D4.4 – Prototype tool for collaboration

Version 1.2
4 May 2016
Final

Public Distribution

BME

Project Partners: ARMINES, Autonomous University of Madrid, BME, IKERLAN, Soft-Maint, SOFTEAM, The Open Group, UNINOVA, University of York

Every effort has been made to ensure that all statements and information contained herein are accurate, however the Project Partners accept no liability for any error or omission in the same.
© 2016 Copyright in this document remains vested in the MONDO Project Partners.
# Project Partner Contact Information

<table>
<thead>
<tr>
<th>ARMINES</th>
<th>Autonomous University of Madrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Massimo Tisi</td>
<td>Juan de Lara</td>
</tr>
<tr>
<td>Rue Alfred Kastler 4</td>
<td>Calle Einstein 3</td>
</tr>
<tr>
<td>44070 Nantes Cedex, France</td>
<td>28049 Madrid, Spain</td>
</tr>
<tr>
<td>Tel: +33 2 51 85 82 09</td>
<td>Tel: +34 91 497 22 77</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:massimo.tisi@mines-nantes.fr">massimo.tisi@mines-nantes.fr</a></td>
<td>E-mail: <a href="mailto:juan.delara@uam.es">juan.delara@uam.es</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BME</th>
<th>IKERLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daniel Varro</td>
<td>Salvador Trujillo</td>
</tr>
<tr>
<td>Magyar Tudosok korutja 2</td>
<td>Paseo J.M. Arizmendiarieta 2</td>
</tr>
<tr>
<td>1117 Budapest, Hungary</td>
<td>20500 Mondragon, Spain</td>
</tr>
<tr>
<td>Tel: +36 146 33598</td>
<td>Tel: +34 943 712 400</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:varro@mit.bme.hu">varro@mit.bme.hu</a></td>
<td>E-mail: <a href="mailto:strujillo@ikerlan.es">strujillo@ikerlan.es</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Soft-Maint</th>
<th>SOFTEAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vincent Hanniet</td>
<td>Alessandra Bagnato</td>
</tr>
<tr>
<td>Rue du Chateau de L’Eraudiere 4</td>
<td>Avenue Victor Hugo 21</td>
</tr>
<tr>
<td>44300 Nantes, France</td>
<td>75016 Paris, France</td>
</tr>
<tr>
<td>Tel: +33 149 931 345</td>
<td>Tel: +33 1 30 12 16 60</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:vhanniet@sodifrance.fr">vhanniet@sodifrance.fr</a></td>
<td>E-mail: <a href="mailto:alessandra.bagnato@softeam.fr">alessandra.bagnato@softeam.fr</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The Open Group</th>
<th>UNINOVA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scott Hansen</td>
<td>Pedro Maló</td>
</tr>
<tr>
<td>Avenue du Parc de Woluwe 56</td>
<td>Campus da FCT/UNL, Monte de Caparica</td>
</tr>
<tr>
<td>1160 Brussels, Belgium</td>
<td>2829-516 Caparica, Portugal</td>
</tr>
<tr>
<td>Tel: +32 2 675 1136</td>
<td>Tel: +351 212 947883</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:s.hansen@opengroup.org">s.hansen@opengroup.org</a></td>
<td>E-mail: <a href="mailto:pmm@uninova.pt">pmm@uninova.pt</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>University of York</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimitris Kolovos</td>
<td></td>
</tr>
<tr>
<td>Deramore Lane</td>
<td></td>
</tr>
<tr>
<td>York YO10 5GH, United Kingdom</td>
<td></td>
</tr>
<tr>
<td>Tel: +44 1904 32516</td>
<td></td>
</tr>
<tr>
<td>E-mail: <a href="mailto:dimitris.kolovos@york.ac.uk">dimitris.kolovos@york.ac.uk</a></td>
<td></td>
</tr>
</tbody>
</table>
## Contents

1. The Model-level MONDO Security Lens .......................... 2
   1.1 Overview ............................................................... 2
   1.2 Running Example: Security Policy for Wind Turbine Modeling .... 4
   1.3 Semantics of the Access Control Policy .......................... 4
      1.3.1 Preliminary: Elementary Model Facts ......................... 4
      1.3.2 Interpretation of Security Rules ............................. 7
      1.3.3 Dependencies Between Security Judgements .................. 8
   1.4 Interpretation of Property-based Locks ........................ 9
   1.5 Lens Transformation Rules ....................................... 10
      1.5.1 Overview ....................................................... 10
      1.5.2 The Five Rule Species ....................................... 11
      1.5.3 The Four Rule Groups ...................................... 12

2. Offline Collaboration Prototype .................................. 13
   2.1 Overview ............................................................. 13
   2.2 The MONDO Offline Collaboration Server ......................... 13
      2.2.1 Offline Collaboration Architecture .......................... 13
      2.2.2 Higher-level Security Lens .................................. 14
      2.2.3 Technical Realization ........................................ 15
      2.2.4 Permanent Unique Identifiers for Model-level Security Lens on Offline Models  16
      2.2.5 Authorization Files .......................................... 17
   2.3 Client-side Support for Offline MONDO Collaboration ........... 18
      2.3.1 Conflict resolution with DSE Merge .......................... 18
      2.3.2 Proposed Enhancement: Client-side Support for Locks and Write Access Rules  20
   2.4 Case Study ......................................................... 20
      2.4.1 Gold and front repositories ................................. 21
      2.4.2 Backward transformation ..................................... 21
      2.4.3 Handling conflicts .......................................... 22
      2.4.4 Access Control .............................................. 23
### 3 Online Collaboration Prototype

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Overview</td>
<td>24</td>
</tr>
<tr>
<td>3.2 Architecture</td>
<td>25</td>
</tr>
<tr>
<td>3.3 Case Study</td>
<td>27</td>
</tr>
<tr>
<td>3.3.1 Security lens in the on-line case</td>
<td>27</td>
</tr>
</tbody>
</table>

### 4 Setup and Management of the Prototype Tool

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Security Lens Daemon Initial Setup</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Version Control System Initial Setup</td>
<td>33</td>
</tr>
<tr>
<td>4.3 Version Control System Management and Maintenance</td>
<td>35</td>
</tr>
<tr>
<td>4.4 Online Collaboration Server</td>
<td>35</td>
</tr>
<tr>
<td>4.5 Collaboration Framework Remote Management</td>
<td>36</td>
</tr>
</tbody>
</table>
## Document Control

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Document outline</td>
<td>20 October 2015</td>
</tr>
<tr>
<td>0.2</td>
<td>Ready for internal review</td>
<td>3 November 2015</td>
</tr>
<tr>
<td>1.0</td>
<td>Finalized</td>
<td>3 December 2015</td>
</tr>
<tr>
<td>1.1</td>
<td>Updates during evaluations</td>
<td>22 April 2016</td>
</tr>
<tr>
<td>1.2</td>
<td>Final review</td>
<td>04 May 2016</td>
</tr>
</tbody>
</table>
Executive Summary

Large-scale system development efforts require the collaboration of multiple engineers, possibly from multiple organizations. For traditional, source code-based software engineering, there already are several off-the-shelf collaborative software development tools capable of supporting multiple engineers in:

- individually developing various parts of the software offline, with occasional synchronizations (using Version Control Systems, VCS) as a form of collaboration, or
- (more recently) simultaneously collaborating on the same piece of software in an on-line coding environment; in both cases
- obeying access control policies that determine who can read or modify the various pieces of source code, and
- reducing conflicts by allowing engineers to lock pieces of the source code to temporarily prevent others from modifying it.

The goal of the MONDO Collaboration Framework is to provide analogous features to Model-driven Engineering (MDE) processes, where models rather than source code are the primary engineering artifacts.

The aim of this document is to report on the Prototype Tool Support of the MONDO Collaboration work package, and in particular, to present the final design decisions, refinements and deviations compared to earlier concepts of the MONDO Collaboration Framework. Additionally, short case studies demonstrate the applicability of the proposed solutions to MONDO usage scenarios.

The next three chapters each focus on a (group of) component(s) of the MONDO Collaboration Framework. First, [Sec. 1] introduces the key algorithm responsible for access control and lock enforcement in the back-end. This security core is then embedded into the user-facing parts of the framework, as described by [Sec. 2] for the offline scenario, and by [Sec. 3] for the on-line scenario. Both of these latter Sections give a conceptual overview of the architecture, then discuss server-side modules and the modeling client, and finally conclude with a case study for demonstration.

