- One case that we want to ignore is the introduction or deletion of additional whitespace in programs. A VC solution that interprets changes in whitespace as textual changes can result in many unnecessary merge conflicts that do not actually affect the merged model. This requires either adding or making sure that the solutions understand whitespace syntax in Arctic Studio models.

The resulting model needs to also be semantically correct, and the VC system should give the user feedback when semantics have been changed in a merge operation. Conflicts were tested on a model instance level and not on model instances' metamodels.

Examples of semantic conflicts that need to be handled:

- Change conflict: Changes to same feature or attribute in an EObject should result in a conflict detection by EMFStore.
- Deletion conflict: Two developers are working on the same model instance. One user deletes an EObject and commits their changes to the serve. The second user adds an Attribute to the same deleted EObject and commits their changes. This commit should be detected as incompatible and the user should be prompted appropriately.

## 3.2 Initial Interview

An interview was conducted to determine what tools users of Arctic Studio currently use for versioning of AUTOSAR model instances. The purpose of the interview was to also gather information about users' experiences of versioning model instances, identify aspects that are missing in Arctic Studio and to find out what users prefer in terms of design choices.

### 3.2.1 Interview Format

The interview was a standardized and semi-structured interview. Each interviewee was asked the same questions about their experiences with version control in Arctic Studio. Some questions were structured, others were open-ended. All questions and answers are located in Appendix A.

A standardized approach was chosen to better fit the medium that the interview was conducted through. All interviews were sent out as emails and filled out by the participants through a

Google Form. An in-person interview was not possible due to the interviewees living in different physical locations.

### 3.2.1 Interview Questions

The questions that will be used in the initial interview and the reasoning behind them are listed below.

***First Question:*** *What form of version control do you or your company use to track changes to AUTOSAR models in Arctic Studio?*

The goal of this question is to get an understanding of the current version control workflow developers are using today.

***Second Question:*** *What is your typical method for handling merge conflicts on model projects through Arctic Studio?*

This question serves both to understand how developers currently perform merges today with as well as to understand the ease of use or quality of the current merge workflow.

***Third Question:*** *Would you be open to using a specialized version control system in Arctic Studio that differs from your current version control system?*

The goal of this question is to get a better understanding if developers were open to using an entirely new VCS or if such a new tool would simply not be used if implemented.

***Fourth and Fifth Question:*** *If you use a textual-based version control system such as GIT, SVN, etc, are there scenarios where those systems detect large changes to files and Arctic Studio models when only small details were in fact modified?*
*Have you experienced any scenarios when two developers made changes to a model and a merge was performed that resulted in a broken model? If you have, please explain the scenario that led to this problem.*

During the literature study portion of the bachelor thesis and while reading shortcomings of traditional text-based version control systems, we decided that we needed to test if certain specific undesirable scenarios could occur while working with versioning of model instances. These two questions will be used to check if two undesirable scenarios have occurred for ARC-CORE developers. Specifically, if small edits to model instances resulted in large changes being detected in text-based VC systems, or if concurrent changes to a model resulted in a broken model instance.

***Sixth Question:*** *Have you experienced any scenarios where two developers made changes to a model and a merge was performed that resulted in one developer's edits being entirely ignored?*

This question is designed to test how developers use their current VCS. If merges are not thoroughly handled, then a remote copy of a model instance may be overwritten during a merge.

***Seventh Question:*** *Have you experienced any scenarios where your version control system detects a change even when you haven't modified a model in Arctic Studio?*

The goal of this question is to see if serialization occurs automatically in Arctic Studio without user input and if the model instance files can change their textual-representation even if developers make no changes to them.

***Eighth Question:*** *If ARCCORE were to work on a custom version control system, would you prefer a conflict handling system that lets you keep your current version control system as a back-end or are you open to using an entirely new version control system built-in to Arctic Studio?*

This question is asked to see if a new VCS would be adopted by developers if it were integrated into Arctic Studio.

***Ninth Question:*** *Do you have any other experiences with version control of AUTOSAR models/projects in Arctic Studio that were less than ideal? If so, can you share some of your experiences?*

This question is designed to be kept open-ended to allow the participants to mention other problems they have had when versioning AUTOSAR models in Arctic Studio.

***Tenth and Eleventh Question:*** *Do you or your co-workers prefer using a graphical user interface or a command-line interface when managing version control of projects?*
*If we were to implement a graphical interface for version control in Arctic Studio, would you prefer it to be integrated as a tab or "view" in Arctic Studio, or as a separate window that pops-up over the Arctic Studio interface when a commit or pull is to be performed?*

The purpose of this question is to decide whether it is worth implementing both a command-line interface as well as a graphical user interface for the plugin.

## 3.3 Implementation

There are likely to be several ways to implement the final solution. Different implementation alternatives will be listed up in a priority order based on the least amount of modifications needed to get them to work with AUTOSAR's existing modelling semantics. This way at least one prototype can be guaranteed to be delivered to ARCCORE AB.

### 3.3.1 Version Control Plugin

First, we will begin by installing and integrating EMFStore to get it to run within Arctic Studio without regard for merge conflicts. The solution will be implemented in Java as an Eclipse plugin to Arctic Studio. The version of Arctic Studio that will be used in this project is version 16.0.0. To boost replicability, we have attached our source changes at the end of the report.

After versioning of model instances appear to work on a basic level in EMFStore, it is time to test the solution according to the conflict scenarios gathered from section 3.1.

## 3.4 Evaluation

It is important to define a structured and systematic method of evaluating the quality of the final solutions. To thoroughly evaluate the project solution, both programmatic conflict scenario tests and in-person interviews will be performed.

### 3.4.1 Conflict Scenario Testing

In section 3.1 we defined a list of merge conflicts that can occur in Arctic Studio's models. The solution will be tested with the same set of scenarios that create the defined merge conflict. Evaluation will be based on if the solution first detects and then handles the merge conflict in a desirable way. It is desirable that the solution first detects the presence of a conflict, and then prompts the user to select which version to keep.

### 3.4.2 Evaluation Interview

To assess the user-friendliness and usefulness of the final solution, an evaluation interview will be performed on a subset of developers at ARCCORE AB. The interview will be conducted by observing and recording how developers first react to the solution as well as how easily they can conduct specific tasks such as commit, update and checkout in the solution. The developers will be asked to create a specific conflict scenario and observations will be recorded. Finally, when the developers understand the interface, they will be free to perform actions as desired and any comments and thoughts will be recorded. Based on this evaluation interview, the project will be either improved or concerns will be marked for future work.

There will be a bias in the evaluation as developers at ARCCORE are likely to be experts in using and modifying Arctic Studio tools, however the main point of the bachelor thesis is to evaluate if integrating EMFStore into Arctic Studio is possible and to identify any limitations of the solution when applied to AUTOSAR models, and thus we believe that the selection of interviewees is adequate for evaluating the final product.

***Interview Questions***

The evaluation interview can be considered qualitative as its purpose is to gather interviewees' experiences in using the project solution.

First, interviewees will be given the same explanation as to how the interface works and the idea behind using model-based version control instead of text-based version control systems on AUTOSAR model instances

Next, interviewees will be asked to perform specific tasks in the regular interface. These tasks include:

1. Open an existing model instance file in the model version control plugin and add another AUTOSAR `ARPackage` to the root level of the model instance using the GUI.
2. Commit this change to the EMFStore local server and provide a commit message.
3. View the change history and verify if the commit went through or not.
4. Checkout an existing project on the EMFStore local server (For the interview, an existing remote project was added ahead of time).
5. Inspect the new project's content.
6. Export this model to an ARXML file and open it in Arctic Studio (Information on where the file is saved was given).

This is the general workflow or actions that users may have to perform quite often when using the model version control plugin. During each of the tasks, the interviewer will record any comments that the interviewee had, answer any questions, and record how the user attempted to perform each task and if there were any difficulties or patterns in using the interface. Some tasks such as Task 4 are designed to see if certain UI elements such as the history view tab are adequate in giving user feedback that a commit has been performed or not.

Interviewees will be given the opportunity to perform their own tasks in the model version control plugin and give feedback after tasks are complete. Once the interviewees no longer have any more comments, they will be given access to the additional conflict simulation tab to test out conflict scenarios.

Tasks that will be requested to be performed:

1. Invoke the first pre-existing conflict scenario test. Progress through any GUI elements that appear and select to accept remote changes.
2. Perform the same first conflict scenario test. Progress through any GUI elements that appear and select to accept local changes.
3. Attempt to create a conflict of your own design from scratch using the conflict tab.

The first two tasks are structured to give the user an idea of how a specific conflict is handled by EMFStore. The third task is open-ended and designed to see how interviewees react to the plugin's handling of their own conflict scenarios. After these above tasks are complete, the interviewer will ask specific questions that can be categorized into two categories: behaviour and opinion questions, per [36].

The two behaviour questions that will be asked are:

1. Can you explain what information EMFStore presented to you when a conflict occurred?
2. Can you describe what you did to resolve a conflict?

The opinion questions that will be asked are:

1. What was the most challenging task to perform? Why was this the most challenging one?
2. What was an aspect that was unclear in the user interface?
3. If you could choose, what modifications would you make to the workflow of versioning model instances in the solution plugin?
4. If you could choose, what modifications would you make to the user interface in the plugin solution?
5. Do you interpret EMFStore as capable or lacking in versioning of AUTOSAR model instances? In what aspects was it lacking?

# 4. Results

What follows is the result of the method named above with reflection over the conducted interviews as well as technical details of our plugin implementation.

## 4.1 Initial Interview

It was important that the resulting prototype solution be useful for developers that use Arctic Studio both within ARCCORE and externally as customers. The purpose of this interview was to identify the current workflow for version control of model instances by developers, identify problems with their current workflow and collect a few opinions on user interface design. The interviewees were composed of developers within the ARCCORE company. A total of seven developers participated. Some questions about current version control were designed to verify or disprove assumed current problems with using textual-based systems. Others were designed to be open ended so that developers could give feedback that could aid in the project.

### 4.1.1 Initial Interview Result

The full list of questions and answers to the initial interview can be found in Appendix A. Names, contact information and personally-identifiable information have been removed to ensure participants' privacy.

***First Question:*** *What form of version control do you or your company use to track changes to AUTOSAR models in Arctic Studio?*

Six of seven participants currently use some form GIT while one person also uses TortoiseSvn. TortoiseSvn provides a graphical user interface for Apache Subversion, which is a text-based version control system. One can conclude that GIT is broadly used within the test-group. The biggest takeaway is that none of the participants utilize a modelling version control system in Arctic Studio. Everyone uses a textual-based VCS, and thus our solution would offer something new to existing users.

***Second Question:*** *What is your typical method for handling merge conflicts on model projects through Arctic Studio?*

The answers show that three methods are used heavily when merging: manually merging with text based tools, merging with the existing EMF Compare tool, or not at all. In some cases, they avoid the need of merging by just letting one developer work with a model instance at a time. These answers conclude that the merging of model instances in Arctic Studio is difficult and a solution to this is needed.

***Third Question:*** *Would you be open to using a specialized version control system in Arctic Studio that differs from your current version control system?*

The answer concluded that the majority were open to try out a new system but only if it were reliable.

***Fourth and Fifth Question:*** *If you use a textual-based version control system such as GIT, SVN, etc, are there scenarios where those systems detect large changes to files and Arctic Studio models when only small details were in fact modified?*

*Have you experienced any scenarios when two developers made changes to a model and a merge was performed that resulted in a broken model? If you have, please explain the scenario that led to this problem.*

For both questions, six of seven participants answered that they have experienced these scenarios or were unsure.

***Sixth Question:*** *Have you experienced any scenarios where two developers made changes to a model and a merge was performed that resulted in one developer's edits being entirely ignored?*

This question could have been improved to be more specific, as it resulted in yes/no answers that do not necessarily explain important information of how the VCS was used to create the scenario. Answers were mixed and it is hard to come to a direct conclusion.

***Seventh Question:*** *Have you experienced any scenarios where your version control system detects a change even when you haven't modified a model in Arctic Studio?*

The answer to this question was no. This was later verified when working in Arctic Studio. Model instances are stored in memory and modifications are only saved to the hard drive when the user manually saves changes in the AUTOSAR navigator or through a text-editor. The final implementation for saving or exporting model instances in the model version control plugin can be viewed in Appendix B, section 5.

***Eighth Question:*** *If ARCCORE were to work on a custom version control system, would you prefer a conflict handling system that lets you keep your current version control system as a back-end or are you open to using an entirely new version control system built-in to Arctic Studio?*

The answers point to using the new VCS for model-instances if it works well, and then traditional text-based VC for everything else. This answer works well in the scope of EMFStore, as EMFStore is ideal for versioning model-instance changes while other text-based systems work well at versioning code and text-documents. According to the answers, only model instances will be versioned by the final solution.

***Ninth Question:*** *Do you have any other experiences with version control of AUTOSAR models/projects in Arctic Studio that were less than ideal? If so, can you share some of your experiences?*

GIT and SVN were mentioned as lacking in version control support of models and required manual merge conflict handling by some of the participants. One participant mentioned that a competitor product by the company Vector has a merge tool that works fairly well. The authors of this report did not have access to a copy of Vector’s software. One participant mentioned that s/he always keeps remote changes and then pushes updates to GIT. The remaining participants had no comments.

***Tenth and Eleventh Question:*** *Do you or your co-workers prefer using a graphical user interface or a command-line interface when managing version control of projects?*
*If we were to implement a graphical interface for version control in Arctic Studio, would you prefer it to be integrated as a tab or "view" in Arctic Studio, or as a separate window that pops-up over the Arctic Studio interface when a commit or pull is to be performed?*

These two questions became obsolete when we started working with integrating EMFStore into Arctic Studio. The purpose of the bachelor thesis was to create an integrated prototype of EMFStore in Arctic Studio, and we concluded that a basic graphical interface would be quicker to implement and thus give more time to working with improving the technical limitations of version control of AUTOSAR models in EMFStore. These questions give feedback for future work on improving the user-friendliness of the implemented tool.

Many developers expressed the desire to use the new system via the command line while others preferred a graphical user interface.

The eleventh question could have been improved by explaining that EMFStore would be used as an integrated plugin to Arctic Studio. This information was however not available when the interview was conducted.

## 4.2 User Interface Design Decisions

Special consideration was made to design the project based on the answers to the interview. Answers to the interview indicated that some developers desired a version control solution to have a separate user interface that could be placed over Arctic Studio. Others expressed that it would be desirable to have the solution as an integrated component in the IDE. See Appendix A for detailed and unmodified answers.

### 4.2.1 Eclipse View

Based on the above input, an Eclipse `View` was selected as the primary GUI component of the project, as it can be attached to various positions in the Eclipse Workbench as well as detached and dragged to a separate screen if a developer chose to do so.

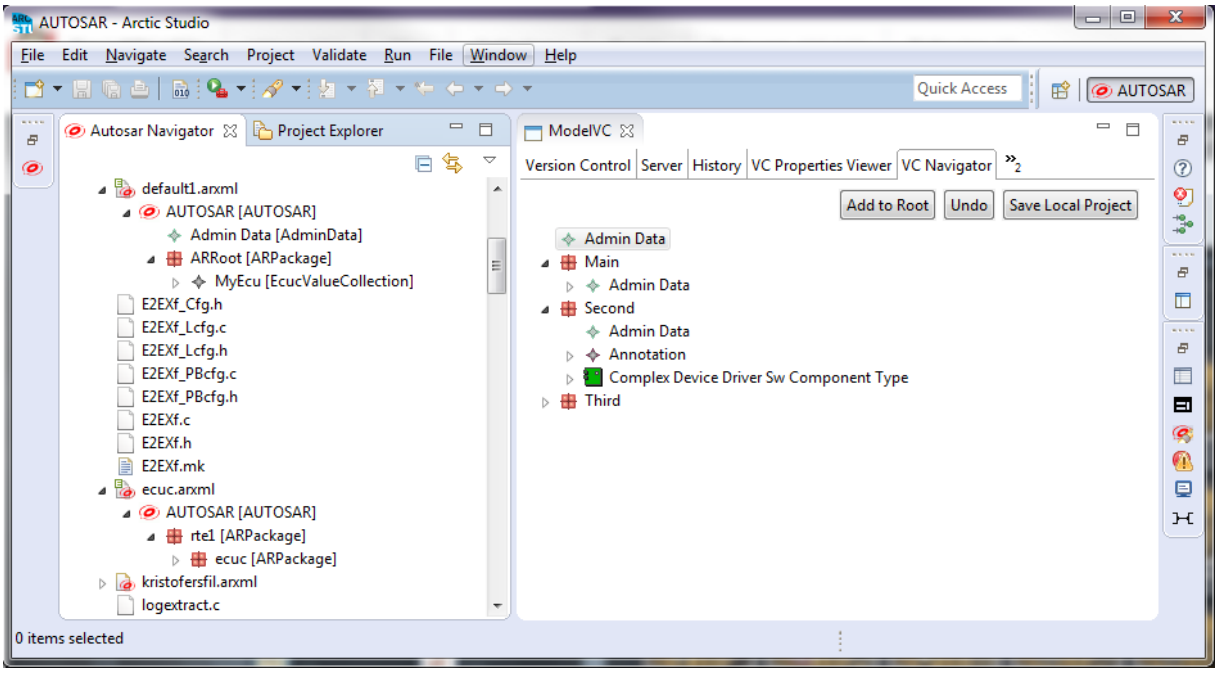


**Figure 6**. The model version control view integrated in Arctic Studio.

The model version control view implemented in this report is located in the right half of figure 6, and is similar in style to other existing user interface components.

