Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Advanced Techniques and Tools for Secure Collaborative Modeling

Ph.D. Dissertation

Csaba Debreceni

Thesis supervisor:
Prof. Daniel Varro, D.Sc., Ph.D.

Co-supervisors:
Gabor Bergmann, Ph.D.
Istvan Rath, Ph.D.

Budapest
2019

Csaba Debreceni

June 2019

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

H-1117 Budapest, Magyar tudósok körútja 2.

Declaration of own work and references

I, Csaba Debreceni, hereby declare that this dissertation, and all results claimed therein are my own work, and rely solely on the references given. All segments taken word-by-word, or in the same meaning from others have been clearly marked as citations and included in the references.

Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Debreceni Csaba kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2019. 06. 12.

Debreceni Csaba

Summary

The advance of information technology allows the engineers to build even more complex systems. However, numerous challenges have arisen due to the increasing level of complexity like the effective development of such systems or the evaluation of their correctness.

Model-based systems engineering (MBSE) envisions a process to design detailed system models through several well-defined abstraction and refinement steps. MBSE enables to detect design flaws early and generate various artifacts automatically from high-quality system models.

Complex systems engineering projects are often carried out collaboratively to meet the aggressive delivery schedules. *Collaborative modeling* involves multiple engineers working together to develop system models concurrently. Modeling artifacts are traditionally developed either in an offline or online manner. In *offline collaboration*, engineers check out an artifact from a repository into a local copy and send back their local changes. In *online collaboration*, engineers may simultaneously edit a model which are immediately propagated to all other users.

Unfortunately, traditional approaches for managing concurrent source-code-based software development do not naturally extend to concurrent model-based development of systems. Therefore, this dissertation is focusing on the following challenges of collaborative modeling: (i) Access control management – What is an engineer allowed to see and modify in a detailed system model? – (ii) Handling and preventing conflicts – How to prevent or resolve numerous contradicting changes introduced by different engineers (e.g. modifying an element that is deleted by another engineer)? – (iii) View models – How to define abstract views of the detailed systems containing only domain-specific information?

My contributions of the thesis are the following: (a) I proposed a generic modeling language to capture fine-grained access control policies and I realized a framework to efficiently evaluate the policies in online and offline scenarios. (b) I integrated the enforcement of high-level fine-grained access control policies into a provenly secure collaborative architecture. (c) I proposed a fine-grained property-based locking technique to avoid conflicts and an automated three-way model merge technique to resolve conflicts. (d) I proposed a novel technique of bidirectional synchronization of view models where the forward incremental synchronization is achieved by unidirectional derivation rules while the backward propagation of changes is generated using logic solvers.

The contributions of my research have been developed within several international research projects: MONDO, TRANS-IMA, CERTIMOT and CONCERTO. The contributions are implemented within the MONDO Collaboration Framework. Industrial partners of the MONDO project (e.g. IK4-Ikerlan, Uninova) have successfully applied the framework in their development processes.

Összefoglaló

A hatalmas technológiai fejlődés lehetővé tette, hogy a mérnökök egyre komplexebb rendszereket készíthessenek. A komplexitás azonban számtalan újabb kihívást állított a mérnökök elé, mint az átlátható és hatékony tervezés vagy a rendszerek helyességének tesztelése.

A modell alapú rendszerfejlesztési (MBSE) paradigma lehetővé teszi, hogy a rendszert precízen modellezzük pontosan definiált absztrakciós vagy finomítási lépéseken keresztül. A részletes rendszermodellek segítségével automatikusan generálhatunk forráskódot, illetve a tervezési hibák már a korai fázisokban azonosíthatóak.

Ahhoz, hogy az egyre komplexebb rendszertervek a szigorú határidőkre elkészülhessenek, több mérnök, esetleg mérnök csapatok együttes tervezési munkájára van szükség. Azt a folyamatot, amikor a rendszermodelleket egyszerre több mérnök fejleszti, a szakirodalom kollaboratív modellezésnek nevezi. Ebben az esetben a modellek egy közös helyen érhetőek el. Az egyes mérnökök külön-külön hozzáférhetnek a modellhez offline vagy online formában. Offline esetben a mérnökök a modell egy másolatát letöltik a saját számítógépükre, ahol módosítják, majd a változtatásokat visszatöltik. Online esetben a mérnökök egyszerre és folyamatosan módosítják az eredeti modellt, így minden módosítás azonnal érvényre kerül.

A forráskód alapú szoftverfejlesztés esetén már kiforrott támogatást találhatunk a kollaboratív fejlesztésre, azonban ezen megközelítéseket nem lehet egyszerűen átemelni az MBSE-be. Ezért disszertációm a kollaboratív modellezés során felmerülő kihívások köré összpontosul, ahol is a következő témakörökkel foglalkoztam: (i) *Hozzáférés szabályozás* – Hogyan lehet biztosítani, hogy adott mérnökök mit láthatnak a rendszermodellekből és mit módosíthatnak ezen modelleken? – (ii) *Konfliktus kezelés és megelőzés* – Hogyan lehet megelőzni vagy feloldani, ha több mérnök munkája akár több ezer helyen is ellentétben áll egymással (pl.: valaki töröl valamit, míg egy másik mérnök módosítja azt) – (iii) *Nézeti modellek kezelése* – Hogyan adhatunk a mérnököknek olyan nézetet, amely csak a saját szaktudásuknak megfelelő információt tartalmaznak.

Az értekezésben a következő új tudományos eredményeket közlöm: (a) Definiálok egy általános modellezési nyelvet finom-granularitású hozzáférési szabályok leírásához, illetve algoritmust definiálok ezen szabályok modellek feletti kiértékelésére online és offline esetben is. (b) Bemutatom a magas-szintű hozzáférési szabályok betartását és integrálását egy kollaborációs architektúrába, melynek biztonságosságát formálisan bizonyítom. (c) Tulajdonság alapú zárolási megközelítést mutatok a konfliktusok megelőzésére és tervezési tér bejárásán alapuló konfliktus feloldási stratégiát adok a fennmaradó konfliktusok feloldására. (d) Új megközelítést definiálok a nézeti modellek kétirányú szinkronizációjára.

Az értekezés eredményeit több nemzetközi kutatási projektben is felhasználták, mint MONDO, TRANS-IMA, CERTIMOT és CONCERTO. Az értekezés eredményei szoftver prototípusként is elkészültek és a MONDO Collaboration Framework keretrendszerben kerültek bemutatásra, amit a MONDO kutatási projekt ipari tagjai (pl.: IK4-Ikerlan, UNINOVA) sikerrel alkalmaztak.

Acknowledgements

First and foremost, I would like to express my gratitude to Dr. Dániel Varró, my supervisor. He provided continuous support and guidance over the recent years.

I would like to thank my co-supervisors Dr. Gábor Bergmann and Dr. István Ráth. Without them I would never have succeeded. I am grateful to Prof. Dr. András Pataricza and Dr. István Majzik, for providing financial and other kinds of support for my research during their leadership of the group. I am very thankful to all my former colleagues in the Fault Tolerant Systems Research Group, especially Gábor Szárnyas, Oszkár Semeráth, Márton Búr, András Szabolcs Nagy and others for being excellent team players, co-authors, co-developers and friends.

I would like to express my gratitude for the support of the MTA-BME Lendület Cyber-Physical Systems Reasearch Group project. This research was partially supported by the EU projects MONDO (ICT-611125) and CONCERTO (ART-2012-333053), the Hungarian CERTIMOT (ERC_HU-09-1-2010-0003) project and a collaborative project with Embraer called TRANS-IMA.

Contents

Chapter 1

Introduction

1.1 Motivations

Modeling is considered as one of the most elementary discipline in engineering. Its purpose is to overcome complexity, increase understandability or ease design. When engineering complex and critical cyber-physical systems (like cars or aircrafts), the *model-based systems engineering (MBSE)* follows this concept by envisioning a process starting form the design of detailed system models through several well-defined abstraction and refinement steps. MBSE enables to detect design flaws early and automatically derive source code, documentation or configuration artifacts from high-quality system models. The adoption of MBSE by system integrators (like airframers or car manufacturers) has been steadily increasing in the recent years [WHR14].

Large-scale systems engineering projects are often carried out collaboratively to meet the aggressive delivery schedules while still maintaining a high standard of system correctness and safety. *Collaborative modeling* involves multiple engineers working together to develop system models concurrently. Modeling artifacts are traditionally developed either in an offline or online manner. In *offline collaboration*, engineers check out an artifact from a repository into a local copy and commit local changes to the repository in asynchronous (long) transactions. In *online collaboration*, engineers may simultaneously edit a model in short synchronous transactions which are immediately propagated to all other users. This strategy is similar to online collaborative office tools like Google Docs[GDocs].

As a common industrial practice, system integrators frequently outsource the development of various components to subcontractors in an architecture-driven supply chain where the collaboration between cross-organizational teams is facilitated by sharing models stored in model repositories [Roc+15]. However, effective collaboration is hindered by numerous factors.

Cross-organizational collaboration introduces significant challenges to protect the respective Intellectual Property (IP) of different parties. For instance, the detailed internal design of a component needs to be revealed to certification authorities, but it needs to be hidden from competitors who might supply a different component in the system. Furthermore, certain critical aspects of the system model may only be modified by domain experts with appropriate qualifications.

Access control is a process that grants/denies permission for resources when users attempts access them based on *access control policies*. *Access control management* is responsible to manage policies which are enforced during access control. Due to the lack of model-level access control management support for cross-company collaboration in existing modeling repositories, very strict infrastructure-level security policies are in place at companies, which prevent effective collaboration.

Effective collaborative development requires to prevent interference between teams potentially making updates to the same portion of the system model; ensure that local changes do not cause global inconsistencies; provide views of the system that are relevant to teams.

For code artifacts, these questions have traditionally been addressed by partitioning the code and assigning portions to different teams using file-level locking and textual merging techniques followed by testing/verification procedures such as integration testing or model checking.

Unfortunately, such traditional approaches for managing concurrent code development do not naturally extend to concurrent model-driven development. Partitioning into fixed model fragments is difficult due to the interconnected, graph-like nature of models. Fixed fragments are inflexible when faced with varying modeling tasks. Conflict avoidance techniques such as locking – that allows modifications only by the owner of the lock – lead to over-locking due to the high degree of interdependence between parts of a model. This significantly limits the degree of concurrent development and does not scale with the increasing number of collaborating teams. Model merging and conflict detection can be complex tasks, relying on comparing graphs instead of strings, and the interdependence within a model makes conflicts easy to introduce and hard to resolve. Finally, some model verification and validation techniques are too complex to be executed frequently, making quality control an expensive afterthought.

While traditional VCS frameworks provide efficient support for handling text-based design artifacts, their model-level counterparts require sophisticated techniques. Furthermore, the seamless integration of a collaboration layer with existing toolchains is a key industrial need

1.2 General Collaboration Scenario

Figure 1.1 depicts a general scenario to develop complex system designs collaboratively which introduces the key concepts related to collaboration used throughout the thesis. Similarly to traditional software development, models are stored in a version control system (VCS) on the server side from and a local copy of the model is edited by collaborators on client-side.

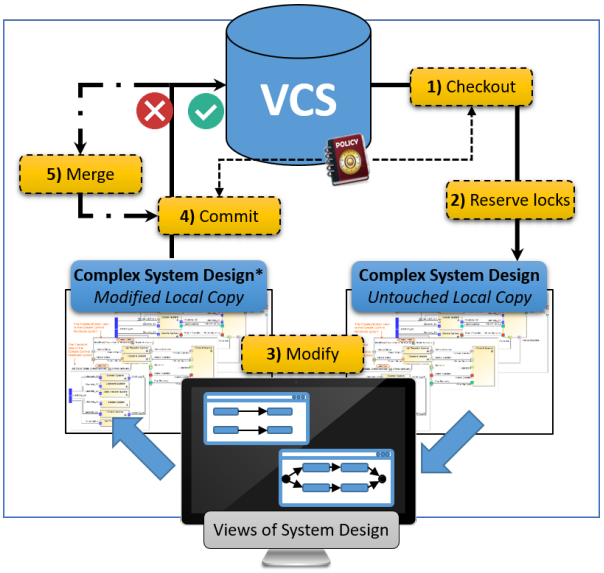


Figure 1.1: Overview of a general collaboration scenario

1. First, a collaborator with specific rights initiates a checkout action to download a local copy of the model. This step includes access control to ensure that collaborators access to those artifacts they are allowed to.
2. Before introducing modifications into the local copy of the model, collaborators can place locks to prevent contradicting modifications on model introduced concurrently by other collaborators.
3. System engineers usually introduce changes via views of the models representing only relevant aspect of the complex system design.
4. Changes are uploaded to the server by initiating a commit action. This step includes access control to ensure that collaborators modify only those artifacts they are allowed to. If the changes satisfies the access control policies, they can be accepted or rejected by the server. A commit is rejected if locks are violated (e.g. another collaborator's lock prevents the acceptance) or conflicts occurred (e.g. another collaborator concurrently modified the same parameter and it was not protected by locks).
5. If conflicts occurred during the commit, the collaborator who initiated the commit has to execute a merge process on client side to eliminate conflicts between their local copy and latest version of the model in the VCS.

1.3 Overview of Challenges

1.3.1 Challenges

Access Control in Collaborative Modeling An increased level of collaboration in a MBSE process introduces additional confidentiality challenges to sufficiently protect the intellectual property of the collaborating parties, which are either overlooked or significantly underestimated by existing initiatives. Existing practices aim to restrict access to the files that store models that often result in inflexible fragmentation of models. In industrial practice, automotive models may be split into more than 1000 fragments, which poses a significant challenge for tool developer. This can be solved by fine-grained access control, where each model element and its features can have its own set of permissions. On the other hand, large industrial models can have millions of model elements, thus explicitly assigning permissions to each of them, as well as maintaining the permissions after changes to the model, would be labor-intensive and error-prone, and would make it difficult to understand the system of privileges.

> **Challenge C-I** How to specify and enforce high-level access control policies during collaborative modeling in a scalable way?

Conflict Prevention and Resolution Enabling a high degree of concurrent edits for collaborators is required to make the traditionally rigid development processes more agile. The increasing number of collaborators concurrently developing artifacts increases the probability of introducing conflicts. Conflicts occur when different collaborators modify the same part of the system model in a contradicting manner (e.g. a collaborator modifies a part that another one deletes). *Conflict avoidance* techniques such as locks try to prevent conflicts by letting the users request that certain engineering artifacts should be made unmodified by all other participants for a duration of time. But it usually leads to unnecessary preventions (locks) which significantly limits the degree of concurrent development and does not scale with the increasing number of collaborating teams. *Model merging* aims to resolve the conflicts, but, it can be complex tasks as the interdependence within a model makes conflicts easy to introduce and hard to resolve.

> **Challenge C-II** How to provide fine-grained conflict prevention and automatized conflict resolution strategies?

Bidirectional Synchronization of View Models Views are key concepts of domain-specific modeling in order to provide specific focus of the system to the engineers with various knowledge and expertise by abstracting the unnecessary details of the underlying model. Usually, these views are represented as models themselves (view models), computed from the source model. On one hand, the efficient forward propagation of changes from the source model to the views is challenging, as recalculating the view from scratch has to be avoided to achieve scalability. On the other hand, the efficient backward propagation of complex changes from one or more abstract view models to the underlying source model resulting in valid and well-formed models is also a challenging task which requires to limit the propagation to a well-defined part of the source model to achieve scalability.