The new [Sec. 4] was appended at the end of the document for the final revised version. It details the technical aspects of setting up, configuring and managing the delivered tool prototype.
1 The Model-level MONDO Security Lens

1.1 Overview

As introduced in D4.2, the MONDO Collaboration Framework uses an access control policy where users can be denied knowledge of some facts in the model (through filtering and obfuscation). This means that the complete model (which we will refer to as the \textit{gold} model) differs from the view of the complete model that is exposed to a particular user (the \textit{front} model).

A central concern of the Collaboration Framework is to keep the front models of users aware of changes performed by other users, as long as the security policy allows divulging this information. This means that changes performed by one user on their front model must be propagated (potentially in modified form) to the front models of other users. For conceptual simplicity, this is always performed by propagating information through a representation of the gold model. Thus the \textit{update chain} is as follows: when a user modifies their front model, the gold model is adjusted accordingly; then the front model of every other user is updated to reflect the modified gold model. The goal of providing change propagation in general is thus reduced to the simpler task of propagating changes between a single front and gold model.

In the literature of bidirectional transformations [7], a \textit{lens} (or view-update) is defined as an asymmetric bidirectional transformations relationship where a source knowledge base completely determines a derived (view) knowledge base, while the latter may not contain all information contained in the former, but can still be updated directly. The two operations that have crucial importance in realizing a lens relationship are the following:

- \texttt{GET}, which obtains the derived knowledge base from the source knowledge base that completely determines it, and

- \texttt{PUTBACK}, which updates the source knowledge base, based on the derived view and the previous version of the source (which is required, since the derived view does not contain all information).

This is exactly the kind of relationship we find between a gold model (containing all facts) and a front model (containing a filtered and obfuscated view). The \texttt{GET} operation applies the security rules of the MONDO policy language (see Sec. 4 of D4.2) for filtering and obfuscating the gold model into the front model. The \texttt{PUTBACK} operation takes a front model updated by the user, and transfers the changes of the user back into the gold model (leaving details that were hidden from the user intact). This is a lens relationship between models, in contrast with different kinds of lens relationships that will be introduced in Sec. 2.2.2. Once this model-level MONDO security lens is defined, it can be used to realize the update chain outlined above: when a user modifies their front model, the \texttt{PUTBACK} operation is invoked to adjust the gold model; then the front model of every other user is updated by \texttt{GET} to reflect the modified gold model.

The lens concept is illustrated by Fig. 1. Initially, the \texttt{GET} operation is carried out to obtain the front model for a given user from the gold model. Due to the read access control rules, some objects in the model may be hidden (along with their connections to other objects); additionally, some connections
between otherwise visible objects may be hidden as well; finally, some attribute values of visible objects may be obfuscated, or hidden altogether. If the user subsequently updates their front model, the PUTBACK operation checks whether these modifications were allowed by the write access control rules, and the locks granted to other users. If yes, the changes are propagated back to the gold model, keeping those model elements that were hidden from the user intact (preserved from the previous version of the gold model).

The final part of the preceding paragraph deserves some explanation, as it is not immediately obvious why write access control checks should be performed by the PUTBACK operation. Previously in this overview, the purpose of applying a lens to the gold model and obtaining a filtered and obfuscated front model was to achieve read access control. However, the MONDO Collaboration Framework defines write access control as well, and also property-based locking (see Sec. 4 of D4.3). Both of these features (a) may prevent a user from writing to the model, and (b) are defined based on queries that are to be evaluated on the model. Since only the gold model contains all information, such queries cannot be evaluated (at best, they can be approximated) directly on the front repository of each user, and thus locking and write access control can only be enforced by taking into account the gold model as well. Therefore, write access control and lock enforcement must be intertwined with the lens transformation. In particular, PUTBACK must check write permissions and lock status, and fail if applying the current modification to the gold model is not allowed.
The scope of Sec. 1 is to define such a lens transformation between gold and front models, by specifying how GET and PUTBACK are performed to achieve read and write access control, as well as lock enforcement.

1.2 Running Example: Security Policy for Wind Turbine Modeling

Our example is related to the Wind Turbine case study of the MONDO project, where different artefacts and algorithms for controlling a wind turbine are specified and connected to sensors and actuators.

During the development of a wind turbine model, several collaborators work separately. These collaborators can be grouped into roles based on their permissions:

- **WT Owners** are administrators for the model. They can add/remove/edit all kind of content.
- **WT IO Manager**s work only with SystemInput and SystemOutput. WTSystem is obfuscated from them and have no access to any other element in the model.
- **WT Manager**s can modify WTSystem, create WTCtrl and reference SystemInput and SystemOutput, but cannot modify them.
- **System Manager**s are able to view all kind of content and they can create WTCtrl and reference SystemInput, SystemOutput and SystemParam.

In the MONDO Collaboration framework, these permissions are configured by query-based access control policies. The policies consist of (i) Security Rules, that grant or revoke permissions for given users to performs reads, writes or both on model elements identified by (ii) Security Queries, which are graph patterns that can precisely select the affected parts of model. Each rule in the policy applies to a given set of users, and denies or permits access to those model elements that are identified by the referenced query.

In case of the Wind Turbine example, Listing 1 shows a sample policy file that defines rules for each role participating, and these rules reference the graph patterns described in Listing 2. The meaning of the rules is informally explained by inline comments, while the precise interpretation of security policies in general will be discussed in detail in Sec. 1.3.

1.3 Semantics of the Access Control Policy

1.3.1 Preliminary: Elementary Model Facts

For the purposes of specifying the lens transformation, a model is represented as a set of elementary model facts. In case of EMF [8], the following kinds of model facts are used:

- **Object facts** are pairs formed of a model element (EObject) with its exact type (EClass), for each model element object.
policy WindTurbine first-applicable {
  //WTIOManager is allowed to work with IO
  rule permitIO permit RW to WTIOManager {
    query "test.project.objectIO"
  }
  //WTIOManager is not allowed to work with Ctrls
  rule denyControls deny RW to WTIOManager {
    query "test.project.objectControls"
  }
  //Model attribute is obfuscated from WTIOManager
  rule obfModel obfuscate to WTIOManager {
    query "test.project.attributemodel"
  }
  //Name attribute is obfuscated from WTIOManager
  rule obfName obfuscate to WTIOManager {
    query "test.project.attributename"
  }
  //Version attribute is obfuscated from WTIOManager
  rule obfVersion obfuscate to WTIOManager {
    query "test.project.attributeversion"
  }
  //WTManager, WTSystemManager are not allowed to modify IOs
  rule denyIO deny W to WTManager, WTSystemManager {
    query "test.project.objectIO"
  }
  //WTIOManager, WTManager are not allowed to see and modify Params
  rule denyParam deny RW to WTIOManager, WTManager {
    query "test.project.objectParams"
  }
}

Listing 1: Rule definitions
// Queries all input and output objects in the model
pattern objectIO(io) {
    WTCtrlOutput(io);
} or {
    WTCtrlInput(io);
}

// Queries all controller objects in the model
pattern objectControls(ctrl) {
    WTCtrl(ctrl);
}

// Queries all parameter objects in the model
pattern objectParams(param) {
    WTCtrlParam(param);
}

// Queries all systems and their model attribute
pattern attributemodel(system,modelString) {
    WTSystem.model(system,modelString);
}

// Queries all systems and their name attribute
pattern attributename(system,nameString) {
    WTSystem.name(system,nameString);
}

// Queries all systems and their version attribute
pattern attributeversion(system,versionString) {
    WTSystem.version(system,versionString);
}

Listing 2: Queries for rules
Reference facts are triples formed of a source EObject, a reference type (EReference) and the referenced EObject, for each containment link and cross-link between objects.

Attribute facts are triples formed of a source EObject, an attribute name (EAttribute) and the attribute value, for each (non-default) attribute value assignment.

Resource facts are pairs formed of a Resource (essentially a file containing a model or a fragment of a model) and its path (actually, URI) relative to the root location of the model.

Root facts are pairs formed of a Resource and a root element (EObject) of the resource. There is one such fact for each EObject that is not contained in other EObjects, but is rather a top-level element in its respective resource.

Note that for multi-valued references and attributes, each element in the collection of attribute values / reference endpoints at a source EObject will be represented by a separate attribute resp. reference model fact.

These elementary model facts together represent all knowledge contained in the model, with one important exception. In EMF, it is possible to designate multi-valued references and attributes as ordered collections. The ordering information is not captured by the model facts defined above; therefore the approach described in this Deliverable does not guarantee that the ordering of such lists is preserved between the gold and front model. The reason for this design decision is that there is no unique way to provide PUTBACK for ordered list that have been filtered; finding an acceptable resolution of the problem to support this EMF feature is left as future work.