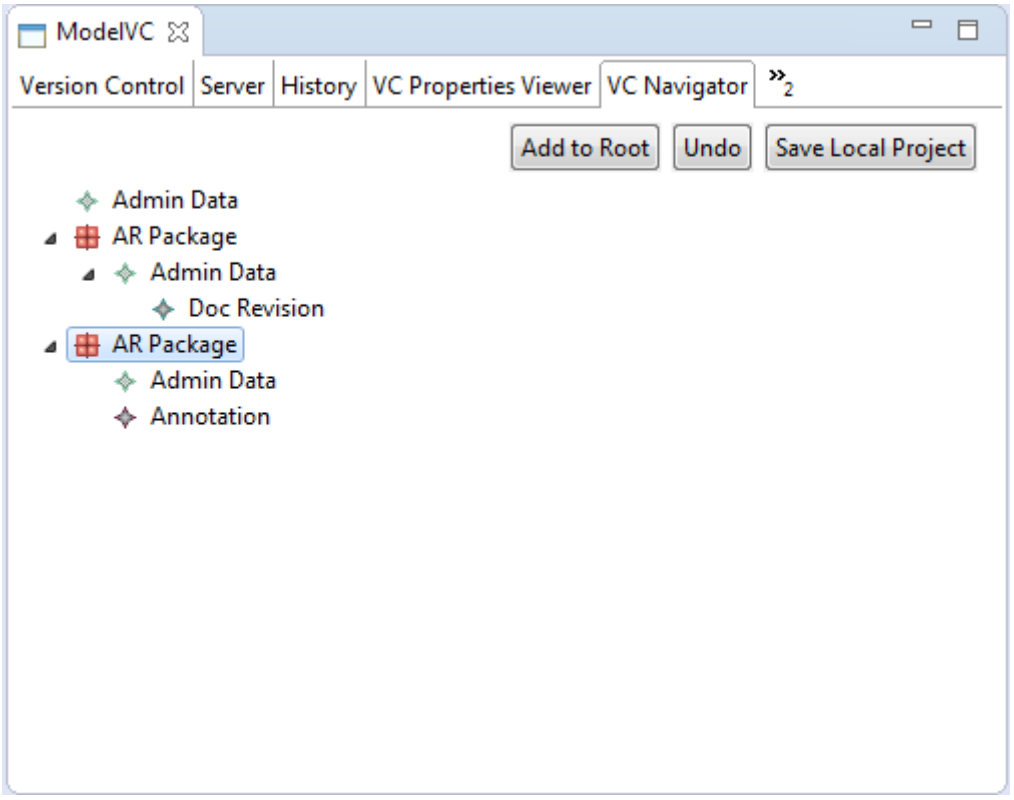


**Figure 7**. The view dragged to a separate monitor.

Figure 7 shows the same plugin view dragged outside of the Eclipse GUI.

`Views` are graphical components that can be viewed within the Eclipse workbench. `Views` are registered via the Eclipse workbench's view extension point called `org.eclipse.ui.views` [39]. `Views` must be registered so that the Workbench is aware of the extension. The workbench also tracks the position of registered views so that the previous state of view layouts can be persisted into the next Eclipse start up [40].

```
<extension
       point="org.eclipse.ui.views">
       <view
              name="VCModelView"
              class="com.arccore.versioncontrol.model.VCModelView"
              id="com.arccore.versioncontrol.VCModelView.view">
       </view>
</extension>
```

**Figure 8**. VCModelView extension point registration.

The XML mark up in the project plugin's manifest file that was required to register the plugin on the views extension point is shown in figure 8.

### 4.2.2 Tabular Organization

Within the plugin view, an SWT `TabFolder` was chosen to represent different portions of the model version control plugin. To aid both development of the plugin as well as enable

end-users to use the plugin productively, several features were implemented. Each set of functionality was organized into a separate `Tab` within the `TabFolder`.

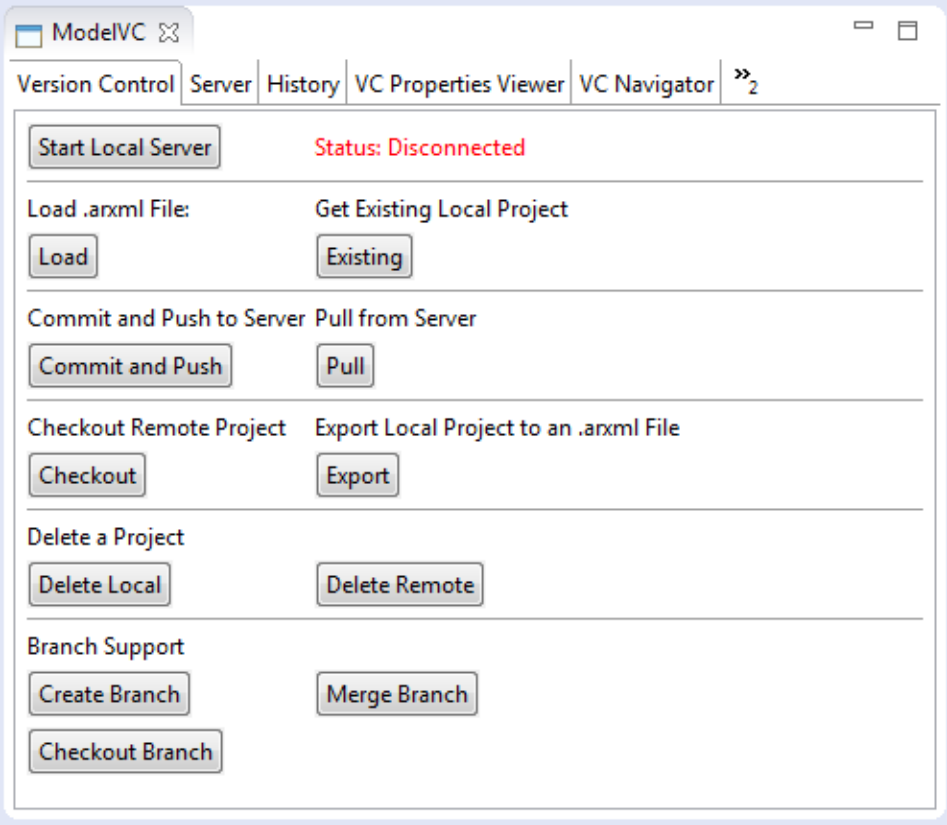


**Figure 9**. Version Control Tab.

The version control tab shown in figure 9 is the main tab central to the model version control plugin's functionality. Here the user can start their local EMFStore server, view their connection status to that server, load a model instance into the plugin from file and commit any changes to the server. Commits are first attempted by `VCEMFStoreManager`. If a commit failed, then it will attempt to fetch updates from the server and reapply the commit. Dialogs for handling conflicts, performing updates before a commit, and to see changes to commit are all provided by EMFStore. See Appendix B, section 9. for the commit implementation used in the project.

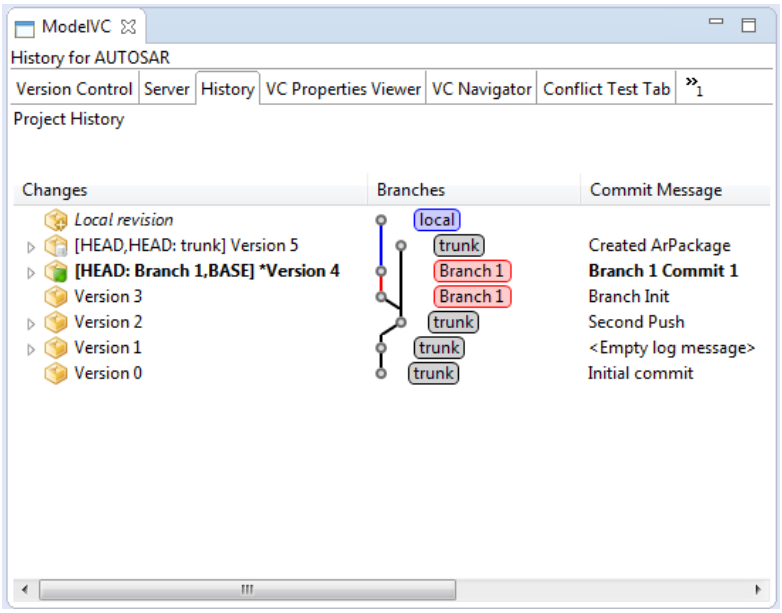


**Figure 10**. History Tab.

The history tab, shown in figure 10, utilizes the existing `HistoryBrowserView` available from the EMFStore project. Commit history including branches of the current repository are illustrated by the `HistoryBrowserView`. Each commit can be expanded to show information of what model elements or attributes were modified.

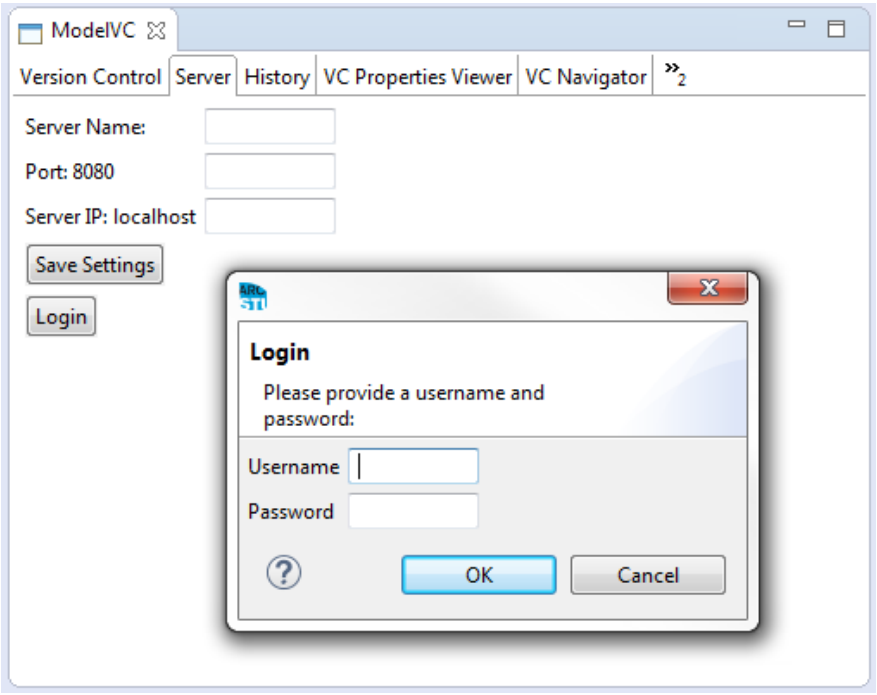


**Figure 11**. Server Tab.

The server tab in Figure 11 acts as a placeholder at this stage. This location is for future work with a remote server. It contains the functionality to select IP address, port and a server name. It also contains a button which enables the user to log into the selected IP address and port. The current selected IP address and port is shown beside the input fields for these variables. The login behind the server tab is not implemented, however it is kept so that the model version control plugin can be expanded on in the future.

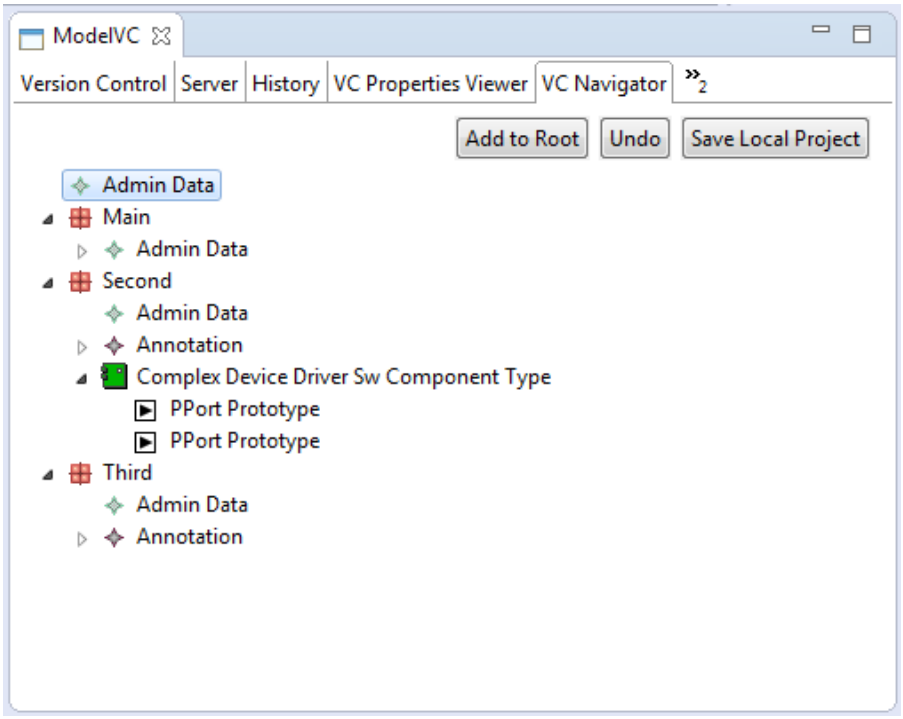


**Figure 12**. VC Navigator Tab.

The VC Navigator tab in figure 12 is essential for informing the user what model instance is loaded into EMFStore. It also served as a verification point during the development process to verify that model components were successfully added. When separate `ResourceSets` were used in the project, adding an `EObject` from one `ResourceSet` to the EMFStore `ESLocalProjectImpl` (`ESLocalProject` is used to store the locally versioned project in EMFStore) would result in that particular `EObject` and its contained `EObjects` disappearing from the original `Resource` and `ResourceSet`. The two-separate domain solution uses `EcoreUtil` helper class to copy the `EObject` when moving it from one resource to another. Solving the problem with it disappearing in the main resource.

When the same editing domain and resource set was successfully shared between the Eclipse Workspace and EMFStore, then the model instance tab automatically reacts to changes to the model instance in the AUTOSAR navigator as they both display the same `EObject` hierarchy. The model instance tab uses a SWT `TreeViewer` to visualize the model instance hierarchy of `EObjects`. `TreeViewers` are used to display model instance component hierarchies within the Eclipse Workbench. Adding the root `EObject` to the `TreeViewer` is enough, as the viewer calls `getText()` and `getImage()` methods for all child `EObjects` to populate the viewer with content [7].