> **Challenge C-III** How to derive and incrementally maintain view models and trace back complex changes to the underlying source models?

1.3.2 Objectives

In this thesis, I propose a secured collaboration framework to address the challenges introduced in Section 1.3.1. The complexity of challenges implies additional objectives to be addressed.

Objective O-I: Fine-grained access control management To address **Challenge C-I** a generic modeling language will be used to capture fine-grained access control policies which needs to be evaluated efficiently in online and offline scenarios. Conflicting access control rules shall be handled where the results should provide a consistent modeling artifact.

Objective O-II: General secure collaboration scheme To address **Challenge C-I** a provenly secure collaborative architecture shall include the enforcement of high-level fine-grained access control policies.

Objective O-III: Conflict resolution and handling To address **Challenge C-II** a fine-grained property-based locking technique will be proposed to avoid conflicts during concurrent modification of models. An automated three-way model merge technique will be proposed to resolve conflicts when locks are unable to prevent conflicts or not considered during the collaboration.

Objective O-IV: Synchronization of view models To address **Challenge C-III** a novel bidirectional synchronization of view models will be proposed where the forward incremental synchronization is achieved by unidirectional derivation rules while the backward propagation of changes is generated using logic solvers.

1.3.3 Contributions

Based on the problems and objectives identified earlier my contribution are formulated as follows:

Contribution 1: Fine-grained access control policies To address **O-I**, I proposed a generic modeling language to capture fine-grained access control policies and I realized a framework to efficiently evaluate the policies in online and offline scenarios.

I propose a query-based approach for modeling fine-grained access control policies. Access control policies consist of rules that allow, obfuscate or deny read and/or write permissions of model parts identified by graph patterns.

However, due to this implicit nature, it is possible that several rules would be in direct conflict with each other, assigning contradictory nominal permissions for some model element. It is also possible for rules to conflict indirectly, if the fragment of the model revealed to a user in not consistent with itself, or with the allowed write operations.

I propose a configurable, multi-stage conflict resolution approach that considers internal and external properties of rules, as well as dependencies among the model parts they apply to, when determining their precedence.

Contribution 2: Provenly secure collaborative architecture To address **O-II**, I integrated the enforcement of high-level fine-grained access control policies into a provenly secure collaborative architecture.

I define a bidirectional model transformations to (i) derive filtered views (*front models*) for each collaborator from the original model (*gold model*) containing all the information and to (ii) propagate changes introduced into these views back to a server in both *online* and *offline* scenarios.

A collaboration scheme between the clients of multiple collaborators and exactly one server is described to support fine-grained access control in offline scenario. The server stores the gold models and the clients can download their specific front models. Modifications, executed by a clients, are submitted to the server and they are accepted if write permissions are successfully checked. Right after the submission, the changes are propagated to the other front model while read permissions are enforced. Finally, clients can downloaded their updated front models.

The scheme is realized by extending *SVN*[SVN] using its hooks. The server and clients are realized as a *gold repository* and multiple *front repositories*, respectively. The *gold repository* contains *gold models*, but it is not accessible to collaborators. Each collaborator is assigned to a specific *front repository* containing a full version history of the front models. Change propagations are maintained between the repositories.

Contribution 3: Property-based locking and Automated model merge using DSE To address **O-III**, I proposed a fine-grained property-based locking technique to avoid conflicts and an automated three-way model merge technique to resolve conflicts.

I introduce the concept of property-based locking where collaborators request locks specified as a property of the model which need to be maintained as long as the lock is active. Hence, other collaborators are permitted to carry out any modifications that do not violate the defined property of the lock. The realization of property-based locking strategy is proposed as a common generalization of existing fragment-based and object-based locking approaches.

Complex properties are described as graph patterns to express structural (and attribute) constraints for a model where the result set, i.e. the matches of graph pattern, can be calculated by pattern matchers or query engines. Only those modifications are allowed that do not change the result set of a list of queries.

I propose DSE-Merge that exploits guided rule-based *design space exploration (DSE)* to automate the three-way model merge. Three-way model merge is applied to DSE problem where the *initial*

model consists of the original model O and two difference models (ΔL and ΔR); the *goal* is that there are no executable changes left in ΔL and ΔR; *operations* are defined by change driven transformation rules to process generic composite (domain-specific) operators; and *constraints* may identify inconsistencies and conflicts to eliminate certain trajectories. The output is a *set of solutions* consisting of (i) the merged model M; (ii) the set of non-executed changes $\Delta L'$,$\Delta R'$; and (iii) the collection of the deleted objects stored in $Cemetery$.

Contribution 4: A novel technique of bidirectional synchronization of view models To address **O-IV**, I proposed a novel technique of bidirectional synchronization of view models where the forward incremental synchronization is achieved by unidirectional derivation rules while the backward propagation of changes is generated using logic solvers.

I introduce an approach where view models are conceptually equivalent to regular models and they are defined using a fully declarative, rule based formalism. Preconditions of rules are defined by graph patterns, which identify parts of interest in the source model. Derivation rules then use the match set of a graph pattern to define elements of the view model. Informally, when a new match of a query appears then the corresponding derivation rule is fired to create elements of the view model. When an existing match of a query disappears, the inverse of the derivation rule is fired to delete the corresponding view model elements.

View models derived by a unidirectional transformation are read-only representations, and they cannot be changed directly. To tackle this problem, we propose an approach to automatically calculate possible source model candidates for a set of changes in different view models. First, the possibly impacted partition of the source model is need to be identified by observing traceability links to restrict the impact of a view modification. Then the modified view models and the query-based view specification are transformed into logic formulae. Finally, multiple valid resolutions of the source model are generated using logic solvers corresponding to the changes of view models and the constraints of the source model from the users can manually select a proper solution.

Overview of contributions with objectives and challenges The overview of the contributions are illustrated in Figure 1.2 together with challenges and objectives.

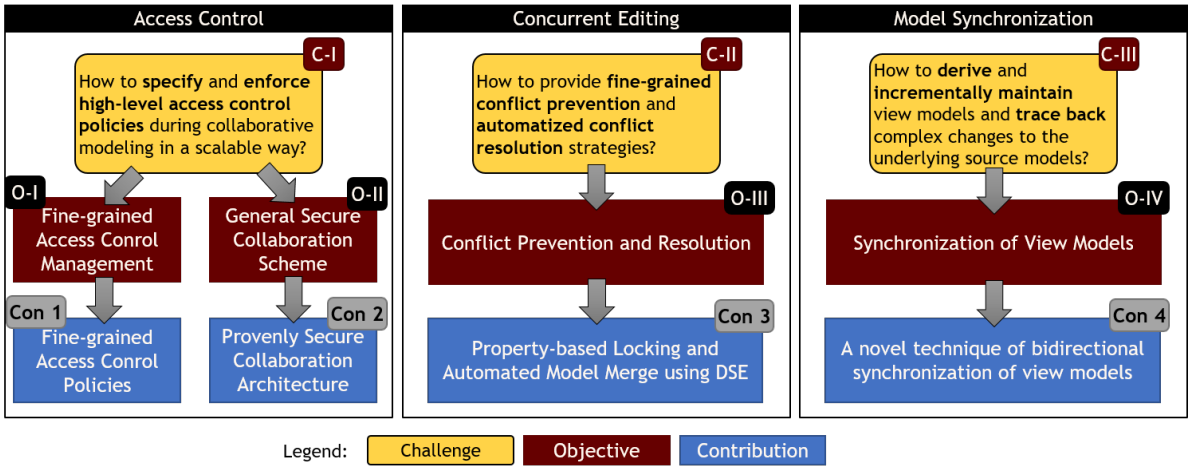


Figure 1.2: Overview of the contributions, objectives and challenges

1.3.4 Research Method

All the presented challenges are driven by industrial needs. In this thesis, I propose novel concepts and algorithms in the field of collaborative modeling to address these challenges. Algorithms and concepts are illustrated and evaluated on case studies carried out in multiple EU projects. I positioned my contributions wrt. state-of-the-art. Scalability evaluation has been carried out for all approaches. The thesis can be categorized as applied research in software engineering. The output of the research is software prototype implementation - which is a major output in mainstream software engineering research.

The contributions of my research have been developed within several international research projects: MONDO, TRANS-IMA, CERTIMOT and CONCERTO. The contributions are implemented within the MONDO Collaboration Framework. Industrial partners of the MONDO project (e.g. IK4-Ikerlan, Uninova) have successfully applied the framework in their development processes.

1.4 Motivating Case Studies

Throughout the paper, challenges are motivated by case studies of development offshore wind turbines extracted from MONDO EU FP7 project [Bag+14] and telecare systems developed in the Concerto European ARTEMIS project[Conc].

1.4.1 Offshore Wind-turbine Control Systems

Offshore Wind Turbine Control Systems where of different artifacts and algorithms for controlling a wind turbine are specified and connected to sensors and actuators to actually operate the physical system.

Real controller systems contains large variety of components, but for sake of simplicity, Figure 1.3 describes the high-level component of offshore wind turbine systems for integrators. A system consists of *heater*, *pump* and *fan* controllers consuming and emitting *signals* to send messages to physical devices of the same type or components.

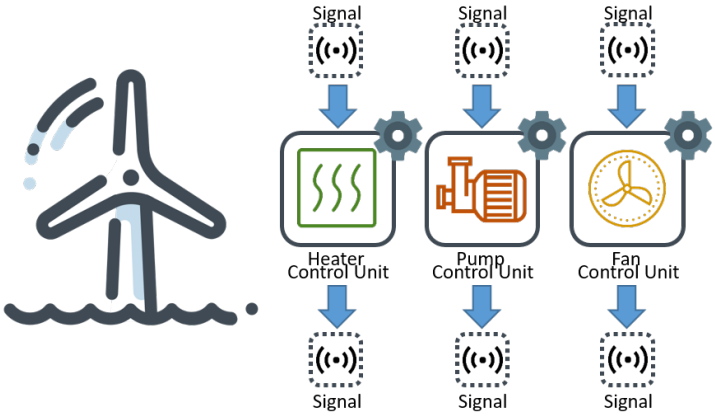


Figure 1.3: Simplified Case Study of Wind Turbine Controllers

Signals are sent after a specific amount of time defined by the *frequency* property and a *documentation* is attached to each of them to clarify their responsibilities. Control units are contained by

modules and organized into a hierarchy of *composite modules* shipped by external *vendors*. However, vendors might treat some of the signals as *confidential intellectual property* and other parties are not allowed access to them.

Access restrictions It is assumed that each control unit type (*heater*, *pump* and *fan* control unit) is associated with a specific person (referred to as *specialist*, but could also be a subcontractor or supplier) who is responsible for maintaining the model of control unit modules of that specific type. Each such user is able to modify the control units that belong to them (along with the signals provided by those modules). Additionally, they are allowed to see all other signals in the model to let them able to consume by other control units. But if the composite module is marked as *protected intellectual property*, they should remain hidden (even from users who could see control units belong to them).

The system integrator company is hosting the wind turbine control model on their collaboration server, where it is stored, versioned, etc. There are two ways for users to interact with it.

Online collaboration. A group of users may participate in online collaboration, when they are continuously connected to the central repository via an appropriate client (e.g. web browser). Each user sees a live view of those parts of the model, that he is allowed to access. Changes need to be propagated on-the-fly between the views of users in short transactions. These transactions contain each modification such as create, update, delete or move. Finally, the collaboration tool has to reject a modification immediately when it violates a security requirement.

The users can modify the model through their client, which will directly forward the change to the collaboration server. The server will decide whether the change is permitted under write access restrictions. If it is allowed, then the views of all connected users will be updated transparently and immediately, though the change may be filtered for them according to their read privileges.

Offline collaboration. In case of offline collaboration, when connecting to the server, each user should be able to download a model file containing those model elements that he is allowed to see. The user should then view, process, and modify his downloaded model file locally. The model shall be developed with unmodified off-the-shelf tool, that need not be aware of collaboration and access control. After the modification, the changes will be uploaded to the server in a long transaction.

Concurrent development. Offshore wind turbine system models are concurrently modified by different engineers to fine-tune the behavior of the system. Engineers may introduce contradicting changes to the model which would cause unexpected behavior. It is necessary that the system model is consistent and well-formed all the time.

1.4.2 TeleCare Systems

TeleCare systems offer remote supervision for health care of elderly and physically less able people by collecting and process data from several sensors available in their home.

A remote health care system is developed in the Concerto project [Conc] where *sensor* devices collect measurement results of a specific type and report them to certain *hosts*. Collecting data and reporting results can be triggered periodically or after certain event occurs.

In the example depicted in Figure 1.4, an environment for *pulse* and *blood pressure* measurement controlled by a *smart phone* is described. Measurements of pulse and blood pressure is measured by the sensors of a mobile phone, which are executed *daily* by the phone timer. After the completion of

measurements triggers the collecting of sensor data into reports. Finally, the blood pressure is sent to the general practitioner of the patient for logging, and signs of hearth failure is sent to hospitals.

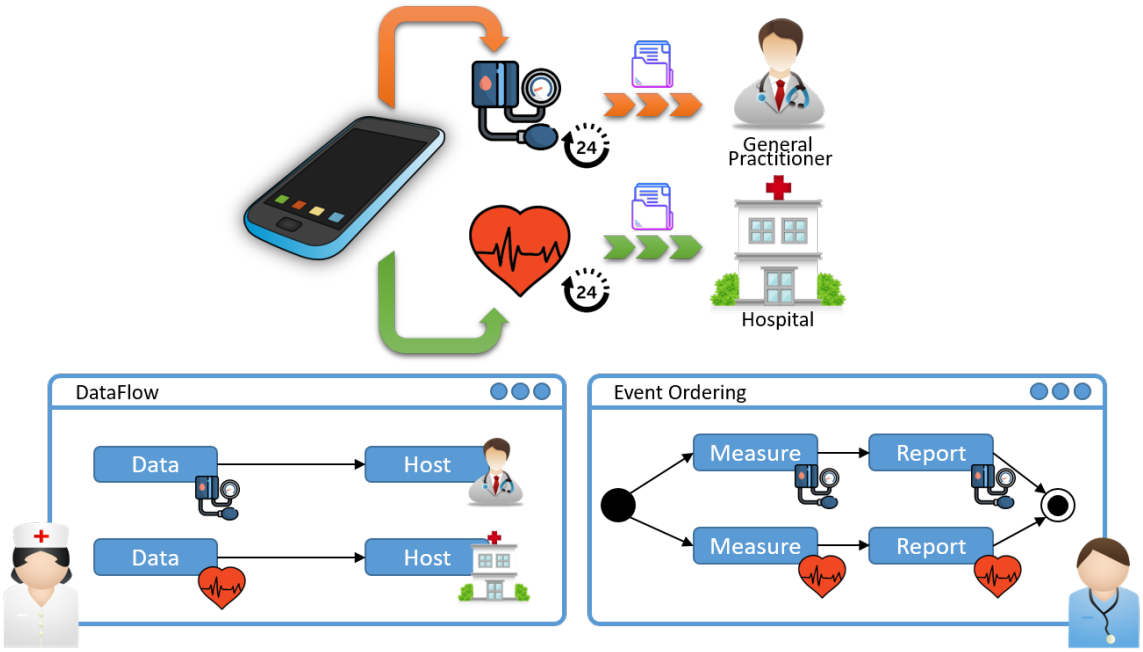


Figure 1.4: A Sample TeleCare Environment for Pulse and Blood Pressure Measurement

Configuration with views. Employees (like *nurses* and other health care workers) are responsible for configuring TeleCare systems, but they may not qualified for setup such complex measurements. To ease the complexity of configuration, *dataflow* and *event ordering* views are required. A dataflow captures where to send certain type of *data*, while event ordering view defines the order of events including the trigger of measurements and reports. Hence, employees should apply changes to the configuration using views instead of modifying the underlying complex setup.