1.3.2 Interpretation of Security Rules

The Security Policy Language (proposed in Sec. 4 of D4.2) distinguishes three kinds of assets whose access can be controlled:

Object assets are defined by a single EObject element of the (gold) instance model. Rules that affect this asset control the access to the object model fact formed of this object and its class.

Reference assets are defined by a triple formed of a source EObject, a reference type (EReference) and a referenced EObject. Rules that affect this asset control the access to the reference model fact formed of the given source, the given reference type and the given referenced object. Furthermore, if the reference type has an eOpposite, a second reference model fact is controlled, formed of the given referenced object, the eOpposite of the given reference type and the given source.

Attribute assets are defined by a pair formed of a source EObject and an attribute name (EAttribute). Rules that affect this asset control the access to any attribute model fact formed of the given source, the given attribute name and an arbitrary attribute value.

Note that access to resource facts and root facts are not directly controllable by the current version of the rule language.
As illustrated by the examples of Sec. 1.2, security rules are associated with graph queries (in the language of EMF-INCQUERY [11]), and may impose further parameter bindings. For each match of the query pattern that is compatible with all specified parameter bindings, the rule affects the asset that is defined according to the rule header using the values in the match. The rule may either allow or deny the write operation, and either allow, deny, or obfuscate the read operation.

Multiple rules may control access to a given model fact for a given operation (read or write); such conflicts are resolved using the resolution strategy associated with the Security Policy. Conflict resolution strategies are proposed in Sec. 4.1 of D4.2; to clarify, access is permitted to a model element if no rules deny it (except for the one strategy where the opposite is explicitly stated). The result of conflict resolution will be referred to as the *explicit intention* of the security policy. In the sequel, we describe how the *effective judgement* is made, taking into account the relationships between model elements in addition to this explicit intention.

### 1.3.3 Dependencies Between Security Judgements

Security judgements cannot always be made independently, or else the resulting model or workflow may become inconsistent.

The first case of dependency is between the two kinds of operations. Model facts that are omitted / obfuscated in the front model should not remove / overwrite the corresponding hidden details of the gold model when a PUTBACK is performed. This is one of the motivations why a user should only be allowed to write model facts that they can also read as well. Formally, write access is denied by the effective judgement (the model fact is effectively frozen) whenever the read operation is effectively denied or obfuscated, even if the security policy does not express the explicit intention to deny the write operation.

EMF models are structured along a strict containment hierarchy: resources contain root objects, which may contain other objects, etc. in an acyclic forest graph representing containment; only objects within this hierarchy are considered part of a model. Thus the interpretation of containment dependencies (obsolenting earlier designs in D4.2) is that if an object is invisible then the entire containment subtree rooted at that object is considered invisible as well. Formally, the effective judgement allows the read of an object fact if and only if (a) the explicit intention of the security policy allows it, (b) the containing object (if any) is readable according to the effective judgement, and (c) the containment link coming from the containing object (if any) is effectively readable. Conversely, the effective judgement allows the write of an object fact if and only if (a) the explicit intention allows it, and (b) the object is effectively readable. Since the containment tree of objects automatically propagates deletions from parent to children, there is an additional restriction on deleting objects: all their transitively contained objects and their attributes and references must be effectively writable.

Accessing attribute values is meaningless if the host object cannot be accessed. The effective judgement allows the read / obfuscated read / write of an attribute fact if and only if (a) this is the explicit intention of the security policy, and (b) the host object of the attribute fact is effectively readable.

Our design decision is that reading references shall obey a similar rule: both endpoints must be visible (there are other possible alternative semantics, such as the semi-visibility described in D4.2, which concept was abandoned since). Writing references are trickier; while most cross-references...
merely point to target objects without changing them, this is not always true: the target object must also be changed in order to manipulate references with eOpposites, as well as containment links. Therefore, the effective judgement allows the read of a reference fact if and only if (a) the explicit intention allows it, (b) the source object is effectively readable, and (c) the target object is effectively readable; while a reference fact is effectively writable if and only if (a) the explicit intention allows it, (b) the source object is effectively writable, and (c) the target object is effectively writable (for containment links and references with eOpposites) / effectively readable (for other reference types). If the metamodel dictates that a reference has to-one multiplicity, the effective write permission has an additional condition that the existing value of the reference, if any, must also be writable (since it is automatically removed upon setting the new value).

Finally, while there is no separate class of assets for root facts, obviously only readable objects should be visible as resource roots, and only writable objects should be moved to/from a position as a resource root. Root facts are effectively readable / writable if and only if the root object itself is effectively readable / writable.

Resources themselves are currently not controlled by the security policy language (although such extensions may be considered in the future), so all resource facts are considered effectively readable and writable. While this may initially seem like a serious security problem, actually attempting to delete a resource that contains unwritable elements will still fail (see Sec. 1.5). Note that there is an additional restriction in place that all resources of the model must be under a given root path (the root folder of the model) in the file system; users cannot add references to external files contained on the disk of the MONDO Collaboration Server that are unrelated to this model, with the hope to bypass security restrictions on another model.

### 1.4 Interpretation of Property-based Locks

In traditional VCSs, locking ensures the owner of the lock that no one can modify the locked file. Focusing on model files, there are approaches that can express more fine-grained locks that only involve a subset of model elements within a model file. In particular, property-based locking generalizes this traditional lock-based approach to managing concurrent changes to a model. Instead of having each team lock a model fragment for which they get exclusive write access, teams specify a property on model elements that no other team can violate. Such a property captures the assumptions/pre-requisites that the team’s changes rely upon. Thus, other teams can make whatever changes they like as long as they do not invalidate these assumptions. The benefit is greater flexibility in allowing concurrent changes. Note that conventional locking approaches are special cases of this general framework.

The property-based locking facility (Sec. 4 of D4.3) of the MONDO Collaboration Framework allows users (or teams) to place locks on the models that prevent the concurrent modification of a precisely (but implicitly) identified set of model facts by other users. While the security lens is not directly concerned with formulating, granting and managing locks, it is the critical component that enforces locks: PUTBACK must reject a write attempt of a user if it conflicts with active locks granted to other users.
Lock management is discussed in Sec. 2. In the sequel, we assume that the set of locks currently granted to various users or teams are known, and focus on interpreting and enforcing these lock by PUTBACK.

A lock is associated with a query (graph pattern), optional value bindings to query parameters, a change direction (appearance, disappearance or both), and the user(s) the lock was granted to. A given model update violates a given active lock if:

- the user that changed the model is different from the user(s) the lock is granted to, and
- when evaluating the query pattern associated with the lock on the gold model, there is at least one match compatible with any parameter bindings specified by the lock that:
  - was not initially present and appeared due to the change (in case the lock is triggered on appearance or both directions), or
  - was initially present and disappeared due to the change (in case the lock is triggered on disappearance or both directions)

The property-based locks are enforced by the following measures. Before PUTBACK starts modifying the gold model, it sets up an incremental query evaluator (EMF-INCQUERY [6] in particular) to continuously match the lock patterns against the gold model, and then registers for change notifications by this evaluator. Whenever the gold model is subsequently updated to reflect a modification in the front model, the incremental query evaluator will update the computed match sets of the patterns. When PUTBACK is notified by the query evaluator of new matches appearing or previous matches disappearing, it can verify whether this change constitutes a violation of a lock.

1.5 Lens Transformation Rules

1.5.1 Overview

Both GET and PUTBACK are implemented as incremental model transformations using EMF-INCQUERY and VIATRA.

Such transformations are specified using transformation rules. A rule is associated with a graph query precondition, an action (parametrized by a match of the precondition pattern), and a numerical priority value. Transformation execution involves repeatedly executing (firing) rules: finding the matches of rule preconditions of all rules (this set of matches is efficiently and incrementally maintained during the transformation), selecting a match that belongs to the rule with the lowest priority value, and executing the action of the rule on that match; the loop terminates when there are no more precondition matches.

We distinguish four groups of transformation rules: both GET and PUTBACK have one group each for adding model facts to the target model if a corresponding fact is present in the source model (additive rules), and one group each for removing model facts from the target model if no corresponding fact is present in the source model (subtractive rules).
All four groups consist of one rule for each of the five kinds of model facts. The four rules in different groups that manipulate a single kind of model fact are considered a rule species. As summarized by Table 1, this makes twenty transformation rules altogether; due to symmetries, they can be automatically generated from the description of the five rule species and four rule groups. This scheme of generating low-level operational rules from high-level mapping specifications is similar but not identical to Triple Graph Grammars (TGG) and QVT [9].