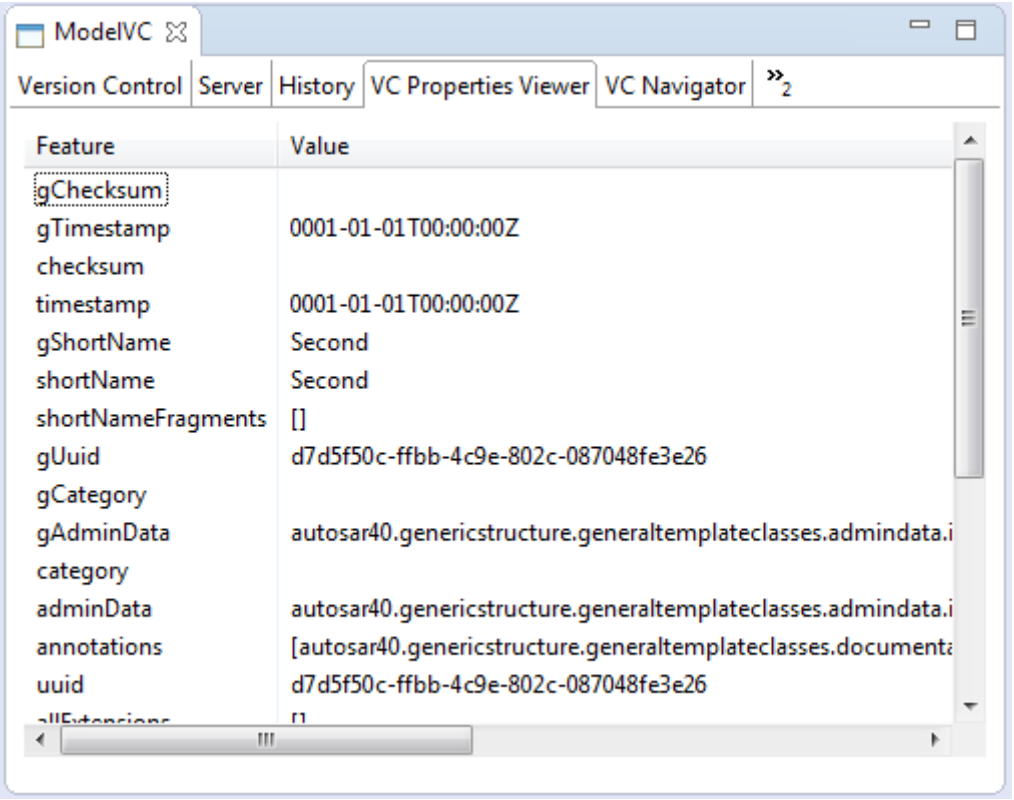


**Figure 13**. VC Properties Viewer Tab.

The VC Properties Viewer tab in figure 13 displays the internal features or contents of an `EObject` or model component selected by the user in the VC Navigator tab. This can be used to aid the user in understanding what information a versioned `EObject` contains in the local EMFStore instance.

### 4.2.3 Helper Dialogs

As EMFStore and the Eclipse Workspace were setup to use the same resource set in the second option for managing model instances detailed in section 4.3.1, a `ResourceSelectionDialog` was created with help of an `ElementListSelectionDialog` from `org.eclipse.ui.dialogs` to let the user pick what model instance within the resource set to track with EMFStore.

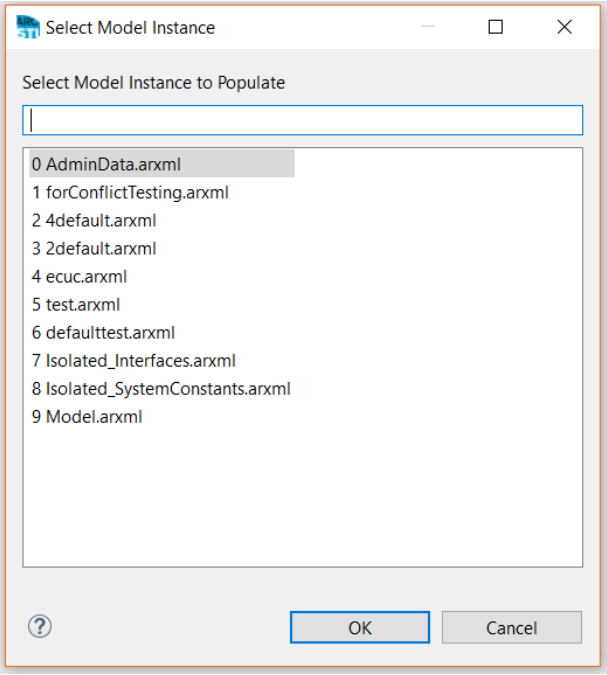


**Figure 14**. Resource Selection Dialog.

Figure 14 shows how the resource selection dialog is presented to the user. Several classes extend this `ResourceSelectionDialog` to display projects from different sources such as from the EMFStore server. Other resources such as `Operations` and `ProjectSpace` resources used by EMFStore were not listed in the dialog to give a cleaner look. These resources still exist within the shared resource set.

***Other Decisions***
Several interviewed developers expressed the desire for a command-line interface for the project result. No command-line interface was built for the project. The reasoning behind not implementing this was to instead focus on getting EMFStore to work as well as possible with AUTOSAR models given the project's time constraints.

## 4.3 Technical Implementation

The following section answers the first research question. Different methods for synchronizing model instances in the Eclipse Workspace and EMFStore are presented. Following this, a method for opening previously versioned model instances in EMFStore is described. To understand how the model version control plugin will be used by end users, a short description of the expected workflow is described.

The final project solution was implemented as an Eclipse plugin added to the Arctic Studio IDE. This plugin was written in Java and is dependent on EMFStore packages.

In the method, it was mentioned that an alternative solution would be to modify AUTOSAR's model representation to better fit the existing model-based version control in EMFStore. It was concluded that this was unnecessary because EMFStore is able to largely handle AUTOSAR objects, with some modifications to the resource type stored in EMFStore.

### 4.3.1 Managing Model Instances

One of the key aspects that had to be solved in the development of the project was how to manage model instances in the solution. In all cases, EMFStore needs a model instance to track changes on. One can choose to design the final solution to handle changes to model instances in either Arctic Studio or internally in a custom client for EMFStore.

Different options exist for managing model instances. In accordance to the method, the different options are listed in order of the least amount of modifications required to implement them. The options include:

1. Maintain a separate environment for modifying model instances where instances are committed to an EMFStore server. All modifications should occur within a separate editing domain from the one used in Arctic Studio, and users can export model instances when they are ready to add them to Arctic Studio and generate code.
2. The second option is to share the same editing domain and resource set between the Eclipse Workspace and EMFStore. Changes performed in the Eclipse Workspace should automatically be tracked in EMFStore and new fetches and updates from a remote EMFStore server should be automatically displayed in the AUTOSAR Navigator used in Arctic Studio as they share the same resources. Changes are performed within the existing AUTOSAR Navigator.
3. The third option is to maintain two separate editing domains, resource sets and resources. Modifications to model instances are performed in Arctic Studio's AUTOSAR navigator. One attaches an `EContentAdapter` adapter to the resource in the Eclipse Workspace and uses the information fetched from a notification message to recreate the command on the resource in EMFStore. This however is not very scalable and requires more code and overhead. Specifically, additional time complexity of iterating through larger models to find target `EObjects` and recreate commands may be less than ideal and could possibly impact the user experience if not performed on a separate thread. The iteration is needed because no IDs exist on the modules in Arctic Studio and thus there is no way to identify the same object on EMFStore side after a population-action.

***First Option***

This was the path taken in the final solution of the project after having explored and worked on the other two options. Two separate editing domains are maintained, one for the Eclipse Workspace and one for EMFStore in the solution plugin. This solution is the least coupled to elements in the Eclipse Workspace compared to the other options.

Users are given three options in terms of selecting what model instance to work with within the model version control plugin: opening a new model instance from an arxml file, opening an existing local project in the EMFStore Workspace or checking out a remote project from the EMFStore server.

When opening a model instance from an arxml file, an `AUTOSARResourceSetImpl` resource set is created, and an entry is given in its `ExtensionToFactoryMap` to use an `AUTOSAR40ResourceFactoryImpl` resource factory when creating resources. A URI for the given arxml file is then used to create a resource from the resource set of type `AUTOSAR40ResourceImpl`.

An AUTOSAR object is the root node in all AUTOSAR models. A typical method of loading a new model instance into EMFStore is to add the root `EObject` of a model to EMFStore's current local project's contents. The children of the root `EObject` are traversed and added as

well to the local project. This does not adequately work when adding an AUTOSAR object, as certain EMFStore features needed in identifying model elements break.

The AUTOSAR root object only has references to its children. These references result in EMFStore interpreting the child `EObjects` or model components as transient. In other words, EMFStore interprets that these child modules are not permanent as they are located elsewhere. This leads to EMFStore not giving components permanent ids. Without these ids, conflict detection and change tracking cannot be sustained on the objects between different instances, such as between different checkouts and remote versions on the server. This results in checksum mismatches on commits to the EMFStore server as well as merge conflicts not being detected.

To fix this issue, an AUTOSAR package (`ARPackage`) is created to act as the container for all direct children of the root AUTOSAR object. This implementation can be seen in Appendix B, section 8. The AUTOSAR object itself is removed from the model instance hierarchy when versioning the model within EMFStore. Later the AUTOSAR object is restored when exporting the model instance from EMFStore to an arxml file for usage in Arctic Studio.

We were unable to identify any information that was lost in removing the AUTOSAR object when versioning the model in EMFStore. This fix resulted in EMFStore correctly setting its own IDs to `EObjects` in the model instance and thus the above-mentioned checksum mismatches and undetected merge conflicts no longer occur. It is worth noting that this fix does not work when operating on a shared editing domain as presented below in the second option.

All changes should be performed within the plugin and any changes performed externally will corrupt version tracking. Users are instead expected to export their model instance to Arctic Studio and then go back to the model version control plugin when they wish to perform modifications. After modifications, the project should be exported again to test it in Arctic Studio.

***Second Option***

This option is superior in terms of usefulness compared to the other two options. By sharing the same `EditingDomain`, `ResourceSet` and `Resources`, changes to model instances can be performed as they usually are within the AUTOSAR Navigator in Arctic Studio and versioning of changes can be tracked automatically within EMFStore.

Since Arctic Studio uses many sphinx components to manage resources and editing domains, sharing the same editing domain between the Eclipse Workspace and EMFStore is not automatic.

To set the editing domain in EMFStore to be the same as the one used in the Eclipse Workspace, one can fetch the editing domain using Sphinx's `WorkspaceEditingDomainUtil` and then set it in EMFStore using `ESWorkspaceProviderImpl` as follows in figure 15.

```
TransactionalEditingDomain editingDomain =
WorkspaceEditingDomainUtil.getAllEditingDomains().iterator().next();
ESWorkspaceProviderImpl.getInstance().setEditingDomain(editingDomain);
```

**Figure 15**. Sharing the Same Editing Domain

The code shown in figure 15 assumes that there is only one editing domain in the Eclipse Workspace. After this, one can create a local project in EMFStore and add a model instance resource's

EObjects to it. Creating a local project before setting the editing domain to be the same as the one from the Eclipse Workspace will result in the local project having the previously set editing domain provided by EMFStore.

To verify that the editing domain used by the local project is in fact the same as the one used in the Eclipse workspace, one can compare the two editing domains' hash codes. To fetch the hash code of a EMFStore local project's editing domain one can use the AdapterFactoryEditingDomain class from EMFStore's API as shown in figure 16 below.

```
AdapterFactoryEditingDomain.getEditingDomainFor(
        _localProject.getAllModelElements().iterator().next()).hashCode();
```

**Figure 16**. Fetching the Hashcode of EMFStore's Editing Domain

However, this alone is not enough to share an editing domain. EMFStore requires that its editing domain contains a command stack that implements EMFStore's ESCommandStack interface. Sphinx's editing domain uses a command stack that does not implement the ESCommandStack interface.

This can be verified by attempting to add EObjects to the newly created EMFStore local project. Figure 17 shows the exception thrown as a result of adding EObjects to a local project housed in an incompatible editing domain.

```
java.lang.ClassCastException
org.eclipse.sphinx.emf.eclipse.emf.workspace.domain.factory.ExtendedWorkspaceEditingDomainF
actory$1 cannot be cast to org.eclipse.emf.emfstore.client.changetracking.ESCommandStack
at
org.eclipse.emf.emfstore.internal.client.model.impl.ProjectSpaceBase.init(ProjectSpaceBase.
java.652)
```

**Figure 17**. Exception Thrown Because of Incompatible Editing Domains.

One place where Sphinx fetches an editing domain is in its WorkspaceEditingDomainManager. The WorkspaceEditingDomainManager then utilizes an ExtendedWorkspaceEditingDomainFactory. A naive solution would be to implement an entirely new version of WorkspaceEditingDomainManager that would instead return a new editing domain with the custom command stack.

A much better solution is to use the extension points that the WorkspaceEditingDomainManager looks at to implement a custom ExtendedWorkspaceEditingDomainFactory equivalent. The purpose of the ExtendedWorkspaceEditingDomainFactory is to construct an editing domain with a custom command stack (that implements EMFStore's ESCommandStack interface requirement) when a new one is needed. The purpose of the DefaultWorkspaceEditingDomainMapping class is to return an editing domain based on what type of meta-model is being used.

The extension point is org.eclipse.sphinx.emf.workspace.editingDomains and one should register both a mapping and factory class. This is illustrated in figure 18.

```
<extension point="org.eclipse.sphinx.emf.workspace.editingDomains">
    <mapping
       class="com.arccore.versioncontrol.model.DefaultWorkspaceEditingDomainMapping"/>
    <factory
       class="com.arccore.versioncontrol.model.VCExtendedWorkspaceEditingDomainFactory">
       <requiredFor
           metaModelDescriptorIdPattern="org.artop.aal.autosar40"/>
    </factory>
</extension>
```

**Figure 18**. Sphinx's editingDomain Extension Point.

We specify the editing domain factory for the AUTOSAR standard used in this project, and that is what the entry `requiredFor` defines. `Org.artop.aal.AUTOSAR40` contains AUTOSAR's own `metaModelDescriptor` and thus one does not need to create a new `metaModelDescriptor`.

To implement this option, we created a class called `VCExtendedWorkspaceEditingDomainFactory` that provides an editing domain that contains a custom command stack that implements EMFStore's `ESCommandStack` interface. The custom command stack is simply an extension of Sphinx's default command stack used in Arctic Studio with the `ESCommandStack` interface implemented.

To force Eclipse to use the new editing domain it is important to know where it is first created. This occurs when the Workspace is being initialized. Sphinx does this by checking the extension point `org.eclipse.sphinx.emf.workspace.editingDomains` shown in Figure 15 when creating an editing domain during workspace initialization.

In order to ensure that EMFStore does not create its own resources when initializing its EMFStore Workspace, one needs to create a custom resource set provider and register it on the `org.eclipse.emf.emfstore.client.resourceSetProvider` extension point. This extension point is checked when EMFStore's `ESWorkspaceProviderImpl` load method is called. By default the resource set provider used by EMFStore is the `ClientXMIResourceSetProvider` class located in the `org.eclipse.emf.emfstore.internal.client.provider` class. However, this provider creates a brand-new `ResourceSet`.

In a custom resource set provider one should override `getResourceSet` to instead return the existing resource set from the Eclipse Workspace instead of creating a new one. An example of this is included in figure 19.

```
@Override
public ResourceSet getResourceSet() {
       ResourceSet set =
WorkspaceEditingDomainUtil.getAllEditingDomains().iterator().next().getResourceSet();
       set.setResourceFactoryRegistry(new ResourceFactoryRegistry());
       set.setURIConverter(new XMIClientURIConverter());
       return set;
}
```

**Figure 19**. Overriding a Custom Resource Set Provider's getResourceSet

It is important to set the `URIConverter` to EMFStore's default `URIConverter`. As named in the theory section of this report, URI converters are used to normalize URIs. Without

EMFStore's `URIConverter`, EMFStore `Resources` will not be understood. If one does not create a new resource set provider like this, then the default resource set provider will create resources that are not associated to a utilized editing domain and will eventually cause a null pointer exception.

The default editing domain provider on the EMFStore side is the `TransactionalEditingDomainProvider` class located in the `org.eclipse.emf.emfstore.internal.client.transaction` package. This editing domain provider creates a new editing domain for EMFStore. To force EMFStore to use the existing shared editing domain, one needs to create an editing domain provider that implements the `ESEditingDomainProvider` interface and register it on the `org.eclipse.emf.emfstore.client.editingDomainProvider` extension point.

To minimize the number of classes used in the project, one can use the same class used to create the editing domain for the Eclipse Workspace with the custom `CommandStack` as named earlier. To make this clearer, an implementation of this `WorkspaceEditingDomainFactory` and `EditingDomainProvider` combination is provided in Appendix B, section 1 with large portions copied from existing source code from Eclipse.

Now changes performed in AUTOSAR Navigator or another `EObject`-level editor will be automatically displayed in the model version control plugin if the contents of the local project are added to a `TreeViewer`.

This combined set of a single editing domain and resource set resulted in checksum mismatches when commits were performed to the EMFStore server. Due to project time constraints when exploring this solution, a checksum error handler was created and registered on the `org.eclipse.emf.emfstore.client.checksumErrorHandler` extension point that simply returned true and pretended that checksum mismatches were handled.