1.5 Structure and Comments

Structure of the dissertation is organized as follows:

- **Chapter 2** provides the preliminaries on modeling foundations necessary for later chapters as well as the terminology that is used throughout the thesis.
- **Chapter 3** addresses **Challenge C-I** to fulfill **Objective O-I**. After discussing the goals, a detailed example is introduced to motivate the modeling language of fine-grained high-level access control policies. Chapter-specific preliminaries are discussed before the access control language is proposed with the formal description of evaluation such policies. Then an initial measurement is presented followed by the discussion of related work. Finally, the detailed description of contributions concludes the chapter.
- **Chapter 4** addresses **Challenge C-I** to fulfill **Objective O-II**. This chapter propose a provenly secure collaboration scheme that enforces the high-level access control policies including the

bidirectional transformation to apply access control rules on modeling artifacts, proving correctness criteria of the scheme and a formal description of a scheme realization.
- **Chapter 5** addresses **Challenge C-II** to fulfill **Objective O-III**. In this chapter an automated model merge technique is proposed for conflict resolution and property-based locks are introduced as fine-grained and generic conflict avoidance technique.
- **Chapter 6** addresses **Challenge C-III** to fulfill **Objective O-IV**. A bidirectional view model synchronization technique is proposed in this chapter.
- **Chapter 7** discusses a prototype collaborative modeling framework developed as part of MONDO Project[Kol+16] and evaluated by the project partners. It contains the prototype implementations of the presented approaches from the previous chapters.
- **Chapter 8** concludes the dissertation by summarizing the novel research results that were achieved by addressing challenges introduced in Section 1.3.1 emphasizing their uniqueness and added values.

Comments The thesis is mainly written in first person plural to emphasize the collaboration with my theses supervisors (and other co-authors). In conclusions after each chapter, I emphasize my own contributions by using first person singular. References to own publications appear as numbers with a letter indicating its type (e.g. [*j*1]), while citations from the bibliography are formatted alphanumerically (e.g., [Sch94]).

Chapter 2

Modeling Preliminaries

In this section, the core concepts of modeling are presented and illustrated with examples to provide a common basis for the theses. However, each contribution might require additional concepts that are introduced in the related chapters.

2.1 Metamodel and Modeling Facts

A domain specification (or domain-specific language, DSL) is defined by a *metamodel* MM that captures the main concepts and relations in a domain. A metamodel is conceived as a set of elementary *model types* to describe the structure of models. Generally, metamodel are consists of classes, references and attributes. Similarly, an *instance model* M that conforms to its metamodel can be decomposed into objects, links and slots.

These concepts are used in *Eclipse Modeling Framework (EMF)*[EMF] which is considered as the *de facto* industry standard for modeling. There are several formal treatments of metamodeling using e.g. *predicate logic*[MSV18] or *type definitions*[Wes14]. This thesis uses a functional formalization of metamodeling as follows.

Definition 2.1 (Metamodel). A *metamodel* $MM{=}\langle C, R, A, \mathit{from}, \mathit{to}, \mathit{owner}, \mathit{isCont}, \mathit{isSuper}\rangle$ is a tuple, where

- C, R, A are set of *classes*, *references*, *attributes*,
- *from* :: $R \to C$ selects the source class of a reference,
- *to* :: $R \to C$ selects the target class of a reference,
- *owner* :: $A \to C$ selects the owner class of an attribute,
- *isCont* :: $R \to$ `BOOLEAN` specifies *containment* reference,
- *isSuper* :: $C \times C \to$ `BOOLEAN` captures supertype relation between classes which is
  - transitive: $\forall c_1, c_2, c_3 \in C : \mathit{isSuper}(c_1, c_2) \wedge \mathit{isSuper}(c_2, c_3) \Rightarrow \mathit{isSuper}(c_1, c_3)$,
  - antisymmetric: $\forall c_1, c_2 \in C : c_1 \neq c_2 \wedge \mathit{isSuper}(c_1, c_2) \Rightarrow \neg \mathit{isSuper}(c_2, c_1)$, and
  - reflective: $\forall c \in C : \mathit{isSuper}(c, c)$ ▪

Notations. To ease the readability, the following shortcuts are introduced:

- $[c.a]$ captures the attribute $a \in A$ owned by the class $c \in C$, formally $\mathit{owner}(a) = c$
- $[a.\mathit{owner}]$ captures the owner class of attribute $a \in A$, formally $\mathit{owner}(a)$
- $[c.r]$ captures the reference $r \in R$ with source class $c \in C$, formally $\mathit{from}(r) = c$
- $[r.\mathit{from}]$ captures the source class of $r \in R$, formally $\mathit{from}(r)$

- $[r.to]$ captures the target class of $r \in R$, formally $to(r)$
- $[r.isCont]$ captures that the reference $r \in R$ is containment, formally $isCont(r)$
- $[c_1 \leftarrow c_2]$ captures that $c_1 \in C$ is supertype of $c_2 \in C$, formally $isSuper(c_1, c_2)$

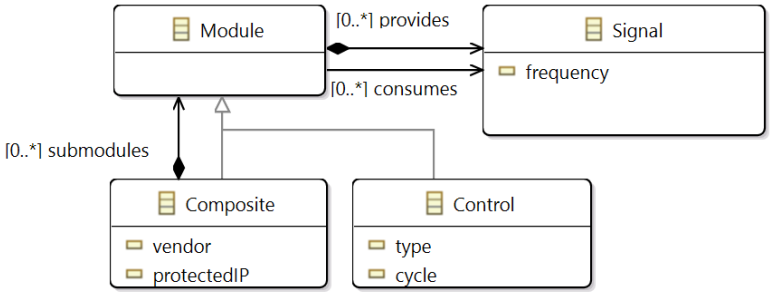


Figure 2.1: Metamodel of Wind Turbine Controllers

Example 1 A simplified metamodel of offshore wind turbine controllers is depicted in Figure 2.1, where boxes represent classes; entries inside boxes mean attributes; edges between boxes ending with closed represents references pointing from the source class (*from*) to the target class (*to*); while edges with open arrows visualize inheritance (*isSuper*) pointing from subclasses to super classes. References starting with black diamonds are classified as containment references (*isCont*).

Wind turbine controller systems build up from modules (`Module`) providing and consuming signals (`Signal`) that send messages after a specific amount of time defined by the `frequency` attribute. Modules are organized in a containment hierarchy of composite modules (`Composite`) shipped by external vendors (`vendor` attribute) which may express protected IP (`protectedIP` attribute), and ultimately containing control unit modules (`Control`) responsible for a given `type` of physical device (such as pumps, heaters or fans: `FanControl`, `HeaterControl`, `PumpControl`, respectively) on certain `cycle` level (*low*, *medium* or *high*).

Definition 2.2 (Instance Model). An *instance model* $M{=}\langle O, D, L, S, src, trg, holder, val, isRoot\rangle$ is a tuple where

- O is a set of *objects*,
- D is a set of *data values* (numbers, text strings and other constants)
- L, S are set of *links* and *slots*,
- $val :: S \to D$ selects the value of a slot,
- $src :: L \to O$ selects the source object of a link,
- $trg :: L \to O$ selects the target object of a link,
- $holder :: S \to O$ selects the holder object of a slot,
- $isRoot :: O \to \text{BOOLEAN}$ decides if an object is root of the model. ▪

Notations. To ease the readability, the following shortcuts are introduced:

- $[o.s]$ captures the slot $s \in S$ held by object $o \in O$, formally $holder(s) = o$
- $[o.a]$ captures the slots of type $a \in A$ hold by the object $o \in O$, formally $\forall s \in \{o.a\} : holder(s) = o \wedge type(s) = a$
- $[s.holder]$ captures the holder object of slot $s \in S$, formally $holder(s)$

- $[o.l]$ captures the link $l \in L$ with source object $o \in O$, formally $\mathit{src}(l) = o$
- $[l.\mathit{src}]$ captures the source object of link $l \in L$, formally $\mathit{src}(l)$
- $[l.\mathit{trg}]$ captures the target object of link $l \in L$, formally $\mathit{trg}(l)$

Definition 2.3 (Typed Instance Model). Function $\mathit{type} :: M \rightarrow MM$ maps an instance model $M\langle O, D, L, S, \mathit{src}, \mathit{trg}, \mathit{holder}, \mathit{val}, \mathit{isRoot}\rangle$ to its metamodel $MM\langle C, R, A, \mathit{from}, \mathit{to}, \mathit{owner}, \mathit{isCont}, \mathit{isSuper}\rangle$ where $\mathit{type}(M) = MM$ iff the following conditions are satisfied:

- Exactly one class from MM is assigned to each object in M:
  $\forall o \in O : \exists c \in C \wedge \mathit{type}(o) = c$.
- Exactly one attribute from MM is assigned to each slot in M:
  $\forall s \in S : \exists a \in A \wedge \mathit{type}(s) = a$.
- Exactly one reference from MM is assigned to each link in M:
  $\forall l \in L : \exists r \in R \wedge \mathit{type}(l) = r$. ▪

Notations. To ease the readability, the following shortcuts are introduced:

- $[o.a]$ captures the slots of type $a \in A$ hold by the object $o \in O$, formally
  $\forall s \in S : \mathit{holder}(s) = o \wedge \mathit{type}(s) = a$
- $[o.a.d]$ captures the slot of type $a \in A$ hold by the object $o \in O$ where value of the slot is $d \in D$, formally $\exists s \in s : \mathit{holder}(s) = o \wedge \mathit{type}(s) = a \wedge \mathit{val}(s) = d$
- $[o.r]$ captures the links of type $r \in R$ which source is the object $o \in O$, formally
  $\forall l \in L : \mathit{src}(l) = o \wedge \mathit{type}(l) = r$
- $[o_s.l.o_t]$ captures the link of type $r \in R$ which source is object $o_s \in O$ while its target is $o_t \in O$, formally $\exists l \in L : \mathit{src}(l) = o_s \wedge \mathit{type}(l) = r \wedge \mathit{trg}(l) = o_t$

Definition 2.4 (Graph-property). Instance models M and metamodels MM must satisfy the following criteria:

- Classes involved by attributes and references exist in MM, formally

$$\forall a \in A, \exists c \in C : a.\mathit{owner} = c$$
$$\forall R \in R, \exists c_f, c_t \in C : r.\mathit{from} = c_f \wedge r.\mathit{to} = c_t$$

- Objects involved by slots and links exist in M, formally

$$\forall s \in S, \exists o \in O : s.\mathit{holder} = o$$
$$\forall l \in L, \exists o_s, o_t \in O : l.\mathit{src} = o_s \wedge l.\mathit{trg} = o_t$$

Definition 2.5 (Typed Conformance Criteria). An instance model $M\langle O, D, L, S, \mathit{src}, \mathit{trg}, \mathit{holder}, \mathit{val}, \mathit{isRoot}\rangle$ *conforms* to a metamodel $MM\langle C, R, A, \mathit{from}, \mathit{to}, \mathit{owner}, \mathit{isCont}, \mathit{isSuper}\rangle$ (denoted by $M \triangleright MM$) iff

- Instance model M is typed of MM, formally
  $\mathit{type}(M) = MM$
- Classes of the source and target object of a link are correct, formally
  $\forall l \in L : \mathit{type}(l).\mathit{from} \leftarrow \mathit{type}(l.\mathit{src}) \wedge \mathit{type}(l).\mathit{to} \leftarrow \mathit{type}(l.\mathit{trg})$
- Attribute of a slot is assigned to the class of object holding the slot, formally
  $\forall s \in S : \mathit{type}(s).\mathit{owner} \leftarrow \mathit{type}(s.\mathit{holder})$ ▪

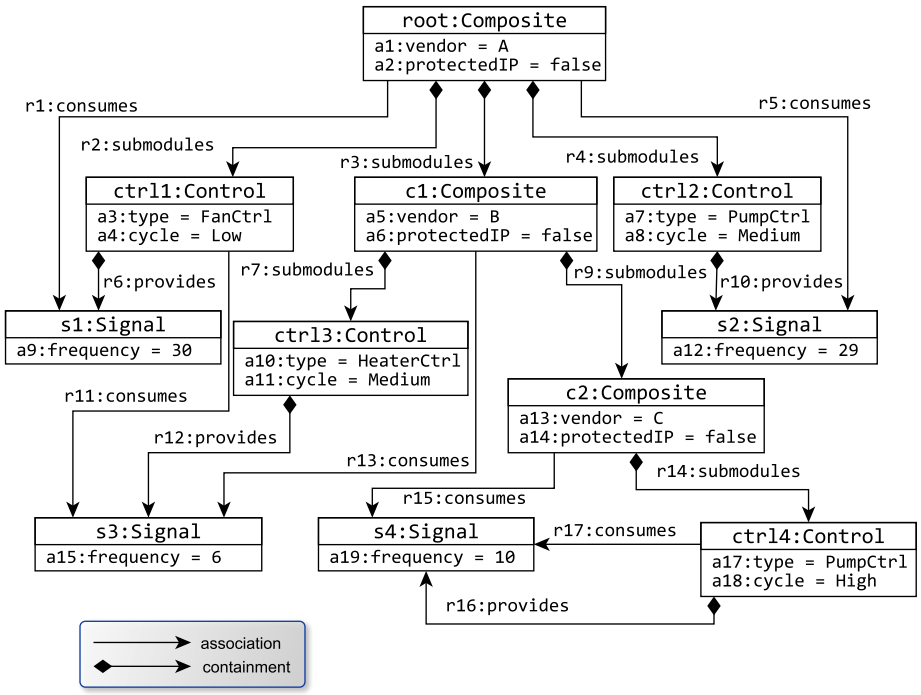


Figure 2.2: Sample Model of Wind Turbine Controllers M_{wt}

Example 2 Figure 2.2 visualizes a sample instance model M_{wt} of wind turbine metamodel. Boxes represent objects (with slots as entries within the box and their class types shown as labels on the tops). Arrows represent links, whereas arrows with diamonds represent links of containment reference types. Each object, slot and link are identified by a unique identifier separated from the type by colon.

Instance model M_{wt} contains a hierarchy of 3 `Composite` modules namely `root`, `c1` and `c2`. There are 4 `Control` units as submodules where `ctrl1` and `ctrl2` are contained by `root`; `ctrl3` is a submodule of `c1` and `ctrl4` is part of `c2`. Each control unit provides a signal (`s1`, `s2`, `s3`, `s4`).

2.2 Consistency of Models

An arbitrary set of model elements does not necessarily constitute a valid model; there may be *consistency criteria* imposed on the elements by the modeling platform to ensure the integrity of the model representation and the ability to persist, read, and traverse models.