<table>
<thead>
<tr>
<th>Group</th>
<th>Species</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformation</td>
<td>Sign</td>
<td>Model fact kind</td>
</tr>
<tr>
<td>GET / PUTBACK</td>
<td>additive / subtractive</td>
<td>resource / root / object / reference / attribute</td>
</tr>
</tbody>
</table>

### 1.5.2 The Five Rule Species

Each of the five rule species establishes a mapping between gold and front model facts of a given kind:

**Resource rules** associate corresponding resource facts of the gold and front model that have the same relative URI.

**Object rules** associate corresponding object facts of the gold and front model that have the same class. Additionally, the gold and front EObjects thus associated are explicitly linked by a binary correspondence relation (EObject correspondence); the contents of this correspondence table is continuously maintained by this rule species, so that preconditions of other rules may find the gold object associated with a given front object, or vice versa.

**Root rules** associate corresponding root facts of the gold and front model, on condition that they refer to corresponding Resources and EObjects.

**Reference rules** associate corresponding reference facts of the gold and front model that have the same reference type, on condition that they refer to corresponding source and target EObjects.

**Attribute rules** associate corresponding attribute facts of the gold and front model that have the same attribute name, on condition that they refer to corresponding source EObjects and corresponding attribute values. Attribute value correspondence is defined as equality for effectively readable attribute facts, and the data obfuscation relation for effectively obfuscated attribute facts.

Additive rules have ascending positive priority values in the order of their description above, while subtractive rules have the opposite priority of the additive rules in their species (a negative number, and thus prioritized in the reverse order, before any additive rules).

In each species, the visible gold facts are defined as the subset of gold facts (of the kind associated with the rule species) that are either effectively readable or effectively obfuscated. Furthermore, a mapped fact pair is a pair of a front model fact and a visible gold fact (of the kind associated with the
species) that are linked according to the correspondence conditions of the species given above. Both of these concepts are computed as incrementally evaluated queries.

Note that the current security policy language does not support metamodel obfuscation, even though a separate metamodel obfuscator tool has been developed during the course of the MONDO project. We plan to eventually extend the policy language to cover this use case as well; it would be trivial to modify object, reference and attribute rules in a way to associate gold model facts belonging to one metamodel type with front facts belonging to the obfuscated version of the type.

1.5.3 The Four Rule Groups

Each of the five species described above consists of one rule for each rule group.

- Additive GET rule: the precondition is a visible gold fact that has corresponding front model elements (found using the correspondence rules specific to the rule species), but is not yet part of a mapped pair. The action carried out by the rule is to create the corresponding front fact. In case of the object rule, the EObject correspondence entry is also created.

- Subtractive GET rule: the precondition is a front fact that is not part of a mapped pair. The action is to remove the front fact. In case of the object rule, the EObject correspondence entry is also removed.

- Additive PUTBACK rule: the precondition is a front fact that has corresponding gold model elements (found using the correspondence rules specific to the rule species), but is not yet part of a mapped pair. The action is to create the corresponding gold fact. After creating the gold fact, locks and effective write permissions are checked for the new fact, and the whole transaction is aborted and rolled back if a violation is detected. In case of the object rule, the EObject correspondence entry is also created.

- Subtractive PUTBACK rule: the precondition is a visible gold fact that is not part of a mapped pair. The action part first confirms write permissions (and aborts if necessary), then removes the gold fact, and finally verifies locks (aborting and rolling back if necessary). In case of the object rule, the EObject correspondence entry is also removed.

As seen above, PUTBACK rules consult the effective security judgement for write permissions after performing fact creation, but before performing fact deletion. The reason for this discrepancy is that security queries are expected to be evaluated when the model fact associated with the asset exists in the model. Lock violations are always checked after the modification is attempted, as changes of complex patterns have to be detected (see Sec. 1.4). Note also that effective read permissions are taken care of (a) by considering visible gold facts only, and (b) by incorporating obfuscation in the attribute value correspondence.
2 Offline Collaboration Prototype

2.1 Overview

In the offline scenario, models are stored in a Version Control System (VCS, see e.g. Git [5] or Subversion [1]). The goal of the MONDO Offline Collaboration framework is to enable users to check out models from, and commit models to VCS repositories using any standard off-the-shelf VCS client, with fine-grained model access control rules enforced.

These requirements imply that users may “see” a filtered and obfuscated view of the gold repository, i.e. the original VCS repository holding complete information. This view, which will be called the front repository, exposes a complete version history. Models contained in a given revision of the front repository are the filtered and obfuscated copies of the models found in the corresponding revision of the gold repository; fine-grained read access control rules determine which model elements of which models are visible. New revisions can be added to the view models, but there may be fine-grained write access control rules limiting which model elements of the models in the parent revision are allowed to be changed.

Since this scheme enforces the access control rules even if users access their personal front repositories using standard VCS clients and vanilla modeling tools, the usage of client-side MONDO Collaboration Framework components are not mandatory. This greatly enhances the usability and applicability of the offline scenario. Optional client-side collaboration tools may still be used to improve user experience, e.g. for smart model merging (see DSE Merge in D4.3), user-friendly lock management, or preemptive warning about potential write access violations (left as future work).

2.2 The MONDO Offline Collaboration Server

2.2.1 Offline Collaboration Architecture

Fig. 2 illustrates the application of the collaboration server in the offline scenario. The gold VCS repository initially consists of three subsequent revisions of the front model. Users A and B are depicted; since different access control rules apply to them, different parts of the model has to be hidden from them. Neither user is allowed to directly access the gold repository, but using the GET operation, a front repository is automatically derived for both of them. The front repositories mirror the three revisions in the gold repository, but with the appropriate model elements hidden or obfuscated in each revision. Both users can access their own front repository as a bona fide VCS repository; they can keep using their familiar VCS clients and modeling tools. In particular, users may check out either the newest revision, or any older one in the revision history; furthermore, they can commit new revisions. If User A creates a new revision and attempts to commit it to their front repository, the collaboration server executes PUTBACK. The commit will be rejected (with a helpful error message) if any of the modifications performed by User A were unauthorized, or conflicted with locks. Otherwise, PUTBACK will append a new revision to the gold repository that reflects the changes performed by User A on their front model, while preserving model details hidden from User A unchanged from the parent revision. Finally, new revisions in the gold model are automatically propagated via GET to front models that do not yet have them (in this case, the front model of User B).
2.2.2 Higher-level Security Lens

Common off-the-shelf VCS tools track the revisions of a collection of files (organized into folder). File versions co-existing in a given revision can be checked out into a working directory to manifest them in the file system of a computer.

Given a gold working directory containing models with complete information (among other files) and a front working directory intended for holding filtered and obfuscated information, bidirectional synchronization can be performed between the two. Since the front working directory can be derived from the gold working directory, but not the other way around, this is once again a lens relationship (in accordance with Sec. 1.1). The GET and PUTBACK operations of the working directory-level lens are implemented by (a) calling the respective transformation of the previously introduced model-level security lens (see Sec. 1.5) on each access controlled model found in the working directory, and (b) simply copying miscellaneous files that are not part of models, without modifying their contents.

The working directory-level security lens can be further lifted on to the level of entire repository revision histories. The entire revision graph of the gold repository can be mapped to an isomorphic revision graph of the front repository. The repository-level security lens has GET to propagate new revisions in the gold repository to their filtered and obfuscated counterparts at an identical position in the revision history of the front repository. The repository-level PUTBACK will, given that write
permissions are not violated, transform new revisions in the front model to gold model revisions that represent the same changes relative to their respective parent revisions.

2.2.3 Technical Realization

The prototype MONDO Offline Collaboration Server is realized by extending an off-the-shelf VCS server (Subversion in particular, but Git integration is also planned as future work) by pre- and post-commit hook scripts that enforce access control and maintain the lens relationship.

The design of the MONDO Offline Collaboration Server consists of a central gold repository instance, and a separate front repository instance for each user (in some VCS systems, it may be possible to implement the same setup as different compartments within a single repository instance). Existing access control mechanisms (such as firewalls) are used to ensure that the gold model is accessible to superusers only, and each regular user can only access their own front repository. Given the address of their personal front repository, regular users can apply any compatible VCS client to communicate with their front repository, unaware of MONDO Collaboration mechanisms in the background.