If certain details such as the lack of IDs, incorrect checksums and potentially other problems are solved, then this is the best solution for model version control in Arctic Studio. Due to time restraints, and feedback from ARCCORE, it was deemed out of scope of this project to fix remaining details with this option. The reasoning was that this combination of Sphinx and EMFStore had never been tested before.

Sharing the same `ResourceSet` results in many EMFStore versioning resources, which can clutter the `ResourceSet` quite quickly. Thus, the first option was chosen to provide a working prototype before the end of the bachelor's thesis deadline.

See the Discussion chapter for more details as to why this method was not chosen in the final solution.

***Third Option***

In this option one mirrors changes that occur on model instances found in Arctic Studio's AUTOSAR Navigator, on the copy of the model instance in EMFStore. This can be performed by creating an `EContentAdapter` adapter and attaching it to the resource that the model instance is located in that is going to be mirrored in EMFStore. The resource informs its adapters when changes have occurred to its contents. An example for detecting an `AddCommand` with an `EContentAdapter` can be found in Appendix B, section 2.

To generate a vector of positions that can be used to locate the affected `EObject` model component in Arctic Studio one can use the method found in Appendix B, section 3.

To then take out a reference to the `EObject` from EMFStore that exists in the same position that the change occurred one can use the method found in Appendix B, section 4. These solutions will add some time complexity to the solution depending on the model instance `EObject` tree depth.

This option of performing mirrored commands on a separate editing domain was not selected as there are numerous commands that need to be properly implemented. We were unable to fetch the latest command from the Eclipse editing domain's command stack's `OperationHistory` and thus would have to create commands from scratch. If one could receive the latest command performed, then this would be a possible option for integrating version control of model instances while keeping editing domains separate between the Eclipse Workspace and EMFStore's Workspace.

### 4.3.2 Opening Previous EMFStore Projects

Having the ability to open old EMFStore projects or model instances after one closes Arctic Studio is an important component of functionality that this project supports. If one were to import model instances from scratch every time one started up the model version control plugin, then previous change history will be lost. One could checkout a copy of a model instance from an EMFStore server, yet many times one may have performed changes to a model instance offline and want to continue to work on that model.

Importing and exporting of project histories is supported by EMFStore's internal `ExportImportControllerFactory` class located in the `org.eclipse.emf.emfstore.internal.client.importexport` package [34]. It is important to note that this exports to EMFStore's own save format, and not ARXML. This factory contains two static nested or internal classes, Export and Import. Each of these nested classes contains controllers for the export and import capabilities of EMFStore, respectively.

To export a `ProjectSpace` or project history from EMFStore, one uses the `ExportProjectSpaceController` through the `ExportImportControllerFactory` as shown in figure 20 below.

```
File export = new File("exportedModel");
try {
	new ExportImportControllerExecutor(export,
		new NullProgressMonitor()).execute(ExportImportControllerFactory.Export
			.getExportProjectSpaceController(((ESLocalProjectImpl)
			localProject).toInternalAPI()));
} catch (IOException e) {
	e.printStackTrace();
}
```

**Figure 20**. Exporting Project History From EMFStore

This results in an exportedModel.esp file that can be used to later re-import the project history.

To import a `ProjectSpace` to EMFStore, one can use the `ExportProjectSpaceController` through the `ExportImportControllerFactory`. However it is also possible to use EMFStore's provided `UIImportController`r to give a dialog for users to import Project Spaces stored in .esp files.

```
UIImportController importUI = new UIImportController(new Shell());
importUI.importProjectSpace();
```

**Figure 21**. EMFStore's UIImportController

There also exists an `UIExportController` that can be used as depicted in figure 21. The above implementation is not necessary unless users want to share their offline copies of their older local projects. Instead it is more relevant to select existing local projects that have been persisted in the EMFStore Workspace from earlier sessions in Arctic Studio. These can be trivially accessed as an `Elist` from the EMFStore Workspace.

To use an existing local project (or ProjectSpace) in the EMFStore workspace, we created a dialog that listed the names of projects in the workspace that the user could choose from. Depending on the source chosen: opening an existing local project or a remote local project, different subclasses of the resource selection dialog was used.

### 4.3.3 Exporting EMFStore Projects to AUTOSAR XML

While exporting EMFStore's representation of `ProjectSpaces` with the `ExportImportControllers` are adequate for restoring model instance histories when the model version control plugin is reopened on say a different computer, it is not in an appropriate format for usage within Arctic Studio. Arctic Studio opens models stored in the AUTOSAR XML format, which is a modification of standard XML. EMFStore on the other hand saves its projects to ecp and esp files. In order to export a model instance to an arxml file we use `AUTOSARXMLSaveImpl` by invoking save on an `AUTOSAR40ResourceImpl` resource on which we have filled with a copy of the model instance `EObjects`. AUTOSAR40 denotes the version of AUTOSAR used in the project. See Appendix B, section 5 for our implementation.

It is important to note that if the root EObject of the model instance added to the targetResource is not an `AUTOSARImpl` object, then saving will fail as `AUTOSARXMLSaveImpl`'s `getSchemaLocationMap()` attempts to fetch namespace declarations based on the root `EObject`.

### 4.3.4 Final Solution Hierarchy

The final solution is built upon option 1: utilizing a separate editing domain with its own model instance editor. In this section, we describe how classes were organized within the model version control plugin. A subset of implementation code is in Appendix B.

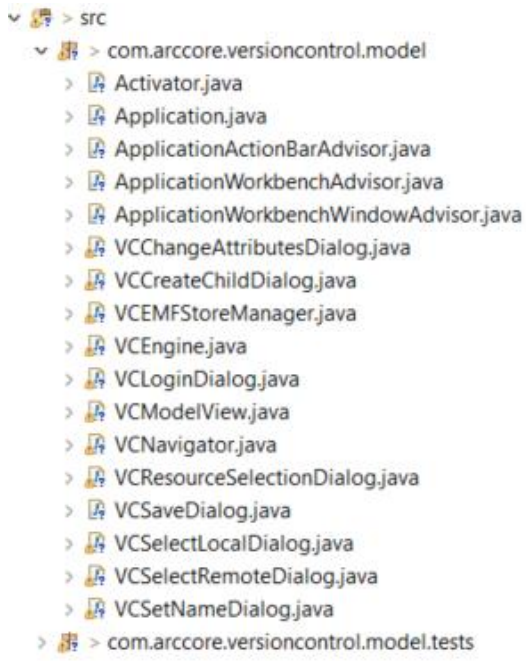


**Figure 22**. The version control plugin contents.

Figure 22 shows all classes used in the final solution and what follows is a description of what every class does. The activator and all application classes used in the project are composed of generated code.

- `VCChangeAttributeDialog`:
  This class takes an `EObject` that it then uses to create a dialog that allows the user to change the `EObject's` attributes. It is utilized in `VCNavigator` to change attributes on a selected `EObject`.
- `VCCreateChildDialog`:
  This class is used with the tree editor to create an `EObject` and place it under the root in `VCNavigator`. Allows the user to create new children to the model instance root in `VCNavigator`.
- `VCEMFStoreManager`:
  Contains all methods used to communicate with the EMFStore server. These include for example: commit to server, login to server, update project with others. Implements server communication, branch support and more.
- `VCEngine`:
  Creates all variables such as local project, remote project, user session, and more. Uses `VCEMFStoreManager` to alter the variables that are connected to EMFStore.
- `VCSelectLocalDialog`:
  Extends `VCResourceSelectionDialog`, but overrides methods to show local project model instances from the local EMFStore workspace and return the selected model instance.

- `VCLoginDialog`:
  Displays a login screen that prompts for a username and password. Used when logging into a remote server.
- `VCModelView`:
  The view used to display the graphical interface of the model version control plugin. Contains all SWT components, initializes `VCEngine`, `VCNavigator` and other classes.
- `VCNavigator`:
  Used to create the tree that is used to display the local project. Contains listeners which enable the user to perform modifications on the model displayed in the tree, such as creating new `EObjects` and changing attributes. The user can delete model components by selecting a component and then pressing the delete key on their keyboard.
- `VCSelectRemoteDialog`:
  Extends `VCResourceSelectionDialog` but overrides methods to show remote projects from the connected server and returns the selected remote project. Used when wanting to checkout a remote project in the plugin.
- `VCResourceSelectionDialog`:
  Abstract superclass for most Dialog classes. Initializes a selectionDialog and provides an interface for methods that all children classes should be using.
- `VCSaveDialog`:
  Is called upon whenever a local project is replaced with a new project if there are unsaved changes. This is so that modifications to a local project will not suddenly disappear without input from the user. Prompts the user to either save or ignore any changes to the local project or model instance.
- `VCSetNameDialog`:
  A dialog that is used when a new local project or other variables require a name to be set. Prompts the user for a name. If none is provided by the user, then the current date and timestamp is used by default instead.

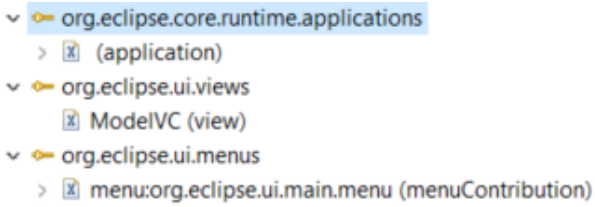


**Figure 23**. The Extension Points Used.

Figure 23 shows what extension points are needed for this class hierarchy. All extension point registrations are generated by Eclipse when one creates a new plugin.

### 4.3.5 Option Two Hierarchy

Option two from section 4.3.1 utilizes a shared domain editor. In this section, we describe in detail what classes and extensions need to be added to the class hierarchy found in the final solution to support shared editing domains.

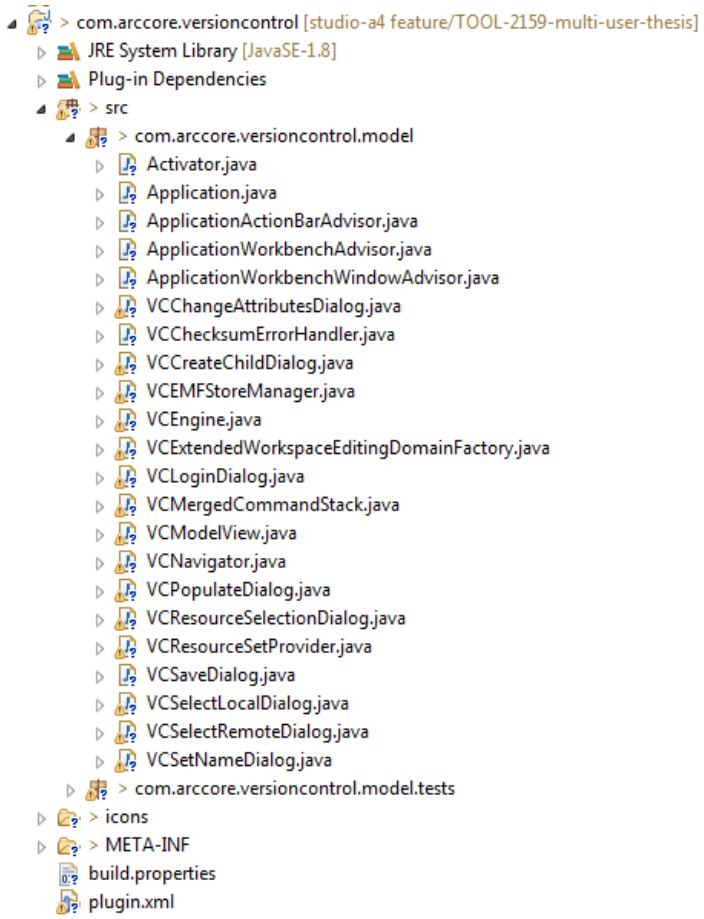


**Figure 24**. Classes required for the second implementation option.

Figure 24 shows the class hierarchy of the second option's implementation. All classes from the final solution are mostly the same except for small changes such as `EMFStoreManager` creates a new project by taking resources from the existing shared editing domain with the help of `VCPopulateDialog` instead of importing it from a file as in the final solution. Other classes that have been added that are needed for shared editing domain support are described below.

- `VCChecksumErrorHandler`:
  Used to ignore checksum errors produced in EMFStore commits. Used for debugging purposes and should technically not be included once all aspects of the shared editing domain option are addressed and completed.
- `VCExtendedWorkspaceEditingDomainFactory`:
  Extends Sphinx's `ExtendedWorkspaceEditingDomainFactory` and differs from the default editing domain factory in that it creates an editing domain with a custom command stack called `VCMergedCommandStack` that implements EMFStore's `ESCommandStack` interface. Otherwise it has the same purpose as the parent class.
- `VCMergedCommandStack`:
  Extends the default `WorkspaceCommandStackImpl` and implements `ESCommand-Stack` to enable EMFStore using the shared editing domain.

- `VCResourceSetProvider`:
  Used to override the default behaviour of the resource set provider used by EMFStore. In the default resource set provider, a new resource set is created. `VCResourceSetProvider` returns the existing resource set used in the shared editing domain instead of creating a new one. This is so that the Eclipse Workspace and EMFStore's Workspace uses the same shared resource set.

Most of the above classes are registered as extensions in EMFStore and Sphinx extension points to enable EMFStore and Arctic Studio to share the same editing domain.

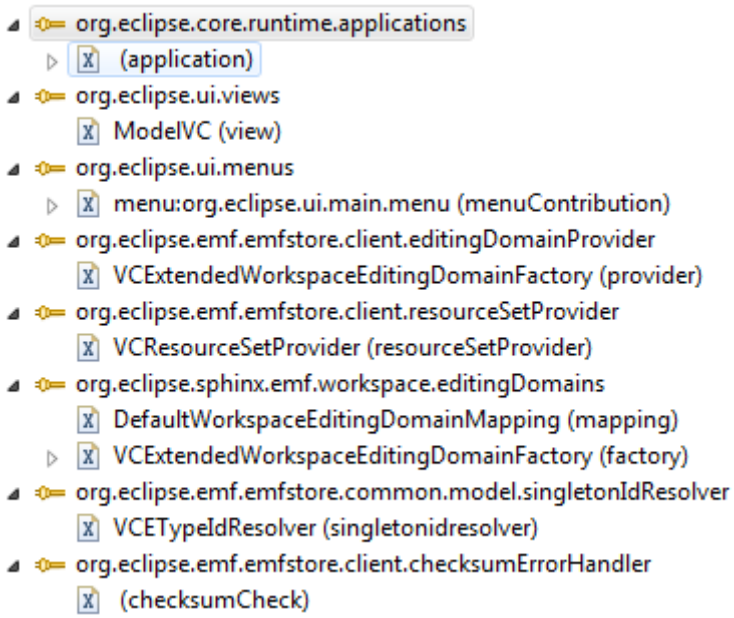


**Figure 25**. Extension point registration list for option two.

Figure 25 shows a complete list of extension points required for the second option detailed in section 4.3.1.

One of the main extension points registered for this option is the `org.eclipse.sphinx.emf.workspace.editingDomains` extension point. The extension we provide uses the `DefaultWorkspaceEditingDomainMapping` (provided by Sphinx) class and `VCExtendedWorkspaceEditingDomainFactory` (created in this project). This extension is used by Sphinx's `WorkspaceEditingDomainManager` and makes the editing domain compatible with EMFStore by replacing the command stack with one that is compatible with both Arctic Studio and EMFStore.

Another main extension point is `org.eclipse.emf.emfstore.client.resourceSetProvider`. The `VCResourceSetProvider` extension is registered on this extension point to make sure Arctic Studio and EMFStore utilize the same resource set.

### 4.3.6 Expected User Workflow

The final solution assumes that users of the model version control plugin conduct all model instance changes within the plugin UI in the Edit Model tab instead of in Arctic Studio's AUTOSAR Navigator. Model elements and attributes can be created and deleted within the plugin UI. This is so that the plugin can manage its own separate editing domain for EMFStore.

With a model instance loaded into the version control plugin, users can commit to a local EMFStore server and test conflict scenarios by checking out several projects within the user interface to view the capabilities of EMFStore when managing AUTOSAR model instances. When an adequate model instance is created, or checked out from the EMFStore server repository, the user can choose to export it serially to an arxml file. This model instance held in the arxml file can then be added to Arctic Studio's editing domain's resource set through the AUTOSAR Navigator plugin, and then Arctic Studio can be used to generate code from the imported model instances.

Whenever the user wants to alter the model instance, the user should go back to the model version control plugin, perform their changes there, commit changes to the EMFStore server, export their model instance to an arxml file and then add the updated model to Arctic Studio. Changes to the model instance should never be performed in the AUTOSAR navigator plugin, as EMFStore does not track these changes.