Definition 2.6 (Internal Consistency Criteria). A model $M\langle O, D, L, S, src, trg, holder, val, isRoot\rangle$ typed against $MM\langle C, R, A, from, to, owner, isCont, isSuper\rangle$ where $M \triangleright MM$ is internally consistent if the following criteria are satisfied:

- Non-root objects are referred by existing containment references, formally

$$\forall o \in O : \neg isRoot(o) \Rightarrow \exists l \in L : type(l).isCont \wedge l.trg = o$$

- Objects are contained by at most one containment reference, formally

$$\forall l_1, l_2 \in L : l_1.trg = l_2.trg \wedge l_1.isCont \wedge l_2.isCont \Rightarrow l_1 = l_2$$

Industrial modeling platforms (like [EMF]) may specify additional consistency criteria like *opposite links*, *multiplicity constraints* etc. Approaches in the dissertation can be naturally extended with additional criteria but Def. 2.6 is enough for the presentation of core concepts.

Example 3 If we remove the reference `s1:submodules` from model M_{wt} the remaining model will be invalid as non-root object `ctrl1:Control` will be not referred by any containment reference. Moreover, if we create a new `:provides` reference from `root` to `s1` the modified model will be invalid again because `s1` is already referred by another containment reference: `p1:provides`.

2.3 Model Queries as Graph Patterns

Model queries are formulae[TR05] over models, in other terms, they are declarative descriptions of a read-only computation. Graph patterns have chosen as the query formalism[NNZ00; Ujh+15], since they are helpful in capturing the structural relationships between objects.

A *graph pattern* consist of a parameter list and structural or numerical constraints against an instance model defined on a given metamodel. Graph patterns can be composed in order to reuse common query parts, and also to express disjunction, negation, and (surpassing the expressive power of first-order formulae) *transitive closure*.

Patterns can be evaluated as queries over a model where the *pattern match set* denotes the query results. Each match is an assignment of pattern variables where all constraints expressed in the pattern are satisfied. The match set of a pattern may be filtered by binding some parameters of the pattern to values, retaining exactly those matches that assign the specified value to each bound parameter.

Definition 2.7 (Graph Pattern Syntax). A *graph pattern* (or formula) is a first order logic (FOL) formula $\textsc{gp}(\mathrm{v}_1 \ldots \mathrm{v}_n)$ over variables $\mathrm{V} = \{\mathrm{v}_1 \ldots \mathrm{v}_n\}$ defined for a metamodel $MM\langle C, R, A, \textit{from}, \textit{to}, \textit{owner}, \textit{isCont}, \textit{isSuper}\rangle$. A graph pattern GP can be inductively constructed by using atomic predicates:

- class constraints $\textsc{c}(\mathrm{v})$ where C is mapped to a class $c \in C$,
- attribute constraints $\textsc{a}(\mathrm{v}_h, \mathrm{v}_v, \mathrm{v}_s)$ A is mapped an attribute $a \in A$,
- reference constraints $\textsc{r}(\mathrm{v}_s, \mathrm{v}_t, \mathrm{v}_l)$ R is mapped to a reference $r \in R$,
- check constraints $\mathrm{v}_1 = \mathrm{v}_2$ or $\mathrm{v}_1 \neq \mathrm{v}_2$,
- standard FOL connectives $\neg$, $\wedge$, $\vee$ of class, attribute and reference constraints.
- standard FOL quantifiers $\exists$v, $\forall$v. ▪

A simple graph pattern only contains (a conjunction of) atomic predicates. Complex graph pattern can reuse other graph patterns and connect them with atomic predicates using standard FOL connectives.

Definition 2.8 (Graph Pattern Matching). A graph pattern $\textsc{gp}(\mathrm{v}_1 \ldots \mathrm{v}_n)$ can be evaluated on an instance model $M \triangleright MM$ along a variable binding B which is a mapping B ::V$\rightarrow O \cup D$ to objects and values in M denoted by $\left[\!\left[\textsc{gp}\right]\!\right]^M_{\mathrm{B}}$. The truth value of a binding B evaluated on M is in accordance with the truth value of predicates as follows:

- $\llbracket \text{C}(\text{V}) \rrbracket^M_{\text{B}} \equiv 1 : \text{C} \mapsto c \wedge c \leftarrow \mathit{type}(\text{B}(\text{V}))$
- $\llbracket \text{A}(\text{V}_h, \text{V}_v, \text{V}_s) \rrbracket^M_{\text{B}} \equiv 1 : \text{A} \mapsto a \wedge \mathit{type}(\text{B}(\text{V}_s)) = a \wedge \text{B}(\text{V}_s).\mathit{holder} = \text{B}(\text{V}_h) \wedge \text{B}(\text{V}_s).\mathit{val} = \text{B}(\text{V}_v)$
- $\llbracket \text{R}(\text{V}_s, \text{V}_t, \text{V}_l) \rrbracket^M_{\text{B}} \equiv 1 : \text{R} \mapsto r \wedge \mathit{type}(\text{B}(\text{V}_l)) = r \wedge \text{B}(\text{V}_l).\mathit{src} = \text{B}(\text{V}_s) \wedge \text{B}(\text{V}_l).\mathit{trg} = \text{B}(\text{V}_t)$
- $\llbracket \text{V}_1 = \text{V}_2 \rrbracket^M_{\text{B}} \equiv 1 : \text{B}(\text{V}_1) = B(\text{V}_2)$
- $\llbracket \text{V}_1 \neq \text{V}_2 \rrbracket^M_{\text{B}} \equiv 1 : \text{B}(\text{V}_1) \neq \text{B}(\text{V}_2)$
- $\llbracket \neg \text{GP} \rrbracket^M_{\text{B}} \equiv 1 - \llbracket \text{GP} \rrbracket^M_{\text{B}}$,
- $\llbracket \text{GP}_1 \wedge \text{GP}_2 \rrbracket^M_{\text{B}} \equiv min(\llbracket \text{GP}_1 \rrbracket^M_{\text{B}}, \llbracket \text{GP}_2 \rrbracket^M_{\text{B}})$,
- $\llbracket \text{GP}_1 \vee \text{GP}_2 \rrbracket^M_{\text{B}} \equiv max(\llbracket \text{GP}_1 \rrbracket^M_{\text{B}}, \llbracket \text{GP}_2 \rrbracket^M_{\text{B}})$
- $\llbracket \exists \text{V} : \text{GP} \rrbracket^M_{\text{B}} \equiv max\big(\llbracket \text{GP} \rrbracket^M_{\text{B},\text{V} \rightarrowtail x} : x \in O \cup D\big)$
- $\llbracket \forall \text{V} : \text{GP} \rrbracket^M_{\text{B}} \equiv min\big(\llbracket \text{GP} \rrbracket^M_{\text{B},\text{V} \rightarrowtail x} : x \in O \cup D\big)$.

Open formulae (with one or more unbound variables) are treated by introducing an (implicit) existential quantifier over unbound variables. Thus, a pattern like $\llbracket \text{GP}(\text{V}_1 \ldots \text{V}_n) \rrbracket^M_{\emptyset}$, where none of the variables are bound, can be transformed into a closed formula without unbound variables as follows: $\llbracket \exists \text{V}_1 \ldots \exists \text{V}_n : \text{GP}(\text{V}_1 \ldots \text{V}_n) \rrbracket^M_{\emptyset}$.

Definition 2.9 (Pattern Match). A binding B is a *pattern match* (PM) of $\text{GP}(\text{V}_1 \ldots \text{V}_n)$ if it is evaluated to 1 over M, formally $\llbracket \text{GP}(\text{V}_1 \ldots \text{V}_n) \rrbracket^M_{\text{PM}} \equiv 1$. .

Definition 2.10 (Match Set). A match set $\text{MS} = \{\text{PM}_1 \ldots \text{PM}_k\}$ is a set of unique pattern matches $\text{GP}(\text{V}_1 \ldots \text{V}_n)$ evaluated over M, formally $\forall \text{PM}_i, \text{PM}_j \in \text{MS} : \exists \text{V} \in \text{V}, \text{PM}_i(\text{V}) \neg \text{PM}_j(\text{V})$. .

Notation. To ease the readability, the following shortcut is introduced:

- $\text{GP}(e_1 \ldots e_n)$ captures a pattern match PM of the pattern $\text{GP}(\text{V}_1 \ldots \text{V}_n)$ on model M where $\forall e_{i \in \{1..n\}} : \text{PM}(\text{V}_i) = e_i$

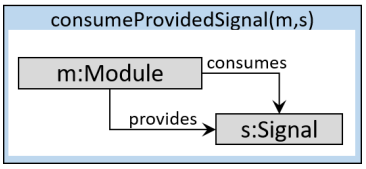


Figure 2.3: Example Graph Pattern

Example 4 A simple graph pattern GP is visualized in Figure 2.3 called `consume ProvidedSignal`. It has two variables `ctrl` and `s` also acting as parameters. The pattern is looking for `Module` and `Signal` pairs of `ctrl` and `s`, where a module provides and consumes the same signal.

After the evaluation of GP on the example model M_{wt}, match set $\text{MS}^{\text{GP}}_{M_{wt}}$ contains one pattern match selecting the tuple of $\langle \texttt{ctrl4}, \texttt{s4} \rangle$.

VIATRA Query VIATRA Query[Ujh+15] provides a declarative textual graph query language. Name of the pattern is presented after `pattern` keyword and parameters are between parentheses (`(` and `)`). Pattern bodies defined by braces ({ and }) contain $\wedge$ connection of constraints and $\vee$ connection can be captured using `or` keyword between two pattern body ({}`or`{}). Graph patterns can be composed (`find` keyword) in order to reuse common query parts, and also to express disjunction, negation (`neg` keyword), and *transitive closure* (`+` symbol).

Listing 2.1 presents the graph pattern depicted in Figure 2.3 in VIATRA Query syntax.

```
1 pattern consumeProvidedSignal(ctrl, s) {
2   Module.provides(ctrl,s);
3   Module.consumes(ctrl,s);
4 }
```

Listing 2.1: Graph Pattern to Lock Signals

2.4 Well-formedness Constraints

Low-level internal consistency rules are distinguished from high-level, language-specific *well-formedness constraints*. Well-formedness constraints (also known as design rules or consistency rules) define additional restrictions to the metamodel that the instance models need to satisfy. These type of constraints are often described using OCL[OCL] or graph patterns[Ujh+15]. The difference between the two concepts is that violating the latter kind does not prevent a model from being processed and stored in a given modeling technology.

Definition 2.11 (Well-formedness Constraint). A *well-formedness constraint* WF is defined as negated graph pattern GP, formally $WF \equiv \neg \text{GP}(v_1 \dots v_n)$. ▪

Example 5 Graph pattern `consumeProvidedSignal` depicted in Figure 2.3 also act as well-formedness constraint because modules cannot consume their provided signals in case of a wind turbine system. As `ctrl4` controller provides `s4` signal via `r16` reference and also consumes it via `r17` the model is ill-formed and the GP selects the tuple of $\langle$`ctrl4`, `s4`$\rangle$.

2.5 Model Transformation

Transformation rules capture complex manipulations of instance model including the combination of creating or removing objects, links or slots. A rule a pair of *precondition* and *action* parts where precondition determines the applicability of the rule while the action captures model manipulation.

A transformation rule is executable if its precondition is satisfied by the current state of instance model and the execution of an rule involves the application of action part resulting in a modified instance model.

Definition 2.12 (Transformation Rule). A transformation rule RULE $= \langle precondition, action \rangle$ consists of a precondition and an action part.

- *precondition* is a graph pattern (GP) determining the applicability of the rule
- *action* $:: M \times \text{PM}_{precondition} \rightarrow M$ is the action capturing model manipulation function based on a pattern match of *precondition*.

▪

Notation. To ease the readability, the following shortcuts are introduced:
- RULE.*precondition* captures *precondition* of RULE⟨*precondition*, *action*⟩
- RULE.*action* captures *action* of RULE⟨*precondition*, *action*⟩

Definition 2.13 (Activation). An activation $act = \langle \text{RULE}, \text{PM}_{\text{RULE.}precondition} \rangle$ on model M is formed as a pair of rule RULE⟨*precondition*, *action*⟩ and a match PM of its precondition evaluated on M. ▪

Definition 2.14 (Execution Step). Execution step is the execution of an activation $act = \langle \text{RULE}, \text{PM}_{\text{RULE.}precondition} \rangle$ on the current state of the model (M_i) which involves the call RULE.*action*$(M_i, \text{PM}_{\text{RULE.}precondition}) = M_{i+1}$ resulting a modified model. Execution steps are denoted as $Step = M_i \overset{act}{\rightarrowtail} M_{i+1}$ ▪

```
1 precondition:
2   pattern consumeProvidedSignal(ctrl, s) {
3     Module.provides(ctrl,s);
4     Module.consumes(ctrl,s);
5   }
6 action:
7   delete ctrl.consumes.s
```

Listing 2.2: Example transformation rule

Example 6 Listing 2.2 captures a transformation rule RULE where
- RULE.*precondition* uses the graph pattern `consumeProvidedSignal` to select module and signal pairs where the module consumes its provided signal
- RULE.*action* removes the `consumes` link between the selected module and signal.

On the example wind turbine model M_{wt}, this rule has one activation *act*⟨RULE, GP(`ctrl4`, `s4`)⟩ as the graph pattern selects the module `ctrl4` and signal `s4`. Firing the rule executes the RULE.*action* which removes the `consumes` link between `ctrl4` and `s4` identified by `r17:consumes`.

Definition 2.15 (Transformation and Transformation Engine). A transformation TR consists of a set of transformation rules ($\text{RULE}_1, \text{RULE}_2 \ldots \text{RULE}_n\}$) that a transformation engine TE executes to incrementally derive an updated target model M'_T from a source and target model M_S, M_T.

$$\text{TR} = \{\text{RULE}_1, \text{RULE}_2 \ldots \text{RULE}_n\}$$

$$\text{TE} :: (M_S, \text{TR}, M_T) \rightarrow M'_T$$

Transformation execution repeatedly fires the rules as follows:
1. finds all the activations of all rules,
2. selects an activation of the rule with the highest priority,
3. executes the activation;

The loop terminates when there are no more activation. ▪

Chapter 3

Fine-grained Access Control Management

3.1 Introduction

In Section 1.3.1, we introduced the challenges of access control in collaborative modeling (**C-I**) which requires fine-grained access control management. This chapter is based on [*c*8] and [*c*12] and the main objective is to propose a generic modeling language that should be used to capture fine-grained access control policies and approaches to evaluate such policies efficiently in online and offline scenarios. In particular, the following goals are aimed to address:

- **G1** *Parameterizable policy language.* The solution must provide a parameterizable fine-grained policy language to define concise access control rules and be able to fine-tune the resolution of conflicting rules.
- **G2** *Deterministic and consistent conflict resolution.* The solution must define a deterministic application of the access control rules to obtain the same effective permissions after every execution where all internal consistency rules are taken into account;
- **G3** *Scalable approaches.* The solution must provide a scalable approach wrt. increasing size of models, size of introduced changes and number of collaborators.
- **G4** *Small maintenance computation.* The solution must provide an approach that requires small maintenance computation which depends on the introduced changes instead of original model size.