The default behavior of the VCS server is extended by MONDO-specific plugins called hook scripts. They process VCS commits to enforce read and write access control (and property-based locking). The front repositories are equipped with MONDO-specific pre-commit hooks, while a post-commit hook is registered at the gold repository. For performance reasons, the actual model-level lens transformation described in Sec. 1 is carried out by a daemon service, which is constantly loaded in server memory. The hook scripts delegate the actual processing of model files to this daemon, so that the Java class libraries for parsing the models and realizing the transformation do not have to be re-loaded upon each hook execution.

When a user attempts to commit a new revision to their front repository, the attached MONDO pre-commit hook will be automatically invoked by the VCS server. The hook will perform the following steps:

1. The contents of the new revision $r'$ are checked out into a working directory in the file system of the server.
2. The parent revision $r$ of the commit is identified.
3. The front revision $r$ is traced to the corresponding revision $R$ in the gold repository.
4. The revision $R$ is checked out from the gold repository to another working directory.
5. The working directory-level PUTBACK transformation (see Sec. 2.2.2) is invoked to update $R$ to $R'$, in order to reflect the changes performed in the new commit.
6. If the PUTBACK detects any attempts by the user to perform model modifications they are not authorized to (either due to write access control or locks, see Sec. 1), then the commit process to the front repository is aborted, and the user receives an error message identifying the unauthorized modification that was attempted.
7. Otherwise, the commit is deemed successful, and \( R' \) is committed to the gold repository (with metadata such as committer name and commit message copied over from the original front repository commit).

8. Working directories and temporary files are cleaned up when the hook script terminates (regardless of success).

The gold repository may be written to by people with superuser privileges, and server-side processes including the MONDO Online Collaboration Framework (see Section 3) and the pre-commit VCS hook described above. When a new commit is added to the gold repository, the attached MONDO post-commit hook will be automatically invoked by the VCS server. The hook will perform the following steps:

1. The contents of the new gold revision \( R' \) are checked out into a working directory in the file system of the server.

2. The parent revision \( R \) of the commit is identified.

3. For each front repository associated with the gold repository, except for the one whose commit is being propagated right now (if the commit to the gold repository is initiated by the pre-commit hook of a front repository)
   
   (a) The gold revision \( R \) is traced to the corresponding front revision \( r \) in the gold repository.
   
   (b) The revision \( r \) is checked out from the front repository to another working directory.
   
   (c) The working directory-level GET transformation (see Section 2.2.2) is invoked to update \( r \) to \( r' \), in order to reflect the changes performed in the new commit.
   
   (d) The new revision \( r' \) is committed to the gold repository (with metadata such as committer name and commit message copied over from the original front repository commit).

4. Working directories and temporary files are cleaned up when the hook script terminates.

### 2.2.4 Permanent Unique Identifiers for Model-level Security Lens on Offline Models

When the MONDO Model-level Security Lens (see Section 1) is applied on models that can be modified offline, additional challenges emerge.

For example, imagine the following scenario:

1. There is a gold model \( M \) containing objects \( S \) and \( T \).

2. GET is invoked on the gold model \( M \) to obtain a filtered and obfuscated front model \( m \), which contains objects \( s \) and \( t \) (some of their attributes may be missing or obfuscated). During GET, the EObject correspondence (see Section 1.5.2) is established, containing pairs \((S, s)\) and \((T, t)\).

3. The user creates a reference link from \( s \) to \( t \) in their front model and persist the modified front model as \( m' \), containing objects \( s' \) and \( t' \).
4. **PUTBACK** is initiated in order to propagate the model update back to the gold model, resulting in the updated gold model $M'$. Let us examine the last step in more detail. **PUTBACK** involves finding the gold model elements $S'$ and $T'$ that correspond to the source and target objects $s'$, $t'$ of the newly created front reference, so that the gold reference from $S'$ to $T'$ can be created. This requires the mappings $(S', s')$ and $(T', t')$ to be present in the EObject correspondence. Since we are discussing the offline case, the model $m$ was modified offline into $m'$, and thus $m'$ has to be loaded anew when performing **PUTBACK**. Consequently, objects $s'$, $t'$ loaded from disk will not be guaranteed to share the memory address of the original objects $s$ and $t$. Similarly, $S'$ and $T'$ are identical copies of, but not the same as, $S$ and $T$. Note that EMF provides no built-in mechanisms to trace $s'$, $t'$, $S'$, $T'$ back to $s$, $t$, $S$ and $T$, respectively. Therefore the pairs $(S, s)$ and $(T, t)$ in the EObject correspondence created by **GET** cannot be taken into account when looking for the source and target gold elements for the new reference link to be created.

The recommended solution is to declare the following requirement against modeling languages that are allowed to exercise read access control using the MONDO Collaboration Framework in the offline scenario: some kind of permanent unique identifier shall be proscribed for all model elements.\(^1\) Such a permanent identifier is preserved across model revisions and lens mappings, and can therefore be used to pre-populate the EObject correspondence relation at the beginning of the lens transformation run.

Returning to our previous example, note that the permanence of the identifier means that the identifier of $S$ necessarily equals that of $s$ (since the lens **GET** preserves the identifier). Also, $s$ and $s'$ must have the same identifier (since the identifier is immutable across model revisions), just like $S$ and $S'$. In the end, it turns out that $s'$ and $S'$ have matching unique identifiers, and so do $t'$ and $T'$. This allows **PUTBACK** to initialize the EObject correspondence relation with pairs $(S', s')$ and $(T', t')$ and then execute the transformation rules successfully.

### 2.2.5 Authorization Files

The authorization files consumed by the MONDO Security Lens are the following:

**Policy files** describe the MONDO Query-based Access Control Policy. They include:

- `.macl` files defining security policies and rules, and their referenced
- `.eiq` query files defining the queries used by the rule specifications.

**Lock files** describe MONDO Property-based Locking. They include:

- `.mpbl` files defining locks granted to users, and their referenced
- `.eiq` query files defining the queries used by the lock specifications.

\(^1\) Furthermore, access control rules shall not deny read access to this identifier of an object, unless by denying read access to the object altogether.
We have taken the design decision that the *authorization files* are stored and versioned in the same VCS as the models. Thus policy files may evolve naturally along with the evolution of the contents of the repository.

Policy files are writable by superusers only, while users may write their own lock files to request locks. But both kinds of authorization files are readable by every user; this means that offline clients (see Sec. 2.3.2) may evaluate security rules and locks on their offline copies themselves. Note that we do not believe that this openness of the security policy causes major security concerns, as security by obscurity is not good security; in any way, security rule names and parameters should not themselves contain sensitive design information.

### 2.3 Client-side Support for Offline MONDO Collaboration

#### 2.3.1 Conflict resolution with DSE Merge

In offline collaboration, several collaborators work on the same model independently. At the end of their work, they commit the changes back to a VCS repository. It is possible that two collaborators concurrently modify the model in a way which results a conflict. In this case, DSE Merge can be applied described in D4.2 in details. We fully integrated our merge tool into Eclipse IDE through the Eclipse Team API.

Processing of DSE Merge can be split into two phases:

- Calculation phase - when the collaborator sets the priorities (*may/must* marks) of the changes and executes the calculation
- Representation phase - after the calculation, the user has to select one solution from several possible ones.

**Fig. 3** shows the user interface for the *Calculation phase*.

1. Shows the original model
2. Shows the differences between the original and the local models ($\Delta L$)
3. Shows the differences between the original and the remote models ($\Delta R$)
4. Each difference has check box to mark it as *must*
5. The *Calculate Solutions* button is responsible for executing the DSE Merge engine.

**Fig. 4** shows the user interface for the *Representation phase*.

6. The solution tab shows the calculated solutions. Each solution contains the deleted objects (elements under the Cemetery) and unexecuted operations
7. Right click on a solution opens the context menu which has a *Select solution #* entry. Clicking on this entry will apply the solution onto the model
8. The *Revert solution* button is responsible for undoing the previous application of a solution
9. After applying a solution, the editor becomes dirty which means, that it can be saved (Ctrl + S by default)
Figure 3: DSE Merge - Calculating phase

Figure 4: DSE Merge - Solutions phase
2.3.2 Proposed Enhancement: Client-side Support for Locks and Write Access Rules

As discussed in [Sec. 2.2.5], authorization files are readable by every user, and available in any working copy checked out from VCS. The availability of these files allows the user to employ optional client-side MONDO Collaboration Tooling to monitor local modifications to models (Workspace Tracker) and evaluate them against the write access rules and locks granted to other users. No such non-essential tools were actually developed during the course of the MONDO, we merely identify the potential benefits such an extension would bring.