Adding non-containment references using the attribute modification feature in `VCNavigator` is not supported.

## 4.5 Testing

A list of tested conflict scenarios is listed below. Following this list, results of testing these conflict scenarios on an AUTOSAR model instance are described.

### 4.5.1 Conflict Scenarios

As mentioned in the result of the initial interview, merges that resulted in conflicts in text-based version control systems were difficult to manage and resulted in broken models (see Appendix A, Question 5). The scenarios listed below are meant to simulate conflicts that can occur when several developers are working with an EMF model instance. The purpose of these tests is to show in what aspects EMFStore can better handle conflict scenarios that are not handled in text-based systems, as well as limitations of the EMFStore versioning system.

Conflict scenarios tested:

1. Change conflict: Changes to the same feature or attribute in an `EObject` should result in a conflict detection by EMFStore.
2. Deletion conflict: Two developers are working on the same model instance. One user deletes an `EObject` and commits their changes to the serve. The second user adds an attribute to the same deleted `EObject` and commits their changes. This commit should be detected as incompatible and the user should be prompted appropriately.

To further test the features of EMFStore:

3. Syntactic non-conflict: Changes to whitespace to the serialized version of an EMFStore project should not result in a change being detected in the model instance.
4. Semantic conflict: Tracking of changes to external references of model components in a model instance should be handled by EMFStore.

### 4.5.2 Results of Conflict Scenarios

Two test cases were simulated on a minimal AUTOSAR model instance. Initially the plan was to perform tests on a model instance provided by ARCCORE. This was scrapped, since the sample model utilized external references, which are not easily supported by EMFStore by default. This is documented in the external reference test.

***Change Conflict***

What this test does is first programmatically create an AUTOSAR model instance within EMFStore. This model instance is composed of an AUTOSAR package (`ARPackage`) and an `AdminData` component as a child EObject to the package. This model instance is committed to the EMFStore server, and a second copy of the project is made by performing a checkout on the remote copy of the project. Once two copies of the same remote project exist, the test performs a change to the `AdminData's` checksum attribute concurrently and attempts to commit the conflicting changes to the EMFStore server. EMFStore detects the conflicting commits and provides user interface dialogs for the user to resolve the conflict. An automated test case of this scenario can be found in Appendix B, section 6.

***Deletion Conflict***

The deletion conflict simulation was performed similarly to the previous conflict scenario with the same initial model instance. The difference with this test is that one user deleted the `AdminData` model component while the other user attempted to modify the removed `AdminData's` checksum attribute. EMFStore was able to detect the deletion conflict, and once again provided a user interface for resolving the conflict. An implementation of the scenario can be found in Appendix B, section 7.

***Whitespace in EMFStore ECP/ESP Files***

We investigated if changes in whitespace to the serialized form of model instances stored within EMFStore resulted in the detection of changes to the model instance in EMFStore. This did not occur, as instances are loaded in as `EObjects` and are parsed and tracked on an `EObject` level. Even when modifying whitespace in xml files produced by EMFStore, it did not create a change within the model instance version in EMFStore.

***External References in AUTOSAR Models***

An important component of the eclipse module framework is to support non-containment `EObject` references. Non-containment `EObjects` can exist somewhere else in the same model instance hierarchy or in another model instance `EObject` hierarchy. To test if non-containment references work in EMFStore when operating on an AUTOSAR model instance, two different arxml files were created: ReferenceOutside and ReferenceTest. ReferenceOutside contains only an AUTOSAR package (`ARPackage`) and a ClientServerInterface interface.

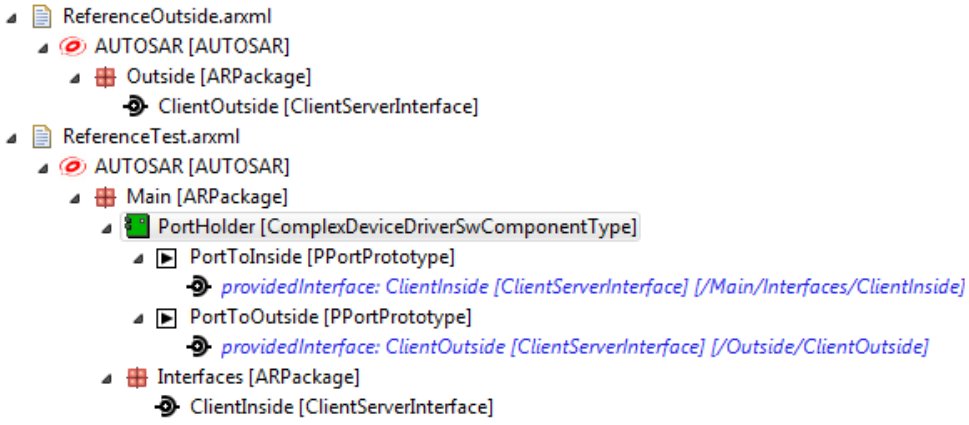


**Figure 26**. Two model instances files tested from AUTOSAR Navigator.

Figure 26 shows two model instance files used in this test as they appear within AUTOSAR Navigator. `ClientServerInterface` was chosen here because `PPortPrototype` can

create a non-containment reference to them. ReferenceTest contains two ARPackages named Main and Interfaces. Interfaces contains a ClientServerInterface named ClientInside, and is used to test if internal non-containment references in the arxml file works when importing to EMFStore.

The ARPackage named Main contains a ComplexDeviceDriverSwComponentType named PortHolder, that allows the creation of a child PPortPrototype. PPortPrototype was chosen because they are able to reference a ClientServerInterface. Two PPortPrototypes were added into the PortHolder: PortToInside, which contains an internal reference to ClientInside, and PortToOutside that contains a reference to ClientOutside in ReferenceOutside.arxml. These references are labelled with a blue colour in AUTOSAR Navigator.

The test is conducted by importing ReferenceTest.arxml into EMFStore. This gave a null exception on PortToOutside reference as EMFStore was not able to resolve the URI of the reference as the reference did not exist within the EMFStore resource set. Then the test was conducted without PortToOutside, which EMFStore imported without any exceptions.

The conclusion of the test is that references to external EObjects will throw a null exception. This because the URI to the external EObject will not be found in EMFStore. EMFStore only has access to the information given in ReferenceTest.arxml and thus cannot find the referenced EObject.

This test was done with separated editing domains and with copying of a model's EObjects into EMFStore using the EcoreUtils helper class [41]. If the test was conducted in the case with shared editing domains this problem will not exist. This because EMFStore and AUTOSAR Navigator shares the same editing domain and thus the same resource set. This will let EMFStore to find the external referenced EObject URI as it is aware of other existing resources and thus will not throw a null-exception.

It is important to note that a text-based version control system such as GIT would not detect external reference conflicts by default. This is not a limitation on AUTOSAR models per-se; if the same editing domain is shared in the project or if the model version control plugin were setup to allow the user to import several model instance files into the same ProjectSpace and resource set, then EMFStore would be able to handle external non-containment references.

## 4.6 Evaluation Interview

To illustrate EMFStore's capabilities in versioning AUTOSAR model instances, an additional tab was added to the plugin to let the user create version control conflicts. Following some of the suggestions by [37], we did our best to ensure that interviewees were comfortable, not stressed and could rely on the interviewer to answer any questions about the final solution. Confidentiality was ensured for all participants, and we asked for permission to record the interviews to minimize the amount of time needed during the interview to transcribe experiences.

We made sure to never argue or defend any design or implementation choice, as it was important to gather information about user experiences and create an open environment where interviewees could speak freely. Following the advice of [37], when interviewees were talking we let them finish what they were saying before switching question or task. Questions were designed to not assume a successful outcome of version control behaviour. This is so that negative as well as positive experiences could be gathered during the interview. After the user

was finished with their tasks and conflict testing, the probe technique of giving a moment of silence followed by asking if the user had any more comments to share was used to make sure all potential experiences were documented. The interview was conducted in ARCCORE's Linköping office on an ARCCORE laptop pre-configured with Arctic Studio and the model version control plugin.

### 4.6.1 Conflict Testing Tab

The reason why this additional tab was included in the model version control plugin was to avoid having to set up an external EMFStore server to demonstrate the capabilities of versioning of AUTOSAR model instances in EMFStore.

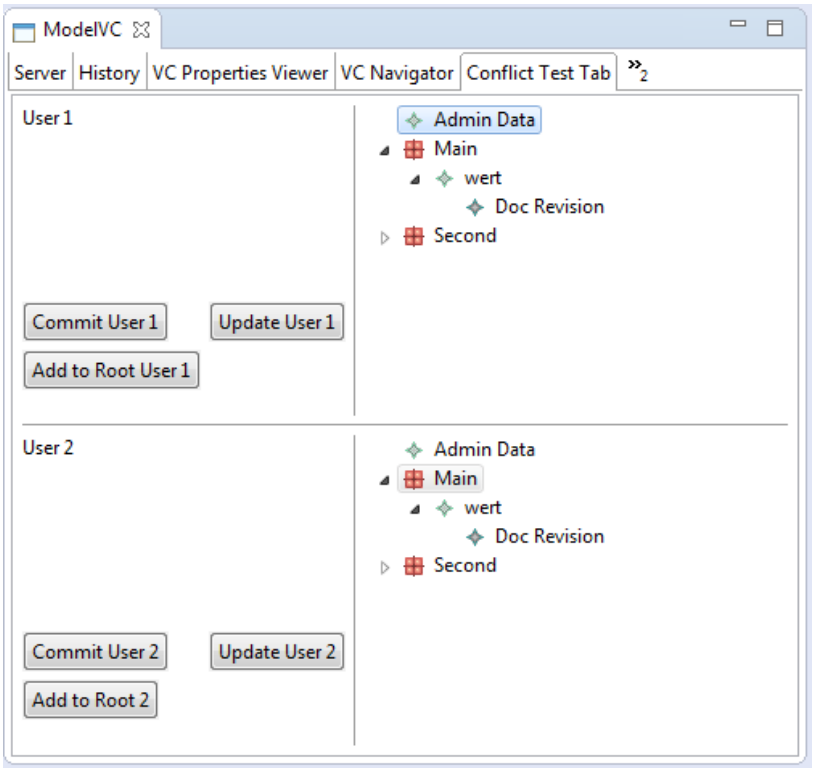


**Figure 27**. Conflict Simulation View and GUI.

Figure 27 depicts that tab constructed for simulating concurrent changes to the same model instance by two different users. This reduced the setup time needed for the evaluation interview. In the additional conflict tab, users can simulate the actions of two users performing concurrent commits to a shared remote project. Users have their own copy of the same checked-out AUTOSAR model instance from the EMFStore server. An additional tab not shown in this report includes buttons to run the automated change and deletion conflicts that were documented in section 4.5.

### 4.6.2 Evaluation Interview Results

Interviews were conducted on developers in ARCCORE's Linköping office. The participants in this interview session were not the same as in the initial interview. The reasoning behind this was solely geographical. The previous interview group was located outside of the Linköping area and the interview was to be conducted in person to record experiences using the solution plugin. All evaluation interviews were conducted in Swedish and then later translated to English when transcribed. This introduces a risk for error, as some words or phrases may have been

incorrectly translated. However, by having one English-native speaker and one Swedish-native speaker in the thesis group, this risk should be small. Interviews took about 20-30 minutes. For more thoroughly transcribed answers, see Appendix C.

In short, for the tested models, interviewees interpreted that EMFStore was capable of handling version control of AUTOSAR models. Since the tested models were small, it is not guaranteed that EMFStore will scale to the largest of models. The user interface was deemed to be lacking, and not something that ARCCORE would use in one of their products. However, it was also noted by interviewees that this was a project to test the capabilities of integrating EMFStore in Arctic Studio and that the user interface was not the focus of the project. Desired improvements from the interviewees is listed below.

***User Interface Modifications***
The terminology used for buttons in the solution plugin should be changed to GIT terms for version control. Examples include changing buttons to be called push and pull, like in GIT. Many dialogs could be potentially merged together. The settings and server tab could also be combined into a single tab as they contain much empty space.

With the current solution for removing the AUTOSAR root object to get AUTOSAR models to work well in EMFStore, the user still expects it to be there, so it creates confusion when the AUTOSAR component is not displayed in the VC Navigator tab. This also introduces confusion as to how the “add to root” button works in the VC Navigator tab. Interviewees expected the navigator to work like AUTOSAR Navigator.

To address the above comments, we modified the button names to reflect GIT-terminology and the server and settings tab were consolidated to a single server tab. The figures in the report were updated to this version.

As for the comments about VC Navigator, it would be a waste of time to implement an entirely new version of the AUTOSAR Navigator, and instead for future work one can integrate the existing AUTOSAR Navigator into the plugin.

***Workflow Modifications***
The best solution would be to continue to use the AUTOSAR Navigator and Properties View that already exist within Arctic Studio. By having two separate editor navigators, it can be confusing for users to perform changes in the right location. This suggests that option two in section 4.3.1, where an editing domain and resource set were shared between Eclipse and EMFStore, is the right direction this project should take if ARCCORE elects to invest more time into it.

# 5. Discussion

## 5.1 Progression of Work

At the start of the project it took a good week or two to fully read through the EMFStore Javadoc to understand any holes that were not covered by the documentation given by EclipseSource and the EMFStore homepage. We quickly realized that some Javadoc pages only gave short explanations as to what different classes did. It can be argued that a large portion of our understanding of EMFStore was gained by taking a clone of the project's GIT repository and digging in the source code. When we hit walls with using Arctic Studio or certain aspects of Eclipse plugin development we made sure to ask developers in the ARCCORE Linköping office for help.

We had made assumptions on certain aspects of version control, such as versioning of metamodels. Specifically, we had thought that we were responsible for both versioning metamodels and model instances. We had read that EMFStore was not designed for versioning of metamodels yet could perform metamodel migration of existing model instances through Edapt.

After talking to our advisor at ARCCORE AB, we learned that an organization called the AUTOSAR Tool Platform User Group (Artop) was responsible for creating metamodels for the AUTOSAR standard and that versioning of the metamodels was Artop's responsibility. Had we not had such great developers to ask questions about the AUTOSAR standard and Eclipse at ARCCORE, then this project would likely have not explored more than one or two options for managing model instances.

Initially we worked on the presumption that one could reload a modified model into EMFStore directly from a file and the changes to that model in the ARXML file would be updated as operations in EMFStore. This was an incorrect assumption, as EMFStore tracks operations performed on `EObject`s in memory and then saves copies of its projects into .ecp and .esp xml save files for persistence. This and other misconceptions were dispelled as we learned more about EMFStore, the Eclipse Workspace, Sphinx, and AUTOSAR.

After having identified that we needed a mechanism to update changes to model instances automatically within EMFStore, we came up with two plans of action: to either detect all commands performed on the model instance resource in the Eclipse Workspace and mirror these changes on the EMFStore copy of the model instance, or to share the same resources between the Eclipse Workspace and the EMFStore Workspace.

We worked on these two options in parallel, but agreed that sharing the same editing domain, resource set and resources would be the best option as changes to a model instance in Arctic Studio's AUTOSAR Navigator editor would automatically be tracked in EMFStore. A shared editing domain solution would eliminate overhead for traversing the two model instances to first select the position of the affected `EObject` in the model instance and then finding its corresponding location in the EMFStore copy of the model instance, then creating a command and executing it on the EMFStore's command stack.

Utilizing a shared editing domain, resource set and resources proved to be non-trivial though. While Arctic Studio is built upon the standard Eclipse Modelling Framework, it uses Sphinx to manage its Workspace and corresponding EMF components. EMFStore also requires that

certain components implement its interfaces. These included implementing an editing domain provider that EMFStore could use that implements the `ESEditingDomainProvider` interface and a command stack that implements the `ESCommandStack` interface.

Other extensions had to be created to make sure that EMFStore used existing EMF components instead of creating its own. A key example is the resource set provider extension that had to be created so that the existing resource set from the Eclipse Workspace would be fetched instead of creating a new resource set for EMFStore.

After having worked on identifying where both Sphinx and EMFStore creates and accesses editing domains, resource sets and resources, other problems arose such as checksums mismatching on EMFStore's side. When a checksum handler was created to ignore checksum errors, problems with id resolution on EMFStore's side arose as well as problems with detecting conflicts in conflicting commits to the EMFStore server.