Structure Rest of the chapter is organized as follows. Related work is overviewed in Section 3.2. A motivating example is introduced in Section 3.3 and Section 3.4 briefly discusses additional preliminaries. In Section 3.5 proposes the high-level fine-grained access control language. In Section 3.6, we give formal treatment of permissions and propose conflict resolution strategies to derive effective permissions from policies. Finally, Section 3.7 describes the evaluation of our approach. Section 3.8 concludes the chapter by a summary and state the contributions achieved in Fine-grained Access Control Management.

Approach	online/offline	Fine-grained	Conflict resolution	Internal Consistency
File-based Access Control	both	-	priority level	-
XACML [Ge03]	both	+	internal and external priorities	limited
[Abe+07]	both	+	not discussed	+
[JF06]	both	+	only restrictive	-
CDO [CDO]	both	type-specific, but extendable	custom	custom
KeyNote [BFK98]	both	+	not discussed	not discussed
AToMPM [Syr+13]	online	+	not discussed	not discussed
[Far+10]	online	manipulation level	-	-
Our solution	both	declarative graph query	+	generic elementary rules

Table 3.1: Comparison of related approaches

3.2 Related Work

File-based Access Control. Off-the-shelf file systems typically require resources (files and folders) to be explicitly labeled with permissions that take the form of an Access Control List (ACL), or the simplified form user/group/others. An ACL consists of entries (judgments) regarding which user/subject is granted or denied permission for a given operation. Conflict resolution is usually priority-based (first entry applies) within the access control list, and restrictive among type II and type III conflicts (e.g. contents of a hidden folder cannot be seen, regardless of ACLs inside).

File-based solutions can be directly applied to MDE, but cannot provide fine-grained access control, where different parts of a model file have different permissions. Our policies are fine-grained, use implicit rules (so that model elements do not have to be explicitly annotated with judgments, which is difficult to manually maintain as the model evolves), and respect the modeling-specific challenges of consistency (such as permission dependencies of cross-references); all the while being more flexible in the conflict resolution method.

Access Control for XML Documents. A number of standards such as XACML [Ge03] (OASIS standard) provide fine-grained access control for XML documents. These type of documents are similar to models in a way, that they consists of nodes with attributes that may contain other nodes. XACML provides several combining algorithms to select from contradicting policies. Similarly to our solution, it may use external and internal priorities together (*ordered-(deny/permit)-overrides*) or only internal priorities (*unordered-(deny/permit)-overrides*). In [FM04], fine-grained access control is formalized using XPath for XML documents, which claims that the visibility of a node depends on its ancestors, thus when a node is granted access, then access is also granted to its descendants. However, other dependencies are not discussed related to XML Documents, while our approach [*c*12] also considers e.g. cross-references.

Context-aware Access Control RDF Stores. Models can be persisted into triples to store them in triple or quad stores (Neo4EMF[Ben+14], EMF Triple). Graph-based access control is a popular strategy for many RDF stores (4store [4store], Virtuoso , IBM DB2) developed for storing large RDF data. In case of RDF, fine-grained specification of access control permissions are defined at triple level. In [Abe+07], a graph-based policy specification language proposed over SPARQL [PS16], but resolution of contradicting rules are not discussed. Amit Jain et. al. [JF06] propose an access control model for RDF and a two-level conflict resolution strategy that also takes inconsistencies into account similar to our solution. But, it uses only restrictive resolution without any configuration of default values or priorities between rules.

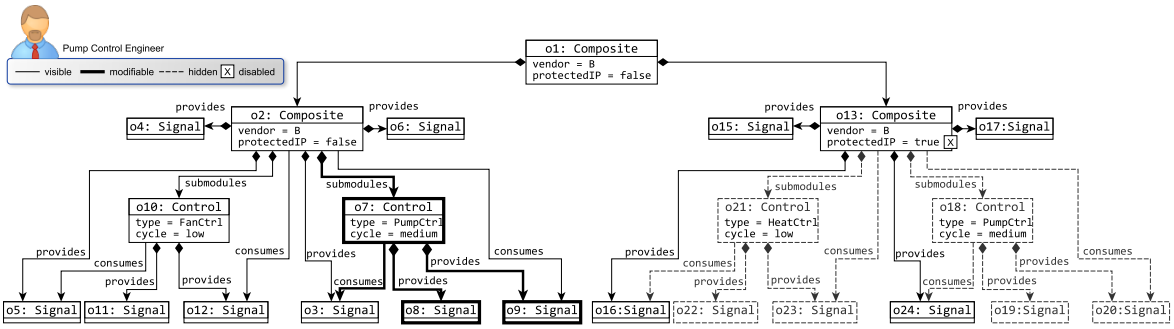


Figure 3.1: Access control for a wind turbine instance model

Collaborative Modeling Environments. Currently, fine-grained access control is not considered in the state of the art tools of MDE such as MetaEdit+[Tol16], VirtualEMF[CJC11], WebGME[Mar+14], EMFStore [EMFStore], GenMyModel [Gen], Obeo Designer Team [Obeo], MDEForge [Bas+14] or the tools developed according to [GBR12]. See also the broader survey in [Roc+15].

The generic framework CDO [CDO] (used e.g. in [Obeo]) provides both online collaboration and role-based access control with type-specific (class, package and resource-level) permissions, but no facility for instance level access control policy specifications. However, there is a pluggable access control mechanism that can specify access on the object level; it should be possible to integrate fine-grained solutions such as the currently proposed system.

The collaborative hardware design platform VehicleFORGE stores their model in graph-based databases and has an access control scheme TrustForge [Cha+13] that uses an implementation of KeyNote [BFK98] trust management system. This system is responsible for evaluating the request addressed to the database, which can be configured in various ways. It supports unlimited permission levels and it is also able to handle consistency constraints by adding them as assertions. Conflict resolution strategies are not discussed. AToMPM [Syr+13] provides fine-grained role-based access control for online collaboration; no offline scenario or query-based security is supported, though. Access control is provided at elementary manipulation level (RESTful services) in the online collaboration solution of [Far+10].

3.3 Motivating Example

In case of a wind turbine system, we assume that each control unit type (`PumpCtrl`, etc...) is associated with a specific person (referred to as *specialist*, but could also be a subcontractor or supplier) who is responsible for maintaining the model of control unit modules of that specific type. Each such user is able to modify the control units that belong to them (along with the signals provided by those modules). Additionally, they are allowed to see all other signals in the model to let them able to create new consume references. But if the composite module is marked as *protected intellectual property*, its `vendor` attribute value and all provided submodules should remain hidden (even from users who could see control units belong to them). (Note, that we omitted `frequency` attribute of signals as these slots have no roles in access restriction.)

Example 7 The user responsible for pump controllers (the *PumpControlEngineer*) can modify the objects and cross-references marked with bold lines, and can additionally see objects and cross-references with weak lines as it is depicted in Figure 3.1. On the other hand, objects and cross-references marked with dashed lines are hidden. Moreover, character *X* in a box means that the attribute next to is neither readable nor writable.

3.4 Terminology for Access Control

Here, a vocabulary for fine-grained access control is briefly introduced; a more detailed treatment can be found in our previous work [*c*14; *j*2].

3.4.1 Model Facts as Assets

In order to provide fine-grained access control, models have to be decomposed into elementary *assets* that can be separately protected. For simplicity, models are considered as a set of elementary *model facts* where objects are *object facts*, slots are *attribute facts* and links are *reference facts*.

Object facts are pairs formed of a model element with its exact type, for each model element object; e.g. o`(o1,Composite)`.
Attribute facts are triples formed of a source, an attribute name and the attribute value, for each (non-default) attribute value assignment; e.g. a`(o1,vendor,A)`.
Reference facts are triples formed of a source, a reference type and the referenced model element for each containment link and cross-link between objects; e.g. r`(o1,submodules,o2)`.

Note that for multi-valued attributes and references, each of the multiple entries at a source will be represented by a separate attribute or reference fact.

We will focus on object facts in the following, but the same formal framework applies to all assets.

3.4.2 Model Obfuscation

Obfuscation is defined as the process of *"making something less clear and harder to understand, especially intentionally"*[Cam17]. The first purpose of obfuscation in programming was to distribute C sources in an encrypted way to prevent access to confidential intellectual property in the code [Jae90]. In a modeling environment, the same concept applies.

A model obfuscation takes a model as input and yields another model as output where the structure of the model remains the same but data values (such as names, identifiers or other strings) are altered. Two data values that were identical before the obfuscation will also be identical after it, but the obfuscated value computed based on a different input string will be completely different. Moreover, all the altered values can be reverted by the original owner of the model using a private key.

Definition 3.1 (Obfuscation). Function *obfuscate* $:: (v, s) \rightarrow \widehat{v}$ takes a data value v and a seed s as inputs and maps the value to $(\widehat{v})$. Function *obfuscate*$^{-1}$ $:: (\widehat{v}, s) \rightarrow v$ is the inverse of *obfuscate* which returns the original data if the same s is used. ▪

3.4.3 Permissions

The above mentioned model facts are the *assets* that the access control policy will protect against *operations* (typically *read* and *write*) that can be performed by the user. The goal of interpreting the

security policy is to come up with an *effective permission function* that assigns a *permission level* to each combination of asset, user and operation.

The assigned permission level takes its value from a *permission lattice* characteristic to the operation. In the simplest case, this is the two-valued lattice $\{deny < allow\}$; in general it is possible to use a more refined lattice instead. In the rest of the paper, we will restrict ourselves to the case where the lattice is a *total order*, i.e. from any two levels, one is considered strictly more restrictive than the other.

For instance, it was suggested in [*c*12] to use $\{deny < obfuscate < allow\}$ as read permission levels, to express the case where an attribute of an otherwise visible object conveys confidential information that shall not be revealed to the given user, but the attribute can not be completely hidden (since it is used as e.g. an identifier or visual label).

As introduced in [*c*8], a sample refinement of the write permission levels could be $\{deny < dangle < allow\}$. Here cross-references with write permission level $dangle$ can not be normally modified by the user, but they can be removed as the side effect of deleting the target object of the reference (if that deletion is permitted), even if the cross-link is not visible to the user. Imagine a traceability link that points from a hidden part of the model to a visible object; the difference between assigning $deny$ or $dangle$ is that the target object can not be deleted by the user in the former case, while its deletion would be allowed (with an invisible side-effect of removing the traceability link) in the latter case.

3.4.4 Consistency of Permissions

After a permission is assigned to each model fact, secured views can be presented to a user which consists of only readable model facts (modulo obfuscation, see above). However, an arbitrary set of model facts does not necessarily constitute a valid model; there may be *internal consistency constraints* (also called referential integrity constraints) imposed on the facts by the modeling platform to ensure the integrity of the model representation and the ability to persist, read, and traverse models. A major goal stated in [*c*14; *c*12] requires that secure models must be synthesized as a set of model facts compatible with all internal consistency rules (see Section 2.2).

Note that we distinguish these low-level internal consistency rules from high-level, language-specific *well-formedness constraints*. Violating the latter kind does not prevent a model from being processed and stored in the given modeling technology, as error markers can be placed on such violations. Thus only internal consistency is essential for access control.

Internal consistency of the filtered view implies that the effective permissions must satisfy a some constraints that are expressed as a *strong dependency* relationship between two values of the effective permission function: a dependant and a dependee. According to the formalization in [*c*12], this relationship is a *Galois connection* as if a lower bound is known for the permission level of the dependant, then a lower bound on the permission level of the dependee is inferred; and conversely, an upper bound on the dependee puts an upper bound on the dependant. For instance, (a) if an asset is writable (write permission level $allow$), then it must also be visible (at least on the read permission level $allow$); (b) if an object is readable at least on the level $obfuscate$, then its containing object (if any) must also be visible (at least on the level $obfuscate$). All of these constraints have a converse, e.g. if an object is not visible (read permission level is at most $deny$), then any contained objects have their read permission level at $deny$ as well.

Apart from the strong dependencies discussed above, the concept of *weak consequence* was also introduced in [*c*12], to express defaults rather than strict implications. For examples, all contained elements and attributes of a visible object should also be made visible by default - unless there is

another reason to deny the read permission. For instance, if the read permission for an object is known to be at most *obfuscate*, its write permission must be at most *dangle* as a strong consequence, but (weak consequence) it should be *deny* by default, unless specifically overridden.

3.5 Access Control Language

To fulfill goal **G1**, a parameterizable rule-based policy language is introduced in [*c*12] which provides the definition of fine-grained access control rules.

Rules use graph-queries to select assets of the model and grant or deny access to them. Rules with higher priority override the others. When none of the rules can be applied to an asset, default read and write permissions will be used. If rules on the same priority level are in conflict, *permissive* or *restrictive* conflict resolution strategy can be applied. In case of permissive resolution, rule that allows an operation takes precedence over a denial rule. But, the restrictive resolution prioritizes denying rules over allowing ones.

Technology independent textual syntax of the language is depicted in Listing 3.1. Each policy has a unique identifier defined after keyword `policy`. Default permission level follows the identifier with keywords `by default`. Permission level allows, obfuscates, dangles or denies certain operation types including read, write or both together only in a valid combination. At the end of a policy, the resolution type needs to be set inside of keywords `with resolution` which can be `permissive` or `restrictive`.

Rules are described inside the policy between the characters { and }. They start with the keyword `rule` followed by their unique identifier, a permission level and one or more roles (denoted by `ID`). List of roles can be constructed based on e.g. users of system, users of VCS, using an external authentication application etc. Optionally, priority level can be set to each rule after the keywords `on priority` where the priority level is an unsigned integer.

Inside a rule bordered by characters { and }, fact selection structure can be defined, similarly to the syntax of LINQ[CHP06], as follows: it references a graph pattern after the keyword `from query` where we assume a graph pattern consists of a unique name and several variables detoned by `Pattern` and `Variable`, respectively. After pattern reference, variables can be bound to certain values using the keywords `where` and `is bound to`. Currently, arbitrary strings and integers can be bound to a variable. Finally, the facts need to be constructed after the keyword `select`: *object fact* consists of exactly one variable of the referenced pattern; *attribute fact* consists of exactly one variable and a modeling language dependent name of certain `Attribute`; finally a *reference fact* refers to *two* variables and a modeling language dependent name of certain `Reference`.

```
1 Policy:
2   'policy' ID PermissionLevel 'by default' '{'
3   Rule*
4   '}' 'with' ResolutionType 'resolution';
5 Rule:
6   'rule' ID PermissionLevel 'to' ID (',' ID)* '{'
7   'from query' [Pattern]
8   Binding*
9   'select' Fact
10  '}' ('on' INT 'priority')?;
11 ResolutionType:
12  'restrictive' | 'permissive';
13 Fact:
14  ObjectFact | ReferenceFact | AttributeFact;
15 ObjectFact:
16  'obj' '(' [Variable] ')';
17 ReferenceFact:
```

```
18   'ref' '(' [Variable] '->' [Variable] ':' Reference ')';
19 AttributeFact:
20   'attr' '(' [Variable] ':' [Attribute] ')';
21 Binding:
22   'where' [Variable] 'is bound to' Value;
23 Value:
24   STRING | INT;
25 PermissionLevel:
26   'allow' Operation | 'obfuscate' | 'dangle' | 'deny' Operation;
27 Operation:
28   'R' | 'W' | 'RW';
```

Listing 3.1: Syntax of Access Control Language

Example 8 Listing 3.2 describes the access control rules for the *PumpControlEngineer* which uses the graph queries defined in Listing 3.3.