This way, modification attempts that are likely to be rejected by the server can be detected by the client, and the user can be warned in advance. Advanced modeling tools may even incorporate this information into their model notation, e.g. to visually show parts of the model frozen. While none of these capabilities are required for successful collaboration, using such optional client-side support components may greatly improve the user experience of the MONDO Collaboration Framework.

Note that since some users may not have full read access, such client-size evaluation and checking is at best an approximation; server-side enforcement is still mandatory to ensure precise rule application, as well as for security.

Since locks (as opposed to security rules) can be created by a majority of write-capable users, a very useful additional client-side feature would be a UI for managing the users own locks.

2.4 Case Study

As a motivating example, we provide a scenario on which all parts of the offline collaboration framework can be demonstrated. See [Sec. 1.2] for an introduction to this case study and the Security Policy that applies.

Our example works with a simplified example of a wind turbine (WT1) depicted in [Fig. 5]. Real models are obviously larger, but to avoid complexity, sample models of this deliverable are a subset containing only artifacts that are related with the cooling of the Generator Subsystem:

![Figure 5: Original model in Gold repository](image-url)
• **Inputs**: Wind turbine WT1 gets data from a temperature sensor specified by the **SystemInput** identified as **GeneratorTemperature**.

• **Outputs**: WT1 acts on two fans for cooling the wind turbine generator specified by the **SystemOutputs**: **GeneratorFan1Activator** and **GeneratorFan2Activator**.

• **Params**: WT1 is parametrized to specify temperature limits for starting generator cooling by the **SystemParams**: **GeneratorCoolingTempLimit1** and **GeneratorCoolingTempLimit2**.

Subsystem **Generator** contains all the control units for cooling the Generator:

• **CoolingFan1**: this control unit (of type **FanCtrl**) specifies the control algorithm for fan #1. It has **High** priority cycle, it is **enabled** and has 1 order value. Its input references **GeneratorTemperature** (**SystemInput**). Its output references **GeneratorFan1Activator** (**SystemOutput**). And its **param** references **GeneratorCoolingTempLimit1** (**SystemParam**).

• **CoolingFan2**: this control unit (of type **FanCtrl**) specifies the control algorithm for fan #2. It has **High** priority cycle, it is **enabled** and has 2 order value. Its input references **GeneratorTemperature** (**SystemInput**). Its output references **GeneratorFan2Activator** (**SystemOutput**). And its **param** references **GeneratorCoolingTempLimit2** (**SystemParam**).

### 2.4.1 Gold and front repositories

The unfiltered, original models are stored in the *gold* repositories. Only the **WT Owners** have permission to commit their changes here. For the rest of roles, the collaboration framework creates *front* repository. Related to the **WT IO Manager**, **WT Manager** and **System Manager**, three different front repositories are required. Content of these *front* repositories are transformed by GETpart of the bi-directional lens transformation. Thus, the models in these front repositories are filtered related to the permissions. Models stored in each repositories are depicted in [Fig. 6](#). In the view related to **WT IO Manager**, the attributes of **WTSystem** are obfuscated and only **SystemInput** and **SystemOutput** objects appear. Model of **WT Manager** shows all the objects in the model except the **SystemParams** while **System Manager** has the same view as **WT Owners**.

### 2.4.2 Backward transformation

In offline scenario, each user separately modifies the local copy of the model and then commits their changes to their repositories. For instance, one of the **WT IO Managers** removes the **GeneratorTemperature** input from the model as it is depicted in [Fig. 7](#) and commits it to their front repository. This request is captured by the **pre-commit hook** which checks the file to be committed. As a model file is committing, so the lens transformation have to be called by the hook. After the transformation successfully executed, the hook commits the changes onto the *gold* repository. This commit is also captured a **post-commit hook** in the gold repository. This hook will call the transformation again, but this time, it will transform the new state of the model to each *front* repositories and make a commit on them.
2.4.3 Handling conflicts

Before the previously discussed commit of WT IO Manager is accepted, a WT Manager also modified a local version of the original model. He/she created a new CoolingPump that references the GeneratorTemperature input and CoolingFan2 controller is removed. After the WT IO Manager successfully committed the deletion, the WT Manager tries to commit his/her changes depicted in Fig. 8. Unfortunately, the local version of WT Manager is behind from the front repository with a commit (created by the post commit-hook on the gold repository). Thus WT Manager has to execute an update, which leads the local version into a conflict state as several the deletion from WT IO Manager is in conflict with changes of WT Manager. In this case, DSE Merge can be called.

In this very simple case, only a few local (creating a new object, setting its attributes and references and removing the old controller) and one remote changes (deleting input) are available. Our WT
Manager user marks all the changes on local and remote side to be must include. DSE Merge results two solutions (Fig. 9): #1 uses only the local changes while #2 skips the set reference to the input and uses the deletion of it. Our user selects #1 that will be applied on the model.

Figure 9: Changes from an WT Manager

2.4.4 Access Control

At the time of merging process, our model in the gold repository has no GeneratorTemperature input. But after the merge, the WT Manager selected a solution that recreated a deleted object. WT Manager tries to commit the merged model to the front repository. Pre-commit hook calls again the lens in the name of the user who made the commit attempt. Lens transformation recognizes that an SystemInput is created by a user, who has no permission for such an operation. Thus, the pre-commit hook fails and the commit is declined.
3 Online Collaboration Prototype

3.1 Overview

In the online scenario, users do not keep copies of the models on their own machine. The models are hosted instead by the MONDO Collaboration Server. Several users can view and edit the single copy of the model simultaneously, by using a modeling tool (typically an online web application) that accesses the model directly on the server.

Such a collaborative modeling session that keeps a model in server memory for remote access may also be called a whiteboard. Similar to modern collaborative editing tools (such as Google Sheets [4]), whiteboards can be operated transparently: whenever the first user attempts to open a given model, a new whiteboard is started; subsequent users opening the model will join the existing whiteboard. When all users have left, the whiteboard can be disposed. The model may be persisted periodically, or on request (“save button”).

Since the MONDO Collaboration Framework defines read access control capabilities, it follows that the users do not actually see the same model contents - each of them can actually connect to a private copy of the model that has been filtered and obfuscated, but kept in synch with (visible) modifications performed by other users.

![Diagram](image)

**Figure 10: Overview of Online Collaboration**

[Fig. 10] shows how this is realized using security lens (see Sec. 1). The collaboration server hosts a number of whiteboard sessions, each equipped with a gold model. Each user connected to a whiteboard is presented with their own front model, connected to the gold model via a lens relationship. The front models are initially created using GET. If a user modifies their front model, the changes
are propagated to the gold model using \textsc{putback}, and propagated further to the other front models using \textsc{get} again. In case of online collaboration, these lens operations are continuously and efficiently executed as \textit{live transformation}, so users always see an up-to-date view of the model during the editing session.

Apart from actual model access, a \textit{session manager} component enables users to start / join or leave whiteboard sessions, persist models to disk or version control, etc. The users perform all of these operations via a client application that directly connects to the server and the whiteboard in particular; this client is typically a modeling tool implemented as a web application (which can also be hosted on the same collaboration server) accessible by any modern web browser.

### 3.2 Architecture

An overview of the proposed internal architecture for the online collaboration server and web-based modeling tool is depicted in Fig. 11.

![Figure 11: Online Collaboration Proposed Architecture](image)

The core MONDO Collaboration Server contains a session manager that oversees a collection of online collaboration sessions (whiteboards). The security lens (see Sec. 1) is invoked to maintain a live relationship between front and gold models within a whiteboard, enforcing query-based read access control, write access control and property-based locks.

When a session is started, the model has to be read from storage into memory; when the contents of a whiteboard is persisted (either when the session is terminated or at an intermediate point), the model has to be written back to storage. In the on-line scenario, models are stored for the long term in a storage backend, typically a Version Control System (VCS, see e.g. Git or Subversion). The core MONDO Collaboration Server has a storage-specific plug-in that communicates with this storage backend (using standard VCS protocols) to check out current versions of models and persist new model versions updated on whiteboards. An additional domain-specific plug-in, aware of the
D4.4 – Prototype tool for collaboration

metamodel and the file format of the modeling language, provides the capability of parsing the model files into in-memory EMF objects and serializing them back.