We also ran into deadlock scenarios where the main Arctic Studio thread was holding onto a lock for the shared editing domain. When EMFStore tried to fork a new thread to perform checkouts of a project, the entire solution would lock up. To temporarily try to work past this deadlock, we created our own checkout method for EMFStore that performed its fetch on the same main thread instead of forking. This seemed to work for the purpose of checking out a project, though other functionality was broken such as the problems with checksums, ids, and conflict detection. It appears that some portion of Sphinx holds onto the editing domain lock.

We worked several weeks in digging in source code and solving one problem only for another to present itself as we became more accustomed to the functionality of EMFStore and Sphinx. There are many aspects of how Sphinx manages EMF components that we still do not fully understand.

At week seven of the bachelor thesis work, a meeting with our advisor at ARCCORE AB was held and it was identified that full integration of Sphinx with EMFStore was out of scope of this project. Thus, all progress on various options was documented and the project focus was switched instead to create a separate environment where we would create a smaller and limited version of the AUTOSAR Navigator within the project plugin so that all model edits and versioning could be performed within the project plugin. The idea had originally been to use the existing editor tools in Arctic Studio to perform model instance edits, as this is the current workflow used in Arctic Studio.

However, by working on this new solution we removed the need to integrate Sphinx with EMFStore, as model changes would occur within a separate editing domain in EMFStore. Final model instances could then later be exported to correct ARXML files that could be imported into Arctic Studio to generate model code. This is the final solution presented in this report and for ARCCORE developers. An additional tab to the plugin was added for the sole purpose of testing conflict scenarios and providing a means to illustrate the capabilities of EMFStore versioning and conflict handling when working on AUTOSAR model instances.

## 5.2 Unsolved Issues

The unsolved problems in this thesis project pertain to the second option of using a shared editing domain detailed in section 4.3.1.

The source of the problem lies in the AUTOSAR root `EObject` used in AUTOSAR model instances. The AUTOSAR `EObject` contains references to all its children `EObjects`. This creates a bug and is due to how EMFStore handles referenced objects. EMFStore categorizes AUTOSAR referenced objects as transient. Transient objects are not provided with permanent ids and neither do their children.

This leads to EMFStore not setting any permanent ids, and instead sets singleton ids to the whole module instance. This causes change tracking and conflict detection to stop working. We debugged this issue by having singleton ids persist by saving them in the EObjectToIdMap set of the EMFStore project instance. With this, the objects received permanent ids but the ids were different between different versions of the project. The remote copy of the project had different ids on its objects compared to the local project. Without having universal ids set to model instance components, versioning fails. It is a bad idea to save all singleton ids to the map and instead some should remain as singletons.

We got around the problem with AUTOSAR root objects and incorrect ids when we had full control over a separate editing domain and resource set in the final solution. This was fixed by removing the AUTOSAR root object when versioning model instances in EMFStore and then later adding it back when exporting the model instance to an ARXML file. With this solution, the AUTOSAR-reference logic does not have to be handled by EMFStore, and the `ARPackage` used as a root wrapper will permit EMFStore to handle all children as permanent objects and give them permanent ids.

We assume that incorrect checksums are due to incorrect IDs being set in the shared editing domain option, though we are not entirely sure. No checksum errors occur when we employed our final solution with removing the AUTOSAR root object. There is a large risk that other aspects of Sphinx and EMFStore functionality was not working in addition to ID and checksum problems, and that we simply were unable to identify them in time before we changed focus of the project to the form it is right now.

## 5.3 Justification of Sources

While this report explored integration possibilities of EMFStore in Arctic Studio as well as the limitations of versioning AUTOSAR models in EMFStore, it was still important to base the exploration on a solid theoretical foundation. To do this, we used published journal articles, workshop articles, and books whenever possible. We tried to avoid secondary sources whenever possible. An example of this is how the book by F. Budinsky et al. [7] titled Eclipse Modeling Framework had Ed Merks, who was the founder of the Eclipse Modelling Framework, as a coauthor. We interpret this book as a primary source.

When published material was lacking, we used links to the homepages of the references' projects or products. Examples of this include the project homepages of Arctic Studio, MinGW, AUTOSAR, Artop and more. This also includes the Javadoc documentation for the EMFStore and EMF project source code as certain portions of those projects have no other documentation other than their Javadoc files.

Two sources that may pose a risk to the legitimacy of the theory portion of this report were the surveys by K. Altmanninger et al. [3] and T. Mens [25]. Using surveys in a report requires that one can judge if the author(s) to that survey came to the correct conclusion of their study. As bachelor's thesis authors, we may not have adequate experience in version control to judge this. We kept the survey by Altmanninger, Sidel and Wimmer, as Altmanninger is the author of

several published articles on model version control and appears to be a credible source on the topic of versioning of models.

# 6. Future Work

There are several features that can be added to the project to further expand its capabilities. In the initial interview, it was mentioned by several participants that a command-line interface for the EMFStore integration would be desirable. This can be implemented at a future date so that users can streamline or automate their version control tasks within their current workflows. To make the current solution useful for users of Arctic Studio, an integration of the AUTOSAR Navigator editor into the solution plugin would be desirable so that users can perform all types of supported operations on a versioned model instance. This should be possible to do if one acquires a full understanding of AUTOSAR Navigator and how it is created, or at least the navigator's tree structure and command creation.

Implementing and testing of EMFStore's Edapt metamodel migration support into the model version control plugin may also be desirable as AUTOSAR metamodels are changed in different versions. EMFStore features metamodel migration support [35]. If new versions of AUTOSAR metamodels are dramatically different from previous versions or if new metamodel versions are seldom created, then introducing metamodel migration support into the plugin implementation may not be of high priority. However, other developers looking to integrate EMFStore into their own extended Eclipse IDE would likely benefit from supporting metamodel migration.

The thesis project implementation only includes support for a local EMFStore server. Versioning was tested on a list of conflict scenarios that were simulated programmatically or by the user through the conflict tab. To increase the value of the project, future work encompasses adding in support for external EMFStore servers.

As mentioned before, the ideal solution for ARCCORE and their Arctic Studio product would be to implement a shared editing domain between the Eclipse Workspace and the EMFStore Workspace. Future work encompasses investigating Arctic Studio's use of Sphinx and fully understanding Sphinx's management of editing domains and other EMF components to either change them in Sphinx or change EMFStore to be able to cooperate with Sphinx.

A big problem for the solution of shared domains is the AUTOSAR reference problem. A solution to this problem may possibly solve most of the problems experienced with the shared domain solution. To solve this problem, an understanding of why the AUTOSAR object makes references to its children is needed and a full understanding of references in the AUTOSAR model is needed.

Future developers need to be well versed in how EMFStore handles references in order to address the issue above. Two possible preliminary solutions include either making EMFStore interpret referenced objects as permanent objects instead of transient ones. This would lead to EMFStore setting permanent ids on referenced objects. A possible problem that this could create is that duplicates of children may be created by EMFStore in the model instance. We performed some testing with forcing the children to receive ids while being transient and it ended with several duplicate copies of the children appearing in the model and that these did not have any ids assigned to them in EMFStore.

In the separate editing domain solution, we tricked EMFStore to interpret AUTOSAR's referenced children to be direct children by removing the AUTOSAR root `EObject` and replacing it with an AUTOSAR package (`ARPackage`) in its place. The wrapping `ARPackage` was later replaced with an AUTOSAR object when serializing the model instance to an

ARXML file. This solution was not possible in the shared domain solution however, as removing the AUTOSAR root object in shared resources would break functionality in Arctic Studio's code generators. The second possible solution would be to change the AUTOSAR module to not having references to its children but adding them as permanent like `ARPackage` does.

# 7. Conclusion

This report concludes to the first research question that EMFStore can be integrated into Arctic Studio and that there are several methods one can choose to achieve this. Each method entailed different workflows with the final Eclipse plugin, with some better than others.

The the plugin solution could handle both change and delete conflicts when versioning an AUTOSAR model instance using the separate editing domain solution (option one). Change and deletion conflict tests failed when tested on the shared editing domain (option two) as ID-handling was broken. We believe it is fair to assume that these conflict tests will pass in option two as detailed in section 4.3.1 if the ID problem is solved.

When testing external references in EMFStore with option one, EMFStore was unable to resolve where external non-containment references were located. This was because referenced `EObjects` in `Resources` had not yet been loaded into the `ResourceSet` used by EMFStore. External non-containment references do work when sharing the same editing domain and resource set in option two in section 4.3.1. This is due to all resources being present in the shared `ResourceSet` right from the start, and thus EMFStore is able to resolve `EObject` URIs. With some form of preloading `Resources` in an intelligent manner, one could solve the external reference issue as EMFStore does support external non-containment references when they exist within the same `EditingDomain`.

To quickly integrate EMFStore versioning of model instances, developers should choose option one as detailed in section 4.3.1, and then expand on the solution according to the future work chapter to support full functionality of model instance modifications in the plugin.

The best solution, if time is not the most pressing factor, would be to employ option two in section 4.3.1 and share the same editing domain, resource set and resources between EMFStore and Eclipse's Workspace. This was also a conclusion made by interviewees in the evaluation interview. By sharing the same data stores, changes can be fully supported and performed in Arctic Studio's AUTOSAR Navigator. Any changes would automatically be tracked in EMFStore, and users can keep their previous workflow and simply use the model version control plugin for versioning of model instances. This could potentially be non-trivial to implement, as Arctic Studio utilizes Sphinx to manage its workspace. This solution would not be free from drawbacks however. By sharing the same resource set, the resource set would quickly become cluttered with EMFStore resources. This would not impact functionality; however, it needs to be taken into account when deciding what method to use.

The evaluation interview pointed out that portions of the user interface were lacking. Several of these were addressed after the interview series, however the responsibility to design a production-ready user interface will be the responsibility of ARCCORE going forward. EMFStore does not feature a difference tool to compare two model instances outside of its version tracking. If this is desired, ARCCORE should investigate other solutions other than EMFStore.

# Bibliography

[1] N. B. Ruparelia, “The history of version control,” ACM SIGSOFT Software Engineering Notes, vol. 35 iss. 1, pp. 5-9, 2010. [Online]. Available: ACM Digital Library, http://dl.acm.org. [Accessed 27 Feb. 2017].

[2] B. de Alwis and J. Sillito, “Why are software projects moving from centralized to decentralized version control systems?”, in Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering, CHASE '09, pp. 36-39, Washington, DC, USA, 2009. IEEE Computer Society.

[3] K. Altmanninger, M. Seidl, M. Wimmer, “A survey on model versioning approaches”, International Journal of Web Information Systems, vol 5, no. 3, pp 271-304, 2009. [Online]. Available: Emerald Insight, http://www.emeraldinsight.com. [Accessed: Feb. 3, 2017]

[4] Vector Davinci Configurator Pro “Configuring AUTOSAR Basic Software with DaVinci Configurator Pro,” Vector Informatik GmbH. [Online]. Available: https://vector.com/vi_davinci_configurator_pro_en.html. [Accessed: May 04, 2017]

[5] Eclipse, “Eclipse Modeling Framework (EMF)”, eclipse.org, [Online] Available: https://www.eclipse.org/modeling/emf/. [Accessed: Feb 5, 2017]

[6] I. García-Magariño, R. Fuentes-Fernández, J. J. Gómez-Sanz, “Guideline for the definition of EMF metamodels using an Entity-Relationship approach,” Information and Software Technology, vol 51, pp. 1217-1230, August 2009.

[7] F. Budinsky, D. Steinber, E. Merks, R. Ellersick, T. J. Grose, Eclipse Modelling Framework. Boston: Pearson Education, Inc, 2004.

[8] Eclipse, “PDE”, eclipse.org [Online]. Available: https://www.eclipse.org/pde/. [Accessed: April 28, 2017]

[9] Eclipse, “Package org.eclipse.emf.edit.domain,” eclipse.org [Online]. Available: http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/edit/domain/package-summary.html. Version 2.11. [Accessed: May 04, 2017]

[10] Eclipse, “Package org.eclipse.emf.common,” eclipse.org [Online]. Available: http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/common/package-summary.html. Version 2.11. [Accessed: May 04, 2017]

[11] Eclipse, “Package org.eclipse.emf.transaction,” eclipse.org [Online]. Available: http://download.eclipse.org/modeling/emf/transaction/javadoc/1.3.0/org/eclipse/emf/transaction/package-summary.html. Version 1.3.0. [Accessed: May 04, 2017]

[12] Oracle, “Solaris ZFS Administration Guide: Transactional Semantics,” Oracle Corporation and/or its affiliates [Online]. Available: https://docs.oracle.com/cd/E19120-01/open.solaris/817-2271/gaypi/index.html. [Accessed: May 02, 2017]

[13] Eclipse, “Package org.eclipse.emf.ecore,” eclipse.org [Online]. Available: http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/ecore/package-summary.html. Version 2.11. [Accessed: April 26, 2017]

[14] Artop, “Architecture,” AUTOSAR Tool Platform User Group [Online]. Available: https://www.artop.org/architecture. [Accessed: April 25, 2017]

[15] AUTOSAR, “Background", AUTOSAR.org, 2017. [Online]. Available: http://www.AUTOSAR.org/about/basics/background/. [Accessed: Feb. 3, 2017].

[16] Eclipse, “Eclipse CDT(C/C++ Development tooling),” eclipse.org, [Online] Available: http://www.eclipse.org/cdt/. [Accessed: Feb 5, 2017]

[17] MinGW.org, “Welcome to MinGW.org,” WinGW.org. [Online]. Available: http://www.mingw.org/. [Accessed Feb. 5, 2017].

[18] jonY, “Msys”, MinGW.org, Jan. 18, 2008. [Online]. Available: http://www.mingw.org/wiki/msys. [Accessed Feb. 5, 2017].

[19] ARCCORE AB, “Arctic Studio", ARCCORE.com, 2017. [Online]. Available: https://www.ARCCORE.com/products/arctic-studio/arctic-studio-for-AUTOSAR-v31/arctic-studio#technical-specification. [Accessed: Feb. 3, 2017].

[20] B. Collins-Sussman, B. W. Fitzpatrick, C. M. Pilato, Version control with subversion, Beijing: O'Reilly, 2004. Chapter 1.

[21] N. Kotwal, and V. Bassi, “A Comprehensive Study of Version Control System in Open Source Software”, International Journal of Scientific & Engineering Research, vol. 3, iss. 4, April 2012.

[22] U. K. Wiil and J. J. Leggett., Concurrency control in collaborative hypertext systems, Conference On Hypertext & Hypermedia, November 14-18, 1993, Seattle, Washington, USA. New York: ACM, 1993.

[23] S. Otte, “Version Control Systems,” Institute of Computer Science, Freie Universität, Berlin, Germany, 2010.

[24] P. Louridas, “Version Control”, IEEE Software, vol. 23, Issue: 1, pp. 104-107, Jan. 2006.

[25] T. Mens, “A state-of-the-art survey on software merging,” IEEE Transactions on Software Engineering, vol. 28, no. 5, pp. 449-462, 2012. [Online]. Available: CiteSeer, http://citeseerx.ist.psu.edu. [Accessed Feb. 7, 2017].

[26] B. Westfechtel., “Merging of EMF models,” Software & Systems Modeling, vol. 13, iss. 2, pp. 757-788, May 2014.

[27] J. Buffenbarger, “Syntactic Software Merging”, in Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, editor J. Estublier, 1995, pp. 153-172.

[28] K. Altmanninger, “Models In Conflict - Towards A Semantically Enhanced Version Control System For Models”, in Proceedings of the MoDELS 2007 Doctoral Symposium, H. Giese, editor, MoDELS Workshops, vol. 5002 of LNCS, pp. 293-304. Springer, 2007.

[29] K. Altmanninger, P. Brosch, G. Kappel, P. Lange, M. Seidl, K. Wieland, M. Wimmer, “Why Model Versioning Research is Needed!? An Experience Report”, in Proceedings of the MoDSE-MCCM 2009 Workshop@ MoDELS. Vol. 9. 2009.

[30] K. Maximilian and U. Technische, 2013, “EMFStore: A Model Repository for EMF Models”, in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 2, ICSE 2010, Cape Town, South Africa [Online]. Available: CiteSeer, http://citeseerx.ist.psu.edu. [Accessed: Feb. 5, 2017].