Queries. Query `allSignals` selects all signals in the model; query `controlWithType` returns all control units with their type; query `providedSignal` collects pairs of signal and type of controls providing the signal; query `protectedComposite` selects all protected composites; and query `protectedModule` lists the submodules of protected composites.

Policy. Initially, all assets are denied by default. Rule `readSignal` grants read permissions for all signals selected by the query `allSignal`. Rule `permitControl` grants read and write permissions for control units selected by the query `controlWithType` where the type is bound to `::PumpControl`. Similarly, rule `permitSignals` grants read and write access to signals selected by the query `providedSignal` where the type is bound to `::PumpControl`. Finally, rules `deny ProtectedVendor` and `denyProtectedModule` deny the readability of modules and vendor attributes selected by queries `protectedModule` and `protectedComposite`, respectively. When two rules are in conflict, *restrictive* resolution will be used.

```
1 policy WTPolicy deny RW by default {
2   //Grant R for signals
3   rule readSignal allow R to PumpControlEngineer {
4     from query "allSignal"
5     select obj (signal)
6   } on priority 1
7   //Grant RW for pump control units
8   rule permitControl allow RW to PumpControlEngineer {
9     from query "controlWithType"
10    where type is bound to ::PumpControl
11    select obj (control)
12  } on priority 2
13  //Grant RW for signals provided by pump control units
14  rule permitSignal allow RW to PumpControlEngineer {
15    from query "providedSignal"
16    where type is bound to ::PumpControl
17    select obj (signal)
18  } on priority 2
19  //Deny RW of protected vendors
20  rule denyProtectedVendor deny RW to restrictedGroup {
21    from query "protectedComposite"
22    select attr (module : vendor)
23  } on priority 3
24  //Deny R of control units contained by protected composites
25  rule denyProtectedModules deny R to restrictedGroup {
26    from query "protectedModule"
27    select obj (module)
28  } on priority 3
29 } with restrictive resolution
```

Listing 3.2: Rules specific to user *PumpControlEngineer*

```
1 //Queries all signals in the model
2 pattern allSignal(signal : Signal) {
3   Signal(signal);
4 }
5 //Queries pairs of control and its type
6 pattern controlWithType(control : Control, type) {
7   Control.type(control, type);
8 }
9 //Queries pairs of signal and type of
10 //control that provides the signal
11 pattern providedSignal(signal : Signal, type) {
12   Control.provides(control, signal);
13   Control.type(control, type);
14 }
15 //Queries protected composites
16 pattern protectedComposite(composite : Composite) {
17   Composite.protectedIP(composite, true);
18 }
19 //Queries all modules contained by protected composite
20 pattern protectedModule(module : Module) {
21   Composite.submodules(composite, module);
22   Composite.protectedIP(composite, true);
23 }
```

Listing 3.3: Graph patterns for access control rules

3.6 Derivation of Effective Permissions

In this section, we recap (in heavily condensed form) our most important results [*c*12; *c*8] regarding rule-based permissions and conflict resolution. In Section 3.6.1, we classify the conflicts based on the types of assets involved, permission levels and operation types. Then Section 3.6.2 introduces the basic steps for reasoning on permissions and conflicts, and formulate the conflict resolution problem using these notions. Then in Section 3.6.3.1, Section 3.6.3.2 and Section 3.6.3.3, we introduce solutions to this problem satisfying goal **G2**, in the form of a from-scratch algorithm, an incremental computation technique and their combinations.

3.6.1 Analysis of Conflicts

Several rules can state contradict permission levels for the same assets. Moreover, the read and write operations of an asset may depend on other assets by consistency rules. These conflicts have to be resolved in order to obtain the conflict-free *effective permissions* which are actually enforced. We classified the conflict types into 3 category represented in Table 3.2 based on the asset on which the rules are applied and the operation (*read,write*) they want to grant or deny.

Type I. A conflict of *Type I.* appears when two or more rules apply to the same asset and specify different permission levels for the same operation.

Type II. A conflict of *Type II.* appears when two or more rules apply to the same asset but on a different operation.

Type III. A conflict of *Type III.* appears when two or more rules apply to different assets that depend on each other (see section 2.2).

Table 3.2: Conflict Categories

	Asset	Operation
Type I.	same	same
Type II.	same	different
Type III	different	-

3.6.1.1 Type I. Same Asset and Operation

An asset cannot both be readable and hidden; neither can it be both writeable and unmodifiable at the same time. Naturally, when two or more different permission levels are applied to the same operation type (read/write) of the same asset, a conflict occurs.

3.6.1.2 Type II. Same Asset, Different Operation

In case of different operations, we have to discuss the conflicting combinations of read and write permissions. As Table 3.3 shows, most of the cases are valid. On the other hand, if an asset is writable then it has to be readable while an unreadable asset cannot be writable. Moreover, giving write permission to an obfuscated asset is questionable. It should mean that the users are only allowed to set obfuscated values which is not common in practice. Our approach works well regardless whether this combination is allowed or not.

Table 3.3: Compatibility matrix of access and operations types

		Read		
Write		Deny	Obfuscate	Allow
	Deny	✓	✓	✓
	Allow	✗	?	✓

3.6.1.3 Type III. Different Assets

An *object* asset can be in conflict with its *attributes* and *references*.

Read Dependency Readability of an object can be in conflict with one of its attributes when the attribute is readable/obfuscated but the object is unreadable.

In case of a reference, its source and target objects have to be taken into account. When a reference has to be readable, both of its source and target objects have to be readable too. Conversely, if either of the source and target object are unreadable references between them have to be invisible.

When an object asset o is readable its parent and the container reference have to be visible as well. This means that all the objects have to be visible in the containment hierarchy until o. Conversely, when a containment reference asset is invisible, none of its children should be visible.

Finally, in case of a readable object, all of its features are also readable by default. However, this is merely a weak consequence: rules can specifically override this default for each feature.

Write Dependency Write permission means that the asset can be modified. In case of a writable/unwritable object, all/none of its features are writable by default. Once again, rules can specifically override this default for each feature.

When an object asset is writable it does not mean that it can be deleted. For that behavior, the containment reference that holds the object must also be modifiable. When a containment reference is writable, all of its children can be deleted (if they themselves are writeable) or moved to under another containment reference (if the other containment reference is also writable).

Multiplicity Constraints The presented conflict resolutions handle the following internal consistency constraints: *object existence*, *containment hierarchy*, *opposite features* described in Section 2.2. Now, we investigate the *multiplicity constraints*. Multiplicity of a feature type defines the number of possible target values or objects and consists of a lower and upper limit ($[lower]..[upper]$). However, not all type of multiplicity constraints are supported by our approach.

[**0..***] If there are no multiplicity restrictions, the presented approach works well without further considerations.

[**0..m**] This type of multiplicity constraint defines an unsettable feature (attribute or reference) may have up to m values per object, where m is a finite integer (typically 1). Applying read permissions may only decrease the number of model facts in the filtered model, thus this multiplicity constraint is met in the secure view regardless whether the value is readable, hidden or obfuscated. However, for a given object with one or more feature values that are invisible to the user (the security rules deny the read permission to the asset formed by the specific feature value), write access control must prevent the user from adding new values to the feature (even if the security rules would allow write permissions for model fact formed by the new value) if, together with the hidden values, they would exceed the upper limit in the unfiltered model.

[**1..1**] This type of multiplicity constraint defines a feature where a value must always be set. If the user is not allowed to read the associated model fact, then an obfuscated (and unmodifiable) value must be provided so that the secure view is a consistent model. For instance, the identifier of a visible object cannot be hidden, only obfuscated.

[**k..m**] This rare type of multiplicity constraint defines a feature with at least $k \geq 1$ and at most $m > 1$ values per object, where m can be unlimited. It implies that at least k distinct values need to be visible to the user, but if fewer than that number are readable, there is no self-evident deterministic way for selecting a subset of hidden values for obfuscation. Therefore our approach is only applicable if no such feature values are hidden from the user when the host object itself is visible.

3.6.2 Reasoning on Conflicts

3.6.2.1 Judgments

During the process of conflict resolution, our approach maintains a set of (typically conflicting) direct and indirect consequences of the access control rules. When dealing with such consequences, we treat lower bounds (e.g. "this object must be visible, at least in an obfuscated form") and upper bounds (e.g. "this object must not be completely readable, it may be obfuscated at best") on a permission level as separate *judgments*, even if both bounds coincide (e.g. "this object must be obfuscated").

Definition 3.2 (Judgement). Each judgment j is a tuple $j = \langle a, o, p, \psi, \pi \rangle$, where

- $a \in$ Assets is an asset,
- $o \in \{R, W\}$ is an operation (read or write) to be performed on the asset,
- $p \in \text{Levels}_o$ is a permission level associated with the operation,

Table 3.4: Initial judgments of rules `denyProtectedModules` `permitControl` and global defaults of `o7`

a Subset of Initial Judgments

$\langle o(\texttt{o18,Control}), R, deny, <, 3\rangle$
$\langle o(\texttt{o21,Control}), R, deny, <, 3\rangle$
$\langle o(\texttt{o7,Control}), R, allow, >, 2\rangle$
$\langle o(\texttt{o18,Control}), R, allow, >, 2\rangle$

b Global Defaults of `o18`

$\langle o(\texttt{o18,Control}), R, deny, <, 0\rangle$
$\langle o(\texttt{o18,Control}), R, deny, >, 0\rangle$
$\langle o(\texttt{o18,Control}), W, deny, <, 0\rangle$
$\langle o(\texttt{o18,Control}), W, deny, >, 0\rangle$

- $\psi \in \Psi = \{>, <\}$ indicates whether the judgment puts that permission level as an upper respectively lower bound for the final permission decision, and
- $\pi \in \Pi$ is a priority class. ▪

Definition 3.3 (Permission Set). We define a *permission set* as a set of judgments, formally

$$\textit{permission set} = \langle j_0, j_1 ... j_n \rangle$$

A valid permission set is *complete*, i.e. for each asset and operation, it must contain at least one upper bound and one lower bound judgment, and by the ordering of permission levels, the lowest (strictest) upper bound must not be higher than the highest (strictest) lower bound. ▪

The *initial permission set* is obtained from rules and defaults. Default judgments have a priority class that is lower than the priority of any query-based rule; they are included as a pair of judgments (one upper bound, one lower bound, both with the same permission level), which ensures completeness. For each match of the queries of security rules that apply for the user, an upper or lower bound judgment is added; the asset is given by the pattern match, and the operation, priority class and permission level are determined by the rule header.

Example 9 The *initial permission set* obtained from the evaluation of access control rules `permitCompositeModules` and `deny ProtectedSignals` is collected in 3.4a. Global defaults deny all access with 4 judgments per asset (both bounds ($>, <$) and operations (R,W) set to $deny$), e.g. the default judgments of the control module `o10` are listed in 3.4b.

We can restate goal **G2** as to derive the *resolved permission set*, where (a) the highest (most permissive) lower bound is equal to the lowest upper bound for each asset and operation (thereby identifying a single permission level as the *effective permission*), and (b) there are no conflicts (neither directly nor indirectly though dependencies) between the judgments. We discuss this in the sequel.

3.6.2.2 Consequence Propagation

In Section 3.4.4 we discussed how the requirement of internal consistency introduces dependencies between the permission levels of different assets and operations.

As a judgment may impose dependency constraints on other assets or operations, its consequences can be propagated and directly represented as additional judgments on foreign asset/operation pairs. Using the *propagated consequence* judgments, all indirect conflicts can be expressed as *local conflicts*, i.e. two directly contradicting judgments (an upper bound one, lower than a lower bound one) for the same asset and operation. It was discussed in [$c12$] how consequence propagation makes it sufficient

to focus on local conflicts, as opposed to dependency-driven indirect conflicts between judgments at different assets or operations.

Definition 3.4 (Strong Consequence). For judgment $j = \langle a, o, p, \psi, \pi \rangle$ with a dependency on asset and operation $\langle a', o' \rangle$, the propagated *strong consequence* judgment is denoted as

$$j_{\langle a', o' \rangle} := \langle a', o', l, \psi, \pi \rangle$$

where l is the ψ-bound associated with p by the Galois connection described in Section 3.4.4. ▪

Extending the above notion, is also possible to propagate *weak consequences* that encapsulate a default effect of a given judgment, not a necessary condition.

Definition 3.5 (Weak Consequence). Unlike its strong counterpart, a weak consequence does not inherit the priority of the original judgment, but is assigned a lower priority instead, and may be overridden by more dominant judgments without conflicting with the original judgment. It can be formally captured as

$$j^*_{\langle a', o' \rangle} := \langle a', o', l^*, \psi, \pi^* \rangle$$ ▪

It was proposed in [c12] to assign a priority class to these weak consequences that is lower than the priority of any user-specified rule, but higher than the priority of global defaults that such a weak consequence may override.

Example 10 The judgment $\langle o(\texttt{o7,Control}), R, deny, <, 3 \rangle$ obtained from rule `denyProtectedModule` needs to be extended with $\langle o(\texttt{o7,Control}), W, deny, <, 3 \rangle$ as *weak consequence*. In addition, judgment $\langle o(\texttt{o18,Control}), R, allow, >, 2 \rangle$ obtained from rule `permitControl` needs to be propagated to the container object `o13` as *strong dependency*. Hence, new judgment appears with exactly the same priority, namely the judgment $\langle o(\texttt{o13,Composite}), R, obfuscate, >, 2 \rangle$.

3.6.2.3 Dominance and Resolution

Take any two judgments that are in conflict. Without loss of generality, we may assume that one must *dominate* the other via its higher priority class (see [c12] for the unimportant nuance of conflicts within the same priority class). Due to the consequence propagation introduced in Section 3.6.2.2, it is enough to consider local conflicts, as any conflict will manifest itself somewhere as a local conflict. Therefore we consider pairs of judgments on the same asset and operation, where one is an upper bound judgment, and the other is an incompatible lower bound judgment with different priority.

Definition 3.6 (Resolution Step). A *resolution step* takes two judgments that are in a local conflict, and *relaxes* the dominated judgment to make it compatible with the dominant judgment. For locally conflicting judgments $j = \langle a, o, p, \psi, \pi \rangle$ and $j' = \langle a, o, p', \psi', \pi' \rangle$ with $\pi > \pi'$ the conflict resolution replaces j' with $j'' = \langle a, o, p, \psi', \pi' \rangle$. Executing such a step transforms a permission set to a different one. ▪

Observations: (a) j'' relaxes j', i.e. upper bounds are raised when replaced, while lower bounds are lowered. (b) j'' is guaranteed by the construction to be non-conflicting with j. (c) The resolution step upholds the completeness of the permission set.

Example 11 In Example 9, multiple local conflicts exist. For instance, judgment $\langle o(\texttt{o18,Control}), R, allow, >, 2\rangle$ is in conflict with $\langle o(\texttt{o18,Control}), R, deny, <, 3\rangle$ Due to domination, the judgment $\langle o(\texttt{o18,Control}), R, deny, <, 3\rangle$ with higher priority relaxes the other to $\langle o(\texttt{o18,Control}), R, deny, >, 2\rangle$.