Modeling tools present users with an efficient and usable way for viewing and editing models conforming to a particular modeling language. The typical way we foresee for providing on-line collaborative modeling tools on top of the MONDO Collaboration Framework is to implement them as web applications that can be accessed by users via their desktop or mobile web browsers. Such a web app, which can optionally be deployed on the same machine as the MONDO Online Collaboration Server, consist of a core model viewer editor logic, and the server-side components of the web UI. The current version of the prototype uses Eclipse RAP [3] as its Web UI technology.

The modeling tool employed by a user can communicate with the MONDO Collaboration Server through an API. One part of the API exposes session manager functionality: joining, leaving and otherwise controlling (e.g. persisting to storage) whiteboards sessions. Furthermore, the same API allows direct access to the front model of the user within a whiteboard that the user has joined: the modeling tool can request a snapshot of current model facts (see Sec. 1.3), subscribe for notifications on changes to the set of model facts, and submit changes (which may be rejected if they violate locks or write access rules). Our design contains plans to expose a future version of the session management and whiteboard access API as a remote invocation interface using appropriate technologies such as Apache Thrift [2], which would allow modeling tools to connect to a remote collaboration server.

![Figure 12: Simplified Online Collaboration Architecture](image)

For the sake of simplicity, the actual prototype was developed with a more restricted architecture (see Fig. 12), as it is intended merely to demonstrate the concept of online collaboration with fine-grained access control and locking. To avoid developing a general purpose remote model access API (which
D4.4 – Prototype tool for collaboration

1 policy WindTurbine first-applicable {
2     //WTParamManager is not allowed to work with IO
3     rule denyIO deny RW to WTParamManager {
4         query "test.project.objectIO"
5     }
6     //WTParamManager is allowed to work with parameters
7     rule permitParams permit RW to WTParamManager {
8         query "test.project.objectParams"
9     }
10    //WTParamManager is not allowed to modify other objects
11     rule denyIO deny W to WTParamManager {
12         query "test.project.objectEObject"
13     }
14 }

Listing 3: Rule definitions

is beside the main goals of this Work Package), we have simply used the EMF model access API, which is strictly local. This results in the limitation that the modeling tool must be installed directly on the MONDO Online Collaboration Server. Going along with this limitation, there was no reason to split the domain metamodel-specific components into two (one for the core model management server, one for the web application UI); the prototype uses a single domain-specific web-based model editor component.

3.3 Case Study

We demonstrate the capabilities of online collaboration related to the Wind Turbine case study. The (simplified version of the) metamodel and instance models are introduced previously in Sec. 2.4. In a real world scenario, employees are fine-tuning of system parameters in real time. It requires an online collaboration session that on-the-fly updates the models. We extend the existing roles with WT Param Manager who is responsible for setting only the parameters of the system but inputs and outputs are hidden from him/her while read access to any other objects is still provided. Rules and related query definitions are listed in Listing 3 and Listing 4 respectively.

3.3.1 Security lens in the on-line case

Users that have logged in to the web access point of the online collaboration server see a list of models that they are allowed to open (see Fig. 13 for sample UI). A whiteboard session is initiated transparently when the first user opens a given model. Each time when a user opens the same model, they connect to the same whiteboard, and the GET method of the security lens is executed to create a new front model for the newly connected user. This model is filtered and obfuscated from the user based on the security access rules defined in the repository.
D4.4 – Prototype tool for collaboration

Listing 4: Queries for rules

```java
1 // Queries all input and output objects in the model
2 pattern objectIO(io) {
3    WTCtrlOutput(io);
4 } or {
5    WTCtrlInput(io);
6 }
7 // Queries all objects in the model
8 pattern objectEObject(obj) {
9    EObject(obj);
10 }
11 // Queries all parameter objects in the model
12 pattern objectParams(param) {
13    WTCtrlParam(param);
14 }
```

Figure 13: Selecting a model to open

Our scenario describes a user with WT Param Manager role. The interface and the model fragment as seen by WT Param Manager is depicted in Fig. 14 while the view of WT Owner (including the complete model) is represented in Fig. 15.
Figure 14: Online view of a WT Param Manager
Figure 15: Online view of a WT Owner
In the on-line case, \texttt{PUTBACK} is executed as a live transformation. This means that each time a front model is changed by the respective user, \texttt{PUTBACK} processes the changes only, instead of full front model. Modifications are evaluated on the fly, resulting in acceptance or rejection. If a modification is accepted and applied on the gold model, \texttt{GET} is called (likewise as a live transformation) for the rest of the front models in order to propagate the change to every user in the current session.

\textit{WT Param Manager} uses the web interface to request two operations on the model as follows:

- Change the value attribute of \texttt{GeneratorCoolingTempLimit1} from 100 to 120 (see Fig. 16a for sample user interface).
- Create a new system parameter \texttt{GeneratorCoolingTempLimit3}.

![Image](image1.png)

(a) Editing an existing System Parameter   (b) Failed attempt to create a new System Output

Figure 16: Modifying the model

One by one, the Modeling Tool UI expresses these two changes as modification commands on the model access API, which are actually applied to the front models at the \textit{Server Core}. Subsequently, the lens associated with the front model translates both changes to the \textit{gold} model via \texttt{PUTBACK}. The first change is accepted, as it conforms to the write access privileges of \textit{WT Param Manager}; it will then be propagated via \texttt{GET} to the other front models. The second change, however, is rejected at this stage; the gold and front models are rolled back to their state before the modification, and an error message is displayed for the user (see Fig. 16b for sample user interface).
4 Setup and Management of the Prototype Tool

4.1 Security Lens Daemon Initial Setup

A continuously running security lens daemon is realized by the Eclipse Feature org.mondo.collaboration.security.lens.offline.server.feature, which must be installed in an Eclipse instance running on the collaboration server hosting the VCS. The recommended way to obtain such an Eclipse server is to use the pre-packaged MONDO Server Product (see [10]).

The configuration of the Online Server entails the following steps:

1. Configuration options similar to the following should be provided to the Eclipse server product either as command line arguments, or (recommended) as separate lines in the .ini file of the Eclipse product (see mondo-server.ini for the integrated MONDO Server). All of these are actually Java VM arguments, and thus should appear strictly after the -vmargs entry.

   - The server daemon must continuously run and listen for requests until explicitly stopped; ensure that -Dosgi.noShutdown=true is set.
   - The HTTP(S) server must listen on designated ports. Use options such as -Dorg.osgi.service.http.port=8080 to specify a HTTP port and -Dorg.osgi.service.http.port.secure=8443 for HTTPS.
   - As the lens transformation is memory-intensive when active, allow the server to allocate ample RAM if needed. This upper bound is set by the -Xmx parameter, e.g. use -Xmx4096m to raise the limit to 4GB RAM.
   - An optional but highly recommended feature is to set up logging using Apache Log4j. One can specify a custom logging configuration in a log4j.properties file, and then instruct the server to perform logging accordingly by the argument -Dlog4j.configuration=file:/path/to/log4j.properties (substitute with actual path). A sane log configuration would be the following:
     - Append all error messages to a first log file.
     - Append all info-level or more severe messages of the logger org.mondo.collaboration.security.lens to a secondary log file. This is useful to find out details whenever the model-level lens is executed, e.g. to learn why a given write attempt was rejected.
     - Optionally, collect debug-level messages from the same logger org.mondo.collaboration.security.lens to a tertiary log file. The role of this very verbose log is to aid the developers of the MONDO Collaboration Framework in debugging problems, should they occur.

2. Ensure that the Eclipse server product is running on the server; e.g. run the entry point (see run-server.sh for the integrated MONDO Server) manually and/or add it to a system startup script.
4.2 Version Control System Initial Setup

The VCS must be installed on the same machine as the lens daemon above. The following steps describe how to create a new MONDO repository, consisting of a gold SVN repo and one associated front repo per user, on a Linux server. The steps from number 4 onwards will have to be repeated if additional MONDO repositories are required.

1. Ensure that SVN is installed on the server and is served through Apache Web Server.

2. Ensure that the offline collaboration components are installed into a designated folder on the server, with file system permissions set (using e.g. `chown`) so that the user impersonated by the Apache Web Server can read and write the installation. The required files can be obtained either by
   - checking out the Git source repository [https://github.com/FTSRG/mondo-collab-framework](https://github.com/FTSRG/mondo-collab-framework) and copying the contents of `scripts/mondo` into the target directory, or
   - downloading a Git snapshot archive from [https://github.com/FTSRG/mondo-collab-framework/archive/master.zip](https://github.com/FTSRG/mondo-collab-framework/archive/master.zip) and unpacking the contents of `scripts/mondo` into the target directory.