[31] EMFCompare, “Overview”, eclipse.org [Online]. Available: https://www.eclipse.org/emf/compare/overview.html. [Accessed: February 28, 2017]

[32] Eclipse, “CDO”, eclipse.org [Online]. Available: http://wiki.eclipse.org/CDO. [Accessed: February 28, 2017]

[33] Eclipse, “CDO/Preparing EMF Models,” eclipse.org [Online]. Available: https://wiki.eclipse.org/CDO/Preparing_EMF_Models. [Accessed: February 28, 2017]

[34] Eclipse Che, “Features,” eclipse.org [Online]. Available: https://eclipse.org/che/features. [Accessed: February 28, 2017]

[35] Eclipse, “EMFStore 1.8.0 API,” eclipse.org [Online]. Available: http://download.eclipse.org/emfstore/releases_18/javadoc/index.html. Version 1.8.0. [Accessed: April 20, 2017]

[36] B. Kitchenham, S. Pfleeger, L. Pickard, C. Canada, C. Canada, P. Jones, J. Rosenberg, D. Hoaglin, K. E. Emam, “Preliminary guidelines for empirical research in software engineering,” IEEE Transactions on Software Engineering, vol. 28, no. 8, Aug. 2002.

[37] S. E. Hove and B. Anda, “Experiences from Conducting Semi-Structured Interviews in Empirical Software Engineering Research,” in Proc. 11th IEEE Int. Soft. Metrics Symp., IEEE 2005, 19-22 Sept. 2005, Como, Italy [Online]. Available: http://www.ieee.org. [Accessed: 11 May 2017].

[38] B. Westfechtel., “A formal approach to three-way merging of EMF models,” in Proc. of the 1st Int. Workshop on Model Comparison in Practice, pp. 31-44, Malaga, Spain, 2010. [Online]. Available: ACM Digital Library, http://dl.acm.org. [Accessed 5 March 2017].

[39] Eclipse, “Package org.eclipse.ui.part,” eclipse.org [Online]. Available: https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fui%2Fpart%2FViewPart.html. Version 4.6. [Accessed: May 04, 2017]

[40] D. Springgay, “Creating an Eclipse View,” Object Technology International Inc. [Online]. Available: https://eclipse.org/articles/viewArticle/ViewArticle2.html. [Accessed: May 04, 2017]

[41] Eclipse, “Package org.eclipse.emf.ecore.util,” eclipse.org [Online]. Available: http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/util/package-summary.html. Version 2.7.0. [Accessed: May 10, 2017]

# Appendix

## A. Result of Interview Questions

Note that the interviewees in the initial interview are not the same interviewees as in the evaluation interview.

Question 1: What form of version control do you or your company use to track changes to AUTOSAR models in Arctic Studio?

Answers:

| |
|---|
| Interviewee 1: git |
| Interviewee 2: git |
| Interviewee 3: git |
| Interviewee 4: Git |
| Interviewee 5: Git |
| Interviewee 6: Git |
| Interviewee 7: TortoiseGit (git), TortoiseSvn (svn) |

Question 2: What is your typical method for handling merge conflicts on model projects through Arctic Studio?

Answers:

| |
|---|
| Interviewee 1: We don't in general, we always reexport from scratch as merging is so clumsy |
| Interviewee 2: Open two separate instances of Arctic Studio or two separate projects, each one using one version of the .arxml. Go through the model using a BSW editor or the RTE editor and take a look at all containers. Most of the time though I just git checkout --theirs or --ours since we try to avoid having several devs working on the same .arxml |
| Interviewee 3: Manually in text editor |
| Interviewee 4: Using KDiff3 |
| Interviewee 5: I merge never in Artic Studio, use Tortoise merge or other merge tool, or even use NotePad++ Compare Plugin to see the differences before I merge manually |
| Interviewee 6: Not handled well. The models (arxml) are handled as if they were binary files. |
| Interviewee 7: 1. Copy file prior to pull (roll back merge if it is discovered when pull ingen. 2. Pull, open studio. 3. Copy the copy into workspace. 4. Use emf compare to compare the two files. 5. Handle the merge. |

Question 3: Would you be open to using a specialized version control system in Arctic Studio that differs from your current version control system?

Answers:

| |
|---|
| Interviewee 1: yes |
| Interviewee 2: yes |
| Interviewee 3: Yes |
| Interviewee 4: Very much :) |

Interviewee 5: I would like to try

Interviewee 6: Not really. It would be difficult to use more than one version control system. The most important thing would be the possibility to diff models. Merge models would be nice, but not as important as diff. Having another version control system is not really wanted.

Interviewee 7: I guess. If it is verified safe and legacies already existing VCSs.

Question 4: If you use a textual-based version control system such as Git, SVN, etc, are there scenarios where those systems detect large changes to files and Arctic Studio models when only small details were in fact modified?

Answers:

Interviewee 1: Yes almost always

Interviewee 2: It happens quite often

Interviewee 3: Yes, this is also true for comparing tools such as Beyond Compare.

Interviewee 4: Sometimes if we just modify small things, it is quite easy to detect the changes in the models. But it's still a lot of details.

Interviewee 5: no

Interviewee 6: Yes, always. Due to different order of tags and different UUIDs.

Interviewee 7: Not sure, Maybe. Order could be a problem.

Question 5: Have you experienced any scenarios when two developers made changes to a model and a merge was performed that resulted in a broken model? If you have, please explain the scenario that led to this problem.

Answers:

Interviewee 1: I've never seen it not be broken

Interviewee 2: Yes, because merging .arxmls by hand is always easy to mess up.

Interviewee 3: Yes, can't pin-point one specific case but it's usual when two developers work with the same file.

Interviewee 4: When one developers for example remap all the swc's in the RTE to other tasks and the other one does something else in the RTE to the same SWc's, it will be a lot of merge conflicts and they will try to solve it by using text based version control to merge it. This usually leds to broken model.

Interviewee 5: yes, concurrent changes in the same file, merging manually and made wrong decisions on using their or mine code

Interviewee 6: This never works well... :)

Interviewee 7: Not really.

Question: Have you experienced any scenarios where two developers made changes to a model and a merge was performed that resulted in one developer's edits being entirely ignored?

Answers:

Interviewee 1: see above

Interviewee 2: No

Interviewee 3: Yes

Interviewee 4: Yes

Interviewee 5: No

Interviewee 6: I don't have this kind of detailed info.

Interviewee 7: Nope.

Question: Have you experienced any scenarios where your version control system detects a change even when you haven't modified a model in Arctic Studio?

Answers:

Interviewee 1: No, however comparison tools built in Arctic studio often report this

Interviewee 2: It used to happen a long time ago, but not nowadays.

Interviewee 3: Not sure, probably

Interviewee 4: don't remember

Interviewee 5: no

Interviewee 6: I don't have this kind of detailed info.

Interviewee 7: No.

Question: If ARCCORE were to work on a custom version control system, would you prefer a conflict handling system that lets you keep your current version control system as a back-end or are you open to using an entirely new version control system built-in to Arctic Studio?

Answers:

Interviewee 1: Entirely new, preferably storing information as diffs but diffs of full arxml objects so that committing reverting etc is possible without having to download a "full" big model.

Interviewee 2: Specifically for .arxmls/models, I would use a custom version control system

Interviewee 3: Depends on the solution, but ideally the new control system only.

Interviewee 4: conflict handling maybe

Interviewee 5: yes, if it works

Interviewee 6: Back-end in that case.

Interviewee 7: Door no. 1. Legacying git for example does not force a lot of infrastructure changes. There are so many tools supporting git, I don't see how using a new just for merging issues is worth it.

Question: Do you have any other experiences with version control of AUTOSAR models/projects in Arctic Studio that were less than ideal? If so, can you share some of your experiences?

Answers:

Interviewee 1: Yes, huge problems with git and svn, typically impossible to track a change from the abstract model (System Weaver Based), into Arctic Studio and then to c code, the only solution we have had that works, is to diff the actual c code generated from studio.

Interviewee 2: I am going to write this here since I can't find a better place for it. The Vector DaVinci Configurator tool has a merge tool which is pretty good. Please take a look at that since it is what we need.

Interviewee 3: Not that I can come up with

Interviewee 4: when I have merge conflicts in the models nowadays, I just keep the remote changes and re-do my updates again.

Interviewee 5: no

Interviewee 6: No answer provided.

Interviewee 7: Well, I dont think git have handled a merge conflict of an arxml automatically for me. Ever. So that seems to be really lacking.

Question: Do you or your coworkers prefer using a graphical user interface or a command-line interface when managing version control of projects?

Answers:

Interviewee 1: We have command lines for everything, and since we end up pushing more and more to servers, I think if there is not a command line, it will end up being left behind. Seeing a good diff however is gonna be difficult without a model viewer, so maybe there can be a possibility to view a diff, but the actual commands are piped via command line.

Interviewee 2: I always use command-line interfaces

Interviewee 3: Usually GUI, but trying to be better on CLI :)

Interviewee 4: I use graphical one right now (git extension), but it doesn't matter..

Interviewee 5: It depends on the problem, I would prefer visual, but sometimes visual is too much and then text based is easier to understand

Interviewee 6: Some prefer CLI some GUI. Most likely both variants are needed, but it should be ok if only a CLI version is available.

Interviewee 7: I use both. GUI when checking diff and committing. CLI when starting merge and reverting changes. Can't speak for my co workers. But for your thesis purpose you should imagine both equally important.

Question: If we were to implement a graphical interface for version control in Arctic Studio, would you prefer it to be integrated as a tab or "view" in Arctic Studio, or as a separate window that pops-up over the Arctic Studio interface when a commit or pull is to be performed?

Answers:

Interviewee 1: Studio has large performance problems with big models, I assume this would then depend on how resource heavy the plugin would be. Most git gui's suffer from terrible performance
compared to the cli, and this would be more pronounced in studio. It is also beneficial for people to be able to just use the version control, even if they don't want the full studio license. However, the program has to be launch-able from within studio without you noticing any large difference (graphic or performance wise) as otherwise there will not visibly be an ARCCORE product.

Interviewee 2: A separate tab would be better, I don't like multiple windows associated with the same process. Also, the merge should be possible to do even before a commit/pull.

Interviewee 3: Can't say at this stage actually

| |
|---|
| Interviewee 4: separate window |
| Interviewee 5: Separate Window, can be put on separate screen |
| Interviewee 6: No opinion. |
| Interviewee 7: If I had the complete choice I would prefer it to be a stand alone product not a part of studio and not dependent on Arctic Studio. Furthermore, Artop and hence Studio do not tend to handle corrupt arxml files very well. If you do it in studio anyway it doesn't matter if it is a tag or view. |

# B. Version Control Plugin Snippets

The following is code specific to the implementation of the model version control plugin that is needed to make this project reproducible.

## B.1 Merged EditingDomainFactory and EditingDomainProvider for Integration

```
/*
* Implements methods required for a unified EditingDomainFactory / EditingDomainProvider for Eclipse EMF and
EMFStore.
*/
public class VCExtendedWorkspaceEditingDomainFactory extends ExtendedWorkspaceEditingDomainFactory implements
ESEditingDomainProvider {
    /*
     * Inherits most functions from ExtendedWorkspaceEditingDomainFactory.
     */
   public VCExtendedWorkspaceEditingDomainFactory(){
        super();
   }

   @Override
   public TransactionalEditingDomain createEditingDomain(Collection<IMetaModelDescriptor>
      metaModelDescriptors, ResourceSet resourceSet) {
      return createEditingDomain(metaModelDescriptors, resourceSet, createOperationHistory());
   }

   @Override
   public TransactionalEditingDomain createEditingDomain(Collection<IMetaModelDescriptor>
                        metaModelDescriptors, ResourceSet resourceSet, IOperationHistory history) {

      // Create new WorkspaceCommandStack and TransactionalEditingDomain using given
      // IOperationHistory and ResourceSet
      VCMergedCommandStack stack = new VCMergedCommandStack(history) {

         @SuppressWarnings("restriction")
         @Override
         public void execute(Command command) {
            try {
               execute(command,
               WorkspaceTransactionUtil.getDefaultTransactionOptions());
            } catch (InterruptedException e) {
               handleError(e);
            } catch (RollbackException e) {
               handleError(e);
            }
         }
      };
      TransactionalEditingDomain result = new ExtendedWorkspaceEditingDomain( new ComposedAdapterFactory(
                        ComposedAdapterFactory.Descriptor.Registry.INSTANCE), stack, resourceSet);
      mapResourceSet(result);
      resourceSet.eAdapters().add(new AdapterFactoryEditingDomain.EditingDomainProvider(result));

      ((ExtendedWorkspaceEditingDomain) result).getMetaModelDescriptors().addAll(metaModelDescriptors);
      firePostCreateEditingDomain(metaModelDescriptors, result);
      return result;
   }

   @Override // Return existing shared EditingDomain from Eclipse Workspace
   public TransactionalEditingDomain getEditingDomain(ResourceSet resourceSet){
      return WorkspaceEditingDomainUtil.getEditingDomain(resourceSet);
   }
}
```

## B.2 Mirroring an Add Command between Eclipse and EMFStore

```
EContentAdapter eContentAdapter = new EContentAdapter() {
   @Override
   public void notifyChanged(Notification message) {
      EObject value = (EObject) message.getNewValue();
      Vector<Integer> vec = null;
      int type = message.getEventType();
      locatePositionEObjectInTree((EObject)message.getNotifier(), vec);
      EObject owner = fetchSameEObjectEMFStore(vec);

      // Create a Command that is of the same type from the stored type.
      // sample is for Add, not entirely complete
      if(type == Notification.ADD) {
         EObject copyValue = EcoreUtil.copy(value);
         Command addCommand = AddCommand.create(
                  ESWorkspaceProviderImpl.getInstance().getEditingDomain(), owner, message.getFeature(),copyValue);
         ESWorkspaceProviderImpl.getInstance().getEditingDomain().getCommandStack().execute(addCommand);
      }
    }
}
resource.eAdapters().add(eContentAdapter);
```

## B.3 Finding Position of Given EObject in Model EObject Tree

```
/*
* Helper that returns position of a given EObject in a model instance tree -> through all levels.
*/
protected void locatePositionEObjectInTree(EObject child, Vector<Integer> positions) {
   if(child instanceof GAUTOSAR || child == null) return;
   EObject parent = child.eContainer();
       int i = 0;
       for(Iterator<EObject> it = parent.eContents().iterator(); it.hasNext();) {
           if(it.next().equals(child)){
              positions.add(i);
              break;
           }
           i++;
       }
       locatePositionEObjectInTree(parent, positions);
}
```

## B.4 Fetching EObject from EMFStore in the Location Specified by Given Vector

```
/*
* Input: a vector from locationPositionEObjectInTree or locationPositionEObjectInTreeRec.
* Output: EObject from EMFStore in given location after stepping through EObject hierarchy with positions.
*/
protected EObject fetchSameEObjectEMFStore(Vector<Integer> positions) {
    // assumes that they come in reverse order
    Collections.reverse(positions);
    EObject emfStoreRoot = localProject.getModelElements().get(0);

    // fetch AUTOSAR object
    EObject eob = emfStoreRoot;
    for(int i = 0; i < positions.size(); i++) {
        eob = eob.eContents().get(positions.get(i));
    }
    return eob;
}
```

## B.5 Exporting EMFStore Local Project to ARXML File

```
/*
* Given a destination file in the form of a string,
* Exports ARXML of current model instance stored in local project.
*/
public void exportLocalProjectARXML(String destinationFileName) {
    AUTOSAR AUTOSAR = AUTOSAR40Factory.eINSTANCE.createAUTOSAR();
    // only contents of wrapper ARPackage included, wrapper is tossed
    GARPackage wrapperPkg = (GARPackage)localProject.getModelElements().get(0);

    for (int i = 0; i < wrapperPkg.eContents().size(); i++) {
        EObject eob = wrapperPkg.eContents().get(i);
        if(eob instanceof GARPackage) {
            AUTOSAR.gGetArPackages().add(EcoreUtil.copy((GARPackage)eob));
        } else if (eob instanceof GAdminData) {
            AUTOSAR.gSetAdminData(EcoreUtil.copy((GAdminData)eob));
        } else {
            System.out.println("Unexpected model element on root level"
                    + " when performing serialized ARXML save. ABORTING.");
            return;
        }
    }

    // Same method as in AUTOSAR Navigator
    Resource targetResource = new AUTOSARResourceSetImpl().createResource(
                    URI.createFileURI(destinationFileName + ".arxml"),
                    AUTOSAR40ReleaseDescriptor.ARXML_CONTENT_TYPE_ID);
    targetResource.getContents().add(AUTOSAR);
    try {
        targetResource.save(Collections.emptyMap());
        System.out.println("Successfully saved model instance.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## B.6 Change Conflict Simulation