3.6.3 Conflict Resolution Approaches

To satisfy goal **G2**, we developed *batch* and *incremental* conflict resolution strategies. As an improvement, we also derived combinations to use the strengths of *batch* and *incremental* approaches.

3.6.3.1 Batch Conflict Resolution

The algorithm Algorithm 1 proposed in [c12] iterates over judgments in the order of dominance (i.e. highest priority first), to propagate their consequences (also the weak consequences, unless contradicted by a previously processed judgment) and resolve any local conflicts that are detected.

Algorithm 1 Batch resolution process in pseudocode

function GetEffectivePermissions(model, user, policy)
  $permissionSet \leftarrow initialPermissions(model, user, policy)$
  $processed \leftarrow \emptyset$
  while $permissionSet \not\subseteq processed$ **do**
    $j \leftarrow chooseDominant(permissionSet \setminus processed)$
    $processed \leftarrow processed \cup \{j\}$ ▷ mark as effective
    for all dependencies ⟨a',o'⟩ of j **do**
      $conseq \leftarrow \{j_{\langle a',o'\rangle}\}$ ▷ strong consequence
      if $\nexists\, j' \in processed$ conflicting $j^*_{\langle a',o'\rangle}$ **then**
        $conseq \leftarrow conseq \cup \{j^*_{\langle a',o'\rangle}\}$ ▷ weak
      end if
      $permissionSet \leftarrow permissionSet \cup conseq$
    end for
    $conflicts \leftarrow localConflictsOf(j, permissionSet)$
    for all $j' \in conflicts$ **do** ▷ resolve locally
      $j'' \leftarrow ResolutionStep(j, j')$ ▷ j dominates
      $permissionSet \leftarrow permissionSet \cup \{j''\} \setminus \{j'\}$
    end for
  end while
  return $permissionSet$
end function

We have the following success criteria against the resolution process:

- **Termination** the process must eventually end.
- **Correctness** the process must yield a resolved permission set upon termination.
- **Confluence** in order to work predictably and deterministically, the effective permissions must be completely determined by the user in question, the model and of course the policy.

The presented algorithm is terminating, as (for given assets) there is a finite space of possible judgments, each of which is processed at most once.

Once a judgment j has been processed, it will not be removed from the permission set anymore, as all judgments that could potentially dominate it have already been processed, and neither propagation nor resolution would create such dominating judgments anymore. It will not enter into new conflicts as the dominating party either, since all of its consequences have been propagated, so any judgment that would propagate a strong consequence that would be dominated by it, would never be processed, but would have previously been removed in a resolution step. Note that weak consequences are not propagated if they would create a conflict with an already processed judgement.

When a judgment is processed, its local (type I) conflicts are resolved; as it cannot enter new conflicts, there will be no more type I conflicts when the process terminates. This also means that there are no more conflicts of any type, since all strong consequences have been propagated. All the while the completeness of the permission set is maintained, thus the end result is a resolved permission set, and the process is correct.

Note that when processing a judgment, the associated propagations and resolutions yield deterministic results. Thus confluence requires that the end result is the same regardless of the order that judgments are selected for processing in `chooseDominant(.)`. The process has free choice only between judgments that do not dominate each other; in this case they are of the same priority class and bound, so the end result will contain all of their transitive consequences (with no domination between them), and conflicts will be confluently resolved to be compatible with the conjunction of all these judgments, independently of the order of steps.

Example 12 The output of the algorithm on the case study is presented in Example 7 and visualized in Figure 3.1. Some key elements of the example:

- Initially, the $permissionSet$ consists of judgements denying the read and write permission of all facts by default. In addition, judgments enabling the read permissions of all signals are added due to the rule `readSignal`; enabling the read and write permissions of `o7` due to the rule `permitControl`; enabling the read and write permissions of `o8` and `o9` due to the rule `permitSignal`; disabling the read and write permissions of `vendor` attribute of `o13` due to the rule `protectedComposite` and disabling the read permission of `o21` and `o18` due to the rule `protectedModule`.
- The effective permission set will disable the read and write permission of signals `o22`, `o23`, `o19` and `o20` due to the relaxation of the read permission of container object namely `o21` and `o18`. Similarly, `o1`, `o2` and `o13` are visible due the relation of the visibility of `o7` and `o24`.
- The `consume` reference between `o7` and `o3` is editable as a relaxation of the modifiability of `o7`. It means the reference can be deleted but the `o3` cannot be removed from the model.

3.6.3.2 Incremental Conflict Resolution

The above described algorithmic conflict resolution process was reformulated [*c*8] into a declarative computation using the language of recursive graph queries (Datalog [Gre+13]), specifically the VIA-

TRA [Ujh+15; Var+16] syntax. The motivation is to take advantage of incremental query evaluation algorithms [GMS93] that are available for queries thus formulated.

The declarative conflict resolver takes the following inputs (extensional relations in Datalog parlance):

- The explicit judgments that are immediately obtained by evaluating the access control rules in the policy; as mentioned above, each match of the associated model query of the rule will yield a judgment (in the initial permission set).
- Low-priority judgments corresponding to global defaults (can be merged with the former).
- Relations describing the connections between assets, in order to enable the propagation of consequent judgments. For instance, in case of object facts, a binary relation describing the containment hierarchy is necessary.

From these extensional relations, the resolved permission set is derived (in multiple stages) in the form of incrementally maintained queries (intensional relations). While it is possible to build such a solution with a fixed number of queries, there are multiple arguments [*c*8] in favor of a more sophisticated query structure that depends on the number of distinct priority classes in the policy. These queries themselves are automatically generated from the policy in a way that intensional relations representing judgments of a given priority may derive from relations representing judgments of higher or equal priority.

For priority class π, we generate the following graph queries (see also Listing 3.4):

`judgment`$_\pi$ matches all inferred judgments $j = \langle a, o, p, \psi, \pi \rangle$ of priority class π, computed as the union of explicit judgments in class π, the strong and weak consequent judgments in class π, and the relaxed forms of dominated judgments in class π. From these, explicit judgments are inputs from extensional relations, while the other cases are computed by the helper queries below.

`relaxedJudgment`$_\pi$ identifies judgments of priority class π that are dominated (see next query) by judgments of any priority $\pi' > \pi$, and computes their relaxed form (inheriting the bound of the dominant judgment).

`dominationBy`$_\pi$ will match (independently of the existence of any dominated judgments) all effective judgments $j = \langle a, o, p, \psi, \pi \rangle \in$ `effectiveJudgment`$_\pi$ of priority π and all potential bounds p' for the corresponding asset and operation that would be dominated by it in local conflict ($p' \psi p$ for $\psi \in \Psi = \{>, <\}$).

`effectiveJudgment`$_\pi$ filters `judgment`$_\pi$ to those judgments that are not dominated, i.e. there is no corresponding match of `dominationBy`$_{\pi'}$ for any $\pi' > \pi$.

`strongConsequence`$_\pi$ computes the propagated consequences of effective judgments at class π, using the extensional relations that describe the dependencies between assets.

`weakConsequence`$_\pi$ analogously.

In the above queries, the priority class comparisons "$\pi' > \pi$" are statically evaluated, and the generated disjunctions only include the clauses where the result is true.

The analysis in [*c*8] shows that this (recursive) query structure has well-defined semantics that qualifies for incremental evaluation as it is *stratified* [Gre+13]. Furthermore, it was proven [*c*8] to yield results equivalent to the non-incremental algorithm.

```
1 pattern relaxedJudgement_at_0(user,asset,op,bound,dir){
2   find judgement_at_0(user,asset,op,dBound,dir);
3   find domination_by_1(user,asset,op,_dBound,bound);
4 } or {
5   find judgement_at_0(user,asset,op,dBound,dir);
6   find domination_by_2(user,asset,op,_dBound,bound);
```

Algorithm 2 Improved resolution process with cached queries

▷ The *engine* is created as a global parameter
engine←CreateIncrementalQueryEngine(*model,policy*)
function GetEffectivePermissions(*model, user, policy, engine*)
    permissionSet← initialPermissions(*model, user, policy, engine*)
    ▷ Rest of the algorithm remains the same
    ...
    return *permissionSet*
end function
▷ The *Users* variable is assumed as an global parameter describing users in the collaboration session
function GetAllEffectivePermissions(*model, policy*)
    results← ∅
    for all u in *Users* **do**
        pSet ← GetEffectivePermissions(*model, u, policy, engine*)
        results ∪⟨u, *pSet*⟩
    end for
    return *results*
end function

```
 7 } or {...}
 8 pattern effectiveJudgement_at_0(user,asset,op,bound,dir){
 9   find judgement_at_0(user,asset,op,bound,dir);
10   neg find domination_by_1(user,asset,op,bound,_);
11   neg find domination_by_2(user,asset,op,bound,_);
12   // ...
13 }
14 pattern domination_by_1(user,asset,op,dBound,dir){
15   find effectiveJudgment_at_1(user,asset,op,eBound,dir);
16   find permissionOutOfBound(dir,eBound,dBound);
17 } // ...
```

Listing 3.4: Generated graph queries in VIATRA syntax (extracts)

3.6.3.3 Combination of Conflict Resolution Methods

Caching Explicit Judgments. In case of the batch method, evaluation of the access control rules to obtain explicit judgment includes the evaluation of set of graph patterns (specified by the rules). To recalculate the permission set after model changes, these queries need to be reevaluated from scratch which can cause noticeable execution time overhead. Similar time overhead can occur, when batch is executed for multiple collaborators where queries cover similar parts of the model.

The incremental approach solves this problem by incrementally maintaining the result set of queries after introducing model changes. Applying the incremental maintenance to result set of explicit judgments related queries using an incremental graph query engine (such as VIATRA Query[Var+16]) can improve execution time in case of recalculation and in case of batched execution for multiple collaborators.

Algorithm 2 presents the improved resolution process with cached queries. It creates a global incremental query engine on the instance model which immediately evaluates the queries of the policy. The same query engine will be used (i) for all users in the session and (ii) if recalculation is triggered. From now, parameter list of function initialPermissions is extended with the $engine$ parameter to obtain query results immediately. In addition, function GetAllEffectivePermissions is introduced to demonstrate how the approach can be invoked in case of multiple users in a collaborative session.

Algorithm 3 Combining the usage of batch and incremental approaches

```
▷ The engine is created as a global parameter
engine←CreateIncrementalQueryEngine(model,policy)
function GetAllEffectivePermissions(model, policy)
    results← ∅
    ▷ Users and IncrementalUsers variables are global parameters
    for all u in Users do
        if u ∈ IncrementalUsers then
            pSet ← GetEffectivePermissions^INCR(model,u,policy,engine)
            results ∪⟨u, pSet⟩
        else
            pSet ← GetEffectivePermissions(model, u, policy)
            results ∪⟨u, pSet⟩
        end if
    end for
end function
```

Combined Usage of Both Methods. During the collaboration, different collaborators may benefit from at conflict resolution approach that is capable of immediate recalculation or handle long transactions of model modifications. For this purpose, instantiating different number of batch and incremental approaches on the same rule set and instance model could be a viable option.

Algorithm algorithm 3 presents the approach where *Users* variable stores all the users participating in the collaborative session while *IncrementalUsers* contains collaborators requesting the incremental approach. At first a global query engine is created that will be used by the incremental approach. In this case, function GetAllEffectivePermissions iterates over all users in the session. If a user requested incremental evaluation, function GetEffectivePermissions^{incr} is called with the query *engine* on the parameter list, otherwise the original function GetEffectivePermissions is called. Without detailed description of function GetEffectivePermissions^{incr}, it yields the result set of generated graph patterns related to effective judgments on the highest priority level (see subsubsection 3.6.3.2).

3.7 Evaluation

In order to fulfill goal **G3**, we have carried out a scalability evaluation for all approaches in the context of *Wind Turbine* case study[Bag+14] of the MONDO FP7 project and compare the results of the proposed approaches. We address the following research questions in the evaluation:

- **Q1** How scalable are the different permission evaluation methods wrt. increasing *model size*?
- **Q2** How scalable are the different permission evaluation methods wrt. increasing number of *collaborators*?
- **Q3** How scalable are the different permission evaluation methods wrt. increasing *model change size*?

3.7.1 Measurement Setup

For the measurement, we used the metamodel of Figure 2.1 with a slight modification: the control unit attribute `type` was changed from an enumeration to a string, with K different permitted values. The corresponding policy file is similar to the extract shown by Listing 3.2, with one specialist engineer for

each control unit type (each having two access control rules dedicated to them). This means altogether K users and $2K + 2$ access control rules.

Measurements were performed with instance models of various sizes. The model of size M contains a root `Composite` object with 1 object asset, 2 attribute assets. It contains M copies of the structure depicted by Figure 3.1. This implies M additional `submodule` reference assets, $3M$ composite modules with $2 \times 3M$ attribute assets and $2M$ `submodule` reference assets; $4M$ control units with $2 \times 4M$ attribute assets and another $4M$ `submodule` reference assets; $16M$ signals with $16M$ `provide` reference assets; as well as $8M$ consumes cross-reference assets between modules and signals. Altogether, the read and write perimssions of $1 + 23M$ object assets, $2 + 14M$ attribute assets and $31M$ reference assets need to be derived which means $6 + 138M$ effective judgments.

The copies are not completely identical: the `vendor` attributes are set to a different value in each copy; and the `protectedIP` attribute of composites as well as the `type` attribute of control units were chosen randomly from their respective ranges with uniform distribution. However, care was taken that all control unit types must occur at least once; this also implies $4M \geq K$.

To evaluate the recalculation of effective permissions, generic operations are defined including addition of objects and references as well as set of attributes.

Attribute Set changes the `protectedIP` of an arbitrary composite.
Object Creation creates a signal under the root.
Reference Creation creates a `consume` reference between arbitrary control and signal.

During the measurement, each operation is introduced Ch times on the model. This behavior covers usual changes a collaborator would do during a collaboration session. On the other hand, varying the size of Ch emulates various size of commits that collaborators submit during a real project.

3.7.2 Scenario

The measurement scenario consists of the following steps:

Step 1 Load the instance model and access control rules.
Step 2 Calculate the effective permissions.
Step 3 Measure execution time of calculation.
Step 4 Apply the change operations.
Step 5 Measure execution time of application.
Step 6 Recalculate the effective permissions.
Step 7 Measure exeuction time of recalculation.

Note that *Step 4* is required to measure the propagation of incremental approach while *Step 6* is irrelevant in that case.

Hardware Configuration All the measurements[1] executed on a personal computer (CPU: Intel Core i3-3217U@1.80GHz, MEM: 8GB) with maximum a 7GB of Java heap size.

3.7.3 Results

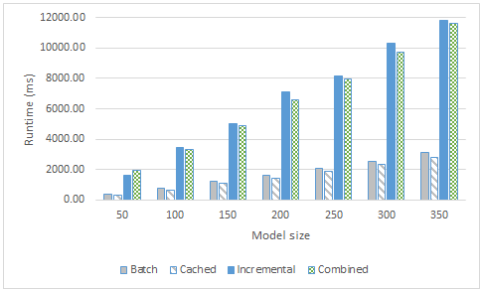
All measurements were executed 10 times and the median of their results was recorded on diagrams. Each diagram visualizes the table above it and uses two types colors for each method symbolizing the

[1] Raw data and reproduction instructions at https://github.com/debrecenics/effective-permission-evaluation

Table 3.5: Average execution time to derive effective permissions wrt. increasing model size.