3. Edit the main configuration file `config/global-config.properties` to configure the VCS integration components, by specifying the following entries:
   - `SVN_PATH_OS=/svn` is a local file system path where SVN is configured to store repositories. The gold and all front repositories will be created here as subfolders.
   - `URL='http://svn.mycompany.com/'` is the address of the SVN host, as visible to clients.
   - `SVN_URL_PATH=svn` is the URL path suffix of the server that is mapped to SVN.
   - `LENS_DAEMON_PORT=8080` shall specify the HTTP port of the locally running Eclipse server that hosts the daemon (must be consistent with the JVM argument `-Dorg.osgi.service.http.port=` given above).
   - `ADMIN_USER=admin` and `ADMIN_PWD=adminPW123` are the credentials of a superuser that can directly access all SVN repositories. These credentials are used by the scripts to propagate changes between the gold and front repos.
   - `APACHE_USER=apache` or `APACHE_USER=apache2` indicates the version of Apache Web Server used, which determines the user impersonated by SVN. Use the former if SVN is served by Apache Server 1.x and the latter in case of Apache Server 2.x.

4. Choose a (unique) name for the new MONDO repository, such as `my-repo-name`. All front repositories deriving from the gold repo will likewise inherit this name.

5. Open a terminal in the `scripts home` (`scripts/mondo/scripts`) folder, and with superuser privileges (e.g. via `sudo`) run `./init-repository.sh my-repo-name` to create the gold SVN repository with appropriately configured hooks, as well as basic configuration items associated with the repository.
6. Edit the newly created repository-specific configuration file config/my-repo-name/config.properties to configure the new MONDO repository, by the specifying following entries:

   • **GOLD_REPO_NAME**=my-repo-name is the a name to this repository; do not modify.
   • **MODEL_EXTENSIONS**=wtspec4m is the file extension that denotes model files in the repository. The model-level security lens will be invoked for these files. Multiple file extensions can be indicated in a comma-separated list.
   • Finally, some information on the repository layout is required: we need to tell the lens where to find the policy file, the lock file, as well as the associated query file. All of these paths are relative to the repository root. For easier modifications in the future, we recommend to develop these files in an Eclipse project, designate a location for the project within the repository, and set up the paths accordingly. The syntax of these properties is the following (note that the line breaks after the equation marks were added only as document formatting, and should not be part of the configuration file):

   ```
   PATH_TO_ACCESS_CONTROL_RULES_FROM_REPOSITORY_ROOT= /x/y/rules.macl
   PATH_TO_LOCK_RULES_FROM_REPOSITORY_ROOT= /x/y/lock.mpbl
   PATH_TO_ACCESS_CONTROL_AND_LOCK_QUERIES_FROM_REPOSITORY_ROOT= x/y/queries.eiq
   ```

7. As a superuser with access to the newly created gold repository, perform an initial commit consisting of the access control policy file, a lock file, and the associated query definition file. This consist of the following steps:

   (a) Create a new temporary directory on the server, and open a command terminal there.
   (b) Check out the currently empty repository by issuing the command line `svn co "http://127.0.0.1/svn/my-repo-name" --username "admin" --password "adminPW123" --non-interactive --quiet`, using the appropriate (local) SVN URL of the gold repository and the user name and password of an SVN superuser permitted to directly access the gold repository.
   (c) Place an initial access control policy (.macl) file, a lock (.mpbl) file, and the associated query definition (.eiq) file in the newly created working copy folder (e.g. my-repo-name). Make sure that the repository-relative paths and file names of these files are consistent with the contents of config.properties.
   (d) Change the current directory to the repository working copy (e.g. `cd my-repo-name`).
   (e) Issue the command line `svn add --force * --auto-props --parents --depth infinity -q` to mark all changes as part of the commit.
   (f) Perform the commit by issuing `svn commit -m "initial commit" --username "admin" --password "adminPW123"`. Once again substitute actual credentials, and optionally a commit message.
   (g) Optionally delete this temporary working copy, as it will not be needed anymore.
4.3 Version Control System Management and Maintenance

The Offline Collaboration Server contains a number of scripts for administrative and management tasks. Shell access on the server is required; in the following, we assume a console that is open in the scripts home folder, and has superuser privileges (use e.g. `sudo`).

- `init-repository.sh <gold repository name> [--delete-gold]` initializes a gold SVN repository. If `--delete-gold` is specified, the gold repository will be deleted first if it already exist.

- `add-front-repository.sh <gold repository name> <front repository name> <user name>` creates a new front repository to an existing gold repository and associates it with the given user.

- `get-front-repository.sh <gold repository name> <user name>` finds the name of the front repository associated with the given user.

- `delete-front-repository.sh <gold repository name> <front-repo-name> [--force]` removes the given front repository. If the action is forced, no confirmation will be asked.

- `reset-front-repository.sh <gold repository name> <user name>` resets a potentially corrupt front repository by reconstructing its entire version history from the gold repository.

- `reset-front-repositories.sh <gold repository name>` resets all front repositories by reconstructing their entire version history from the gold repository.

- `wipe-repositories.sh <gold repository name> [--force]` removes the gold and front SVN repositories. If the action is forced, no confirmation will be asked.

- `lookup-gold-repository.sh <gold repository URL>` finds the name of the gold repository with the given public URL.

4.4 Online Collaboration Server

Online collaboration is an optional feature; but if provided, it must be located on the same machine as the lens daemon and the VCS.

The Online Collaboration Server is realized by the Eclipse Feature `org.mondo.collaboration.offline.management.server.feature`, which must be installed in an Eclipse instance. It is integrated by default in the pre-packaged MONDO Server Product (see [10]).

The configuration of the Online Server requires that the following configuration option is provided to the MONDO Server Product as a separate line in the `mondo-server.ini` file. The `-Dmondo.scripts.folder=/path/to/scripts/mondo/scripts` option should indicate the location of the scripts home folder (see Sec. 4.2) in the local file system. As it is actually a Java VM argument, it should appear strictly after the `-vmargs` entry.
4.5 Collaboration Framework Remote Management

Remote management is an optional (albeit recommended) feature; but if provided, it must be located on the same machine as the lens daemon and the VCS. The remote API is protected by MONDO authentication and access control (see [10]); this implies that for every SVN user that has a front repository, we assume a MONDO user with the same name must exist; otherwise they will not be able to properly use the remote API.

The Collaboration Framework Remote Management Server is realized by the Eclipse Feature org.mondo.collaboration.security.online.server.feature, which is integrated by default in the pre-packaged MONDO Server Product (see [10]).

Once that the MONDO Server product has already been set up to act as the lens daemon (see Sec. 4.1), configuring the remote management servlet entails the following additional steps:

1. (Can be skipped if online collaboration has already been set up.) The following configuration option should be provided to the MONDO Server Product as a separate line in the mondo-server.ini file. The -Dmondo.scripts.folder=/path/to/scripts/mondo/scripts option should indicate the location of the scripts home folder (see Sec. 4.2) in the local file system. As it is actually a Java VM argument, it should appear strictly after the -vmargs entry.

2. Add MONDO Users that are to be allowed to access the MONDO APIs by connecting to the server via a MONDO Client Eclipse Product. (Skip this step if MONDO Users have already been created). See [10] for more information on MONDO Users, the remote management API, the Client Products, and the Console Commands.

   (a) Ensure that the server is run with arguments (or .ini entries) containing -Dartemis.security.enabled=false, at least temporarily, since no users have been defined yet.

   (b) Start the remote client product, and issue the following commands on the (OSGi) Console of the Eclipse instance:

      i. usersConnect http://127.0.0.1:8080/thrift/users. Substitute the host name and HTTPS(S) port of the management server appropriately.

      ii. For an administrator user called admin with password adminPW123, issue usersAdd admin Administrator true adminPW123. and then usersUpdatePassword admin adminPW123.

      iii. For any non-admin user such as alice with password alicePW123, issue usersAdd alice Alice false alicePW123.

   (c) In the server console, issue shutdown and then close to stop the server temporarily.

   (d) Set -Dartemis.security.enabled=true in the server configuration so that only authenticated users can manipulate the MONDO API from now on.

   (e) Locate the shiro.ini file that the server product has just created using the following command: find . -name shiro.ini; edit the file and change authcBasic.enabled = false to authcBasic.enabled = true. The server is now secure.
(f) Restart the MONDO Server (`run-server.sh`).
References