```
/*
* Change Conflict.
* Init() followed by simulateChangeConflict() is run
* in a test class.
*/

public void init() {
    // Initialize
    ESWorkspace workspace = ESWorkspaceProviderImpl.getInstance().getWorkspace();
    userOneProject = workspace.createLocalProject("userOneProject");
    ESRemoteProject remoteProject = null;
    ARPackage arPackage = null;

    try {
        remoteProject = userOneProject.shareProject(userSession,
                    new ESSystemOutProgressMonitor());
    } catch (ESException e) {
        e.printStackTrace();
    }

    // Populate Model
    arPackage = Autosar40Factory.eINSTANCE.createARPackage();
    userOneProject.getModelElements().add(arPackage);

    userOneAdminData = Autosar40Factory.eINSTANCE.createAdminData();
```

```
    arPackage.gSetAdminData(userOneAdminData);

    // Commit userOneProject
    try {
        userOneProject.commit(new ESSystemOutProgressMonitor());
    } catch (ESException e) {
        e.printStackTrace();
    }

    // Checkout copy, user 1 and 2 have same project
    try {
        userTwoProject = remoteProject.checkout("userTwoProject",
                new ESSystemOutProgressMonitor());
        userTwoAdminData =
                ((ARPackage)userTwoProject.getModelElements().get(0)).getAdminData();
    } catch (ESException e) {
        e.printStackTrace();
    }
}

/*
 * Simulates two users that concurrently change same attribute in a model.
 * Creates all model elements needed for the test.
 * Deletes the local projects after completion of the simulated conflict.
 *
 * Model composed of an ARPackage and an AdminData.
 */
public void simulateChangeConflict() {
    // Change checksum on user 1 copy
    userOneAdminData.setChecksum("user one checksum set.");

    CommitProjectHandler commitHandler = new CommitProjectHandler(userOneProject);
    commitHandler.handle();

    // Change checksum on user 2 copy, create conflict
    userTwoAdminData.setChecksum("user two checksum set.");
    commitHandler.dispose();
    commitHandler = new CommitProjectHandler(userTwoProject);
    commitHandler.handle();

    clean();
}
```

## B.7 Deletion Conflict Simulation

```
/*
 * Simulates two users, one deletes a model element and commits, the other
 * attempts to add an attribute to the deleted model and commits.
 * Creates all model elements needed for the test.
 * Deletes the local projects after completion of the simulated conflict.
 *
 * Model composed of an ARPackage and an AdminData
 */
public void simulateDeletionConflict() {
    // Delete AdminData on user 1 copy
    CommandStack commandStack = ESWorkspaceProviderImpl.getInstance().getEditingDomain().getCommandStack();
    Command deleteCommand = DeleteCommand.create(ESWorkspaceProviderImpl.getInstance().getEditingDomain(),
                userOneAdminData);
    commandStack.execute(deleteCommand);
    CommitProjectHandler commitHandler = new CommitProjectHandler(userOneProject);
    commitHandler.handle();

    // Change checksum on user 2 copy, create conflict
    userTwoAdminData.setChecksum("user two checksum set.");
```

```
    commitHandler.dispose();
    commitHandler = new CommitProjectHandler(userTwoProject);
    commitHandler.handle();

    clean();
}
```

## B.8 Populating an EMFStore Local Project from a File

```
public void populateLocalProject(String filePath) throws Exception {
    EditingDomain storeEditingDomain = ESWorkspaceProviderImpl.getInstance().getEditingDomain();

    Resource resource = null;
    String fileName = new File(filePath).getAbsolutePath();

    URI uri = URI.createFileURI(fileName);
    AUTOSARResourceSetImpl res = new AUTOSARResourceSetImpl();

    res.getResourceFactoryRegistry().getExtensionToFactoryMap().put("*", new AUTOSAR40ResourceFactoryImpl());

    resource = res.getResource(uri, true);
    localProjectInit();

    GAUTOSAR AUTOSAR = (GAUTOSAR)resource.getContents().get(0);
    GARPackage wrapperPkg = AUTOSAR40Factory.eINSTANCE.createARPackage();

    if (AUTOSAR != null && AUTOSAR.eContents().size() >= 1 && AUTOSAR.eContents().get(0) instanceof EObject) {
        if(localProject.getModelElements().isEmpty()) {
            // Only add contents of AUTOSAR in LocalProject.
            // Wrap contents in an ARPackage.
            // Later when exporting, we create a new AUTOSAR object.
            for(Iterator<EObject> i = AUTOSAR.eContents().iterator(); i.hasNext();){
                EObject e = EcoreUtil.copy(i.next());

                if(e instanceof GARPackage){
                    wrapperPkg.gGetSubPackages().add((GARPackage) e);
                } else if (e instanceof GAdminData){
                    wrapperPkg.gSetAdminData((GAdminData) e);
                }
            }
            wrapperPkg.gSetShortName("AUTOSAR");
            localProject.getModelElements().add(wrapperPkg);
        } else {
            System.out.println("Local project already contains EObjects.");
        }
    }
}
```

## B.9 Committing a Local Project to EMFStore Server

```
public void commitLocalToEMFStore(ESLocalProject localProject, ESUsersession userSession) throws Exception {
   if (localProject == null) {
       throw new Exception("Local project not initialized");
   }
   if (userSession == null) {
      throw new Exception("Not connected to a server!");
   }

   try {
      localProject.commit("" ,new ESCommitCallback() {

         @Override
         public boolean baseVersionOutOfDate(ESLocalProject project,  IProgressMonitor monitor){
            if(MessageDialog.openConfirm(new Shell(), "Update", "Update is needed. Do you want to update now?") ){
               update(project);
               return true;
            }
            return false;   // returns false if one cancels update of project.
         }

         @Override
         public boolean inspectChanges(ESLocalProject project, ESChangePackage changepackage,
                  ESModelElementIdToEObjectMapping idmapping){
            if(changepackage.isEmpty()){
               System.out.println("No operations were found, commit aborted!");
               return false;
            }

            ProjectSpace projectspace = ((ESLocalProjectImpl)project).toInternalAPI();
            CommitDialog d = new CommitDialog(new Shell(),
                  ESAbstractChangePackageImpl.class.cast(changepackage).toInternalAPI(),projectspace,
                  ((ESModelElementIdToEObjectMappingImpl)idmapping).toInternalAPI());
            d.open();

            if(d.getReturnCode() == 0){ // If OK was pressed
               String CommitText = d.getLogText();

               // Add message to projectspace
               EList<String> oldMessages = projectspace.getOldLogMessages();

               // If the message list is empty or if not in list, add it
               if(oldMessages.size() == 0 || !oldMessages.contains(CommitText)){
                  oldMessages.add(CommitText);

                  // If message in list, move up the message so it becomes the latest message.
               } else if(oldMessages.contains(CommitText)){
                  oldMessages.move(oldMessages.size()-1, oldMessages.indexOf(CommitText));
               }

               LogMessage LogMessage = LogMessageFactory.INSTANCE.createLogMessage(d.getLogText(),
                     projectspace.getUsersession().getUsername());

               // Adds the message to the changepackage
               changepackage.setLogMessage(LogMessage.toAPI());
               return true;
            } else { // Cancel was pressed. Abort commit.
               return false;
            }
         }
```

```
            @Override
            public void noLocalChanges(ESLocalProject project){
                System.out.println("No local changes, no need for commit.");
                MessageDialog.openError(new Shell(), "Error", “Commit is not needed.");
            }
        }, new ESSystemOutProgressMonitor());
    } catch (ESProjectNotSharedException e) {
        System.out.println("Need to share project before commit!");
        MessageDialog.openError(new Shell(), "Error", "Error project needs to be shared first.");
        e.printStackTrace();
    } catch (ESUpdateRequiredException e) {
        // Do nothing. Update was ignored by user.
    } catch (ESException es) {
        System.out.println("Exception in commit.");
        es.printStackTrace();
    }
}
```

## C. Evaluation Interview Result

Questions and answers were translated to English from Swedish. Observations are also recorded below.

Initial Learning Tasks Notes:

Interviewee 1: It took a few seconds for the interviewee to find the export to arxml button for task 6,
however the interviewee had no problem solving all the given initial tasks.

Interviewee 2: For the first task the interviewee selected the existing project button instead of the populate button, this showed that one can interpret button names in different ways. Adding a package to the root level was not obvious and required that one explained the idea behind the add to root button in the VC Navigator tab. A bug was also found in the history tree, that after one checked out a remote project, the history tree continued to use the old local project for its
history. This bug was fixed for the rest of the interviews.

Interviewee 3: It was not obvious how adding an ARPackage to root was conducted in the VC Navigator tab. The interviewer had to explain how this is performed. It was not obvious where one could find a button or function to export the local project to arxml.

Interviewee 4: The interviewee did not initially understand that the populate button loads a model from a file.

Interviewee 5: The interviewee clicked on the existing project button when instructed to open a model from a file.

**Behavior Questions**

Question 1: Can you explain what information EMFStore presented to you when a conflict occurred?
Answers:

Interviewee 1: One received information that a merge conflict occurred, as well as information about what different changes had occurred on both the local project and the one located on the server. One could open the details button to see and identify what changes occurred.

Interviewee 2: With EMFStore, information about what the variable was called, that the checksum was changed and what value that conflicted was presented. One could even bring up a view where one could see the treeview and the conflict.

Interviewee 3: Changes that had been performed by me and changes that had been performed on the server version were shown, and then I got to choose what to keep.

Interviewee 4: That depends on what level you the question is referring to. I was shown information that we have conflict and I could see what it was that conflicted. I also had the ability to choose what to do about the conflict.

Interviewee 5: What the server had, what objects were pushed to it, and what attributes were set.

Question 2: Can you describe what you did to resolve a conflict?
Answers:

Interviewee 1: One chose what change one wanted to have, and whether to save or commit the local changes to the server. So I assume that one selects the change one wants to keep and then that is
committed to the server [via the dialog].

Interviewee 2: I chose either to “Keep all my changes” or “Keep all their changes” [via buttons].

Interviewee 3: See previous answer.

Interviewee 4: It felt that I could choose either one of the solutions and say that this should be accepted or, no, I will not send in my changes to the server.

Interviewee 5: Chose from the two options given, in this case.

**Opinion Questions**

Question 3: What was the most challenging task to perform? Why was this the most challenging one?
Answers:

Interviewee 1: There is nothing that is exactly super difficult, but one needs to keep track in one’s head that there exists a local project and then one on the server. This is a little new, even if it isn’t so different compared to GIT.

Interviewee 2: No, there was nothing that was specifically difficult, however the GUI could be improved. That the GUI isn’t perfect always happens when doing these kinds of projects, and isn’t so important. It was difficult to get an overview when working, and certain names [of buttons] could be changed to be more explanatory of what they actually do.

Interviewee 3: Not difficult, however the automated test buttons contained a bit of magic and I did not
exactly know what happened.

Interviewee 4: The most difficult part was to understand the connection between version handling and the actual project. That one couldn’t right click [in the AUTOSAR Navigator] and like the export button, have it show up [in the solution plugin]. It did not feel at all intuitive, and it is just something one would have to know.

Interviewee 5: It would be understanding the user interface. This is a test to test the idea of EMFStore, not to actually make a product.

Question 4: Were any aspects of the user interface unclear?
Answers:

Interviewee 1: When dialogs for naming an instance come up, what happens exactly is not obvious. When project names are set and when they are not is not entirely obvious either.

Interviewee 2: One could use the same terms [as in GIT] for pull and push buttons. You two have used certain terms such as commit, so it would be nice to have a consistent theme throughout the project. Change the button called populate to be called load project or load ARXML. Yes it is true that one also populates, but renaming this button would still be clearer. One had to read and think about what buttons do.

Interviewee 3: No I don’t think so. It is a bit unclear how larger merge conflicts will be shown.

Interviewee 4: Instinctively it feels like the VC Navigator should appear before the properties tab as there is a connection between them. No, otherwise it is mainly to get a clearer understanding of the connection between the project and the version control functions. If it is GIT that is usually used, then it would be good to be able to say, okay, this is a thing that it performs [referring to buttons and their function].

Interviewee 5: Commit to local was not clear, as well as other things.

Question 5: If you could choose, what modifications would you make to the workflow of versioning model instances in the solution plugin?
Answers:

Interviewee 1: It is obvious that the optimal solution would be if one continued to use the AUTOSAR Navigator and then somehow could right click to commit the model file directly from the Navigator to the server. This we have talked about earlier though. So in that workflow, one could in principle have a button where one pulled up a server and then checked out from the server, made a change [in AUTOSAR Navigator], committed and then get the dialog windows that you guys have for merge conflicts to appear.
Additional Question by Interviewer: How could it feel if our navigator was replaced with AUTOSAR navigator?
It would be an improvement at least, however it still ends up being a bit of a duplication-issue as there would be several AUTOSAR Navigators. It could be that a user works here [the AUTOSAR Navigator in Arctic Studio] and then one needs to get those changes into EMFStore [as the Navigators would be working on two different model versions].

Interviewee 2: One could automate things more. Right now that exported files end up somewhere in the eclipse project folder is not intuitive. Instead of populating with a file with a file selection dialog, one could choose to populate a project instead and then also export the project so that it is saved in an appropriate location. This way one would understand where one was working.

Interviewee 3: The best thing would be to be able to skip the manual steps, especially the file explorer step with selecting an existing file on the disk [searching for the arxml file]. There also is unlikely to be a case where one doesn't want to update from the server using the GUI. So that first dialog on a commit is unnecessary.

Interviewee 4: It would be desirable to drag over or right click [in AUTOSAR Navigator] and then have the version control pop up. One is used to working with their GIT plugin where one can
directly work with versioning.

Interviewee 5: The workflow is like in the regular Arctic Studio. It is a given that one needs to integrate [EMFStore with the existing AUTOSAR Navigator] so that we get real-time updates.

Question 6: If you could choose, what modifications would you make to the user interface in the plugin solution?
Answers:

Interviewee 1: I am thinking about the property-viewer in Arctic Studio, getting that to work with the plugin would be an improvement. Also, if you could get the file name to be displayed in the tree view, that would help as well [AUTOSAR Navigator shows the root EObject in a tree with the model's file name].

It would be pretty cool if one could see one's own local project and the remote project tree beside one another so that one could inspect differences before performing a commit. Right now one needs to commit in order to see if someone has made a change.

Interviewee 2: When one creates dialogs, certain things could be combined into one dialog instead of several ones. For example, in the test conflict tab, one has two dialogs for creating project names. One can instead change this to a single dialog.

Interviewee 3: Simply the dialog steps. I would maybe remove the cancel commit update dialog step as mentioned before, but that is my preference. Or as a programmer one can add it as a setting and then the user can choose where to have that step or not.

Interviewee 4: See question 4.

Interviewee 5: See question 4.

Question 7: Do you interpret EMFStore as capable or lacking in versioning of autosar model instances? In what aspects was it lacking?
Answers:

Interviewee 1: It feels good. Everything was quick and one could view differences [in the models]. It would be interesting to see a large amount of merge conflicts, however that is hard to set up. It feels that EMFStore should handle it well.

Interviewee 2: Yes, I do think it is capable. It is much better than GIT in the way that one could always go in and see a diff on the actual model-level. When I went in on an object, I could not really get its attributes up in Arctic Studio's built-in property view. It would be nice to attach the property view to the model in EMFStore so that one can see what attributes are set and not just the shortname. Moving between two tabs for the tree view and the properties tab as it is right now is not ideal.

Interviewee 3: Right now it is with small models, so it is hard to answer that question. It works well how it is right now.

Interviewee 4: There were no problems from what I saw. From what I have seen it appeared to work. I have seen much worse models at another workplace that were entirely impossible to perform a diff on.

Interviewee 5: Yes, it seems to.

Other comments during the interview:

Interviewee 1: Additional Question: How difficult would it be as a user to see an ARPackage that is called AUTOSAR [as the root object] instead of an AUTOSAR icon like that shown in the
AUTOSAR Navigator? [We asked this in response to many having difficulties with the add to root button and understanding the tree structure that we modified to get EMFStore to work well with AUTOSAR models]
I believe that that would most likely cause confusion. One can then interpret that the AUTOSAR object in the ARXML file has been removed when adding it to EMFStore.

Interviewee 2: The settings and server tabs could be combined into a single tab.

Interviewee 3: The interviewee did not push the set value button when changing attributes in a model component.

Interviewee 4: Populate did not feel right [referring to name of button].

Interviewee 5: The GUI is nothing that ARCCORE will use.