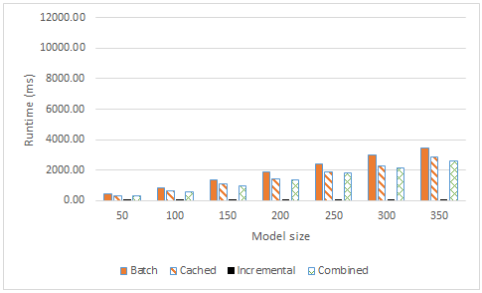
a Evaluation on original models ($U = 10$)

M	*Batch (ms)*	*Cached (ms)*	*Increm. (ms)*	*Comb. (ms)*
50	353.97	285.47	1606.96	1960.34
100	751.52	634.79	3467.04	3327.72
150	1255.85	1108.86	5026.71	4891.47
200	1595.29	1429.65	7131.55	6583.99
250	2060.48	1880.82	8128.19	7980.75
300	2505.67	2322.81	10328.72	9723.54
350	3099.79	2810.95	11782.69	11627.99



b Reevaluation on modified models ($U = 10, Ch = 30$)

M	*Batch (ms)*	*Cached (ms)*	*Increm. (ms)*	*Comb. (ms)*
50	364.54	290.55	47.46	311.56
100	741.01	625.81	42.49	600.91
150	1220.56	1112.56	44.98	974.48
200	1597.67	1408.67	43.85	1376.43
250	2058.48	1882.90	41.70	1785.92
300	2513.51	2305.59	41.88	2150.71
350	3262.51	2877.11	43.75	2595.01



first time evaluations and the reevaluations after certain changes are introduced. In case of combined version, $50\% - 50\%$ of the users used batch and incremental versions.

3.7.3.1 Increasing model size

To address **Q1**, Table 3.5 depicts the methods with increasing model sizes form $M = 50$ ($1,151$ objects, 702 attributes, $1,550$ references) to $M = 350$ ($8,051$ objects, $4,902$ attributes, $10,850$ references) whereas fixed number of active users $U = 10$ and size of changes $Ch = 30$ were used.

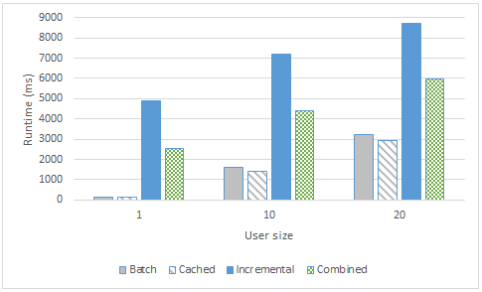
On the original model, it is shown that each approach is proportional to model size. Batch and cached methods are much faster (76% in average) than incremental and combined version. Moreover, cached approach has slightly better performance (1.94% in average) than batch. Combined method is between batch and incremental approaches, but it is clearly visible that the incremental part of the combined has larger impact on the execution time as its results are closer to the measured values of incremental method.

After introducing changes, batch and cached methods need almost the same time ($\pm 30ms$) for the recalculation as it was required for the first time. It means the change size is negligible in comparison of the model sizes and these approaches starts the recalculation from scratch. Incremental method needs a constant time ($40.50ms$ in average) which means that the approach is independent from the model size, hence the incrementality is proved. In case of the recalculation, combined version is mostly influenced by the batch method resulting in similar curve on the diagram. However it is still faster (16.51% in average) than batch version.

Table 3.6: Average execution time to derive effective permissions wrt. increasing user size

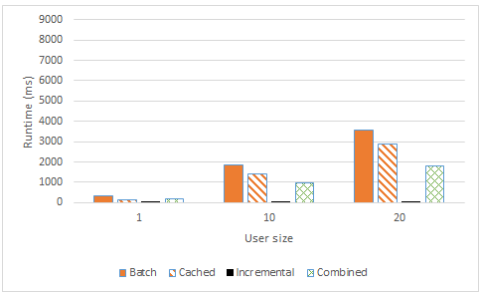
a Evaluation on original model ($M = 200$)

U	*Batch (ms)*	*Cached (ms)*	*Increm. (ms)*	*Comb. (ms)*
1	157.04	141.17	4897.63	2527.34
10	1589.45	1430.12	7187.78	4388.62
20	3242.24	2925.56	8700.63	5971.44



b Reevaluation on modified model ($M = 200, Ch = 30$)

U	*Batch (ms)*	*Cached (ms)*	*Increm. (ms)*	*Comb. (ms)*
1	158.15	140.58	16.77	87.47
10	1597.67	1408.66	43.85	821.85
20	3215.97	2857.87	78.62	1647.30



3.7.3.2 Increasing user size

To address **Q2**, Table 3.6 represents results of the scalability evaluation wrt. increasing user size from $U = 1$ to $U = 20$ whereas the model size is fixed $M = 200$ ($4,601$ objects, $2,802$ attributes, $6,200$ references). We believe that 20 users cover the largest typical development team working on the project.

For the first execution, it is clearly visible that all approaches are proportional to user size. In case of one user, all methods require negligible execution. Batch and cached versions are faster (77.89% and 80.10% in average, respectively) than the incremental one in all cases while combined version is between batch and incremental methods mostly influenced by its incremental part.

On the modified model, approaches are still proportional to user size. In case of batch and cached versions, results are unchanged as the execution time is influenced mostly by the model size while the change size is negligible. Incremental version is much faster ($\%95.3$ in average) than batch and cached versions. Again, combined version is between batch and incremental methods, but this time its batch part has the most impact.

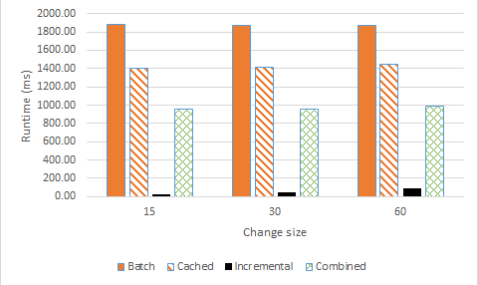
3.7.3.3 Increasing change size

To address **Q3**, Table 3.7 shows results of our evaluation wrt. increasing change size from $Ch = 15$ to $Ch = 60$ whereas the model size ($M = 200$) and user size ($U = 10$) are fixed. We believe that 60 changes on models cover the largest typical size of user modifications in a commit (thus we exclude large changes derived by automated model transformations, for instance).

The results show that incremental approach is proportional to size of change. In case of the other approaches, the changes are negligible in comparison to the model size hence there are only small

Table 3.7: Average execution time to derive effective permissions wrt. increasing change size ($M = 200, U = 10$)

Ch	*Batch (ms)*	*Cached (ms)*	*Increm. (ms)*	*Comb. (ms)*
15	1595.89	1398.06	24.30	810.10
30	1589.46	1408.67	43.85	817.74
60	1611.17	1443.92	92.77	851.97



differences ($\pm 50ms$) between the execution times. However, it is also visible that increasing change size implies longer execution time.

3.7.4 Discussion and Conclusion

Small maintenance computation. According to the measurement results, the incremental version fulfills the goal **G4**. Table 3.7 shows that incremental version is independent from the original model size, and Table 3.7 displays that only the size of introduced changes impacts the execution time.

Online Collaboration Scenario. As it is shown, incremental conflict resolution method provides almost immediate recalculation of effective permissions upon model changes independent from the model size up to 20 collaborators. This approach fits for online scenarios where fast response time is required during collaboration. However, the initialization takes more time in case of increasing model sizes or number of collaborators.

Offline Collaboration Scenario. Batch and cached conflict resolutions take less time than the incremental method for the first time but its performance is poor in case of recalculation. These approaches fit for offline scenarios where there is no permanent connection to the server during the collaboration and the resolution method can be invoked on demand such as upon a commit.

Batch vs. Cached. Results show that caching explicit judgments provides better performance than batch in all aspects. Hence, we can state reusing information from previously evaluated queries improves the execution time of the derivation of effective permissions.

Combination of approaches. Combined approach where collaborators can choose the collaboration scenario that most fits for them is a good trade-off between the online and offline collaboration.

3.8 Contributions

In this chapter, we have summarized the formal foundations for describing the interpretation of rule-based access control policies in MDE. A textual syntax is defined for the proposed security policy language that includes defaults, query-based rules and also the conflict resolution options. A framework

of conflict resolution processes is proposed that is guaranteed to be deterministic and conflict-free including *batch* and *incremental* approaches and their combinations. We have compared these conflict resolution approaches using an experimental scalability evaluation.

Contribution 1 I proposed a domain-customizable modeling language to capture fine-grained access control policies and I realized a framework to efficiently evaluate the policies in online and offline scenarios. [*j*1], [*j*2], [*c*7], [*c*8], [*c*12],

- **1.1 Access Control Language.** I proposed a rule-based access control language to describe high-level and fine-grained policies in both online and offline scenarios. Rules may allow, obfuscate or deny read and/or write permissions of model parts identified by graph patterns. [*j*1], [*c*7], [*c*8], [*c*12],
- **1.2 Read and Write Dependencies.** I analyzed read and write dependencies implied by high-level access control policies as read and write permissions of a model part may depend on other model parts implied by internal consistency rules. [*c*12]
- **1.3 Deriving Effective Permissions.** I implemented a prototype framework to derive a set of effective permissions from access control policies in the context of models providing batch and incremental evaluation to support offline and online collaboration, respectively. [*c*8], [*c*12]
- **1.4 Evaluation.** I evaluated the scalability of the proposed prototype framework on a case study of offshore wind turbine controllers. [*c*8], [*c*12]

The algorithm and its formalization to derive effective permission based on the proposed language is the contribution of Gábor Bergmann whereas my contributions are the proposed language, dependency analysis, implementation of the algorithm and its evaluation. István Ráth introduced fine-grained access control as an extra protection layer for modeling tools used in collaborative modeling on top of traditional version control systems.

Chapter 4

General Secure Collaboration Scheme

4.1 Introduction

This chapter is based upon [*j*2] and addresses the challenge of access control in collaborative modeling (**C-I**) which requires to achieve secure collaborative modeling with fine-grained access control relying upon existing storage back-ends to follow current industrial best practices. In the context of collaborative modelling, the term *secure* refers to the fact no confidential information stored on client side, hence only non-confidential data is sent to users to work with.

We aim to address the following high-level goals in this chapter:

G1 *Secure and versatile offline collaboration* where each collaborator can work with a model fragment filtered in accordance with read permissions, and processed using off-the-shelf MDE tools (e.g. editor, verifier). A user may be disconnected from any server or access control mechanism, and then submit (commit) his updated version in the end.

G2 *Secure and efficient online collaboration* where multiple users can view and edit a model hosted on a server repository in real-time while imposing different read and write permissions. Small changes performed by one collaborator are quickly and efficiently propagated to the views visible to other users, without reinterpreting the entire model.

G3 *General collaboration schema* that is adaptable for online and offline scenario defining workflows of a server and multiple clients to handle fine-grained access control management.

Challenges Deriving from the goals stated in Section 4.1, we identify the following challenges.

C1.1 *Model compatibility.* To meet **G1** in off-line collaboration scenario, the approach must be able to present the information available to a given user as a self-contained model, in a format that can be stored, processed, displayed and edited by off-the-shelf modeling tools.

C1.2 *Offline Models.* To meet **G1** in an off-line collaboration scenario, the approach must be able to present only the available information to a given user without maintaining connectivity with any central server or authority responsible for access control.

C2.1 *Incrementality.* To meet **G2** in an on-line collaboration scenario, the approach must be able to process model modifications initiated by a user and apply the consequences to the views available to other users without re-processing the unchanged parts of the model.

C3.1 *Correctness criteria.* To meet **G3**, the approach must define the correctness criteria of the collaboration schema and prove their fulfillment.

C3.2 *Adaptability.* To meet **G3**, the approach must realize the collaboration schema both in offline and online scenarios.

Approach	Asset Selection	Use Model Trans.	Operations
[PGC16]	any language	+	generated rules
[Jür08]	single object asset	-	
[BPA07]	OCL	-	
Our solution	declarative graph query	+	generic elementary rules

Table 4.1: Comparison of related approaches

Structure Rest of the chapter is organized as follows. Related work is overviewed in Section 4.2. In Section 4.3, we overview our bidirectional model transformation for access control, while Section 4.4 describes our secure collaboration schema and proves its correctness. In Section 4.5, we give a brief overview on how to adapt this collaborative modeling schema to online and offline scenarios. Finally, Section 4.6 describes the evaluation of our approach. Section 4.7 concludes the chapter by a summary and state the contributions achieved in Secure Collaboration.

4.2 Related Work

Access control using Model Transformation. Closest to our approach, [PGC16] uses model transformations to build infrastructures that can manage access control policies written in any policy language. It means that their approach takes a policy model as input and derives transformation rules to enforce read and write permissions. To compare it to our solution, we use elementary model transformation rules that take the effective permissions as input, instead of integrating the permissions into our rules.

Model-driven Security. Model-based techniques have also been used for access control purposes. In [Jür08], similarly to our solution, access control is enforced at runtime by program code that has been automatically generated from a model-based specification, which captures both system and security policy descriptions. This technique can provide runtime checks only on single entities by using the guarded object design pattern. A similar approach is suggested by [BPA07], which specifies access control policies by OCL. Although this idea enables the formulation of queries that involve several objects, the efficient checking of these complex structural queries highly depends on the algorithmic experience of the system designer due to the fact that OCL handles model navigation in an imperative style, in contrast to declarative graph patterns, where several sophisticated pattern matching algorithms are readily available.

The book chapter [Luc+14] about Model-driven Security provides a detailed survey of a wide range of MDE approaches for designing secure systems, but does not cover the security of the MDE process itself.

Access Control using Bidirectional Programming. *Bidirectional Programming* (BP) is an approach for defining lenses concisely, e.g. by only specifying one of Get and PutBack, and deriving the other. Such lenses can be directly applied for read filtering. However, [FPZ09] demonstrates that conventional BP is not sufficient for write access control. It also proposes such an integrity-preserving BP extension, focusing on string transformations (and therefore not directly applicable in MDE). There is no notion of access control policy either, so the security engineer has to develop their own lens transformation to implement access control.

4.3 Bidirectional Model Transformation for Access Control Management

4.3.1 The Access Control Lens

Due to read access control, some users are not allowed to learn certain model assets. This means that the complete model (which we will refer to as the *gold* model, M_G) differs from the view of the complete model that is exposed to a particular user (the *front* model, M_F).

In theory, access control could be implemented without manifesting the front model, by hiding the entire gold model behind a model access layer that is aware of the security policy and enforces access control rules upon each read and write operation performed by the user. However, challenge **C1.1** requires users to access their front models using standard modeling tools; moreover, while challenge **C1.2** requires that in the offline collaboration scenario, they can "take home" their front model files without being directly connected to the gold model. In order to meet these goals, we propose to manifest the front models of users as regular stand-alone models, derived from a corresponding gold model by applying a *bidirectional model transformation*.

In the literature of bidirectional transformations [Dis08], a *lens* (or view-update) is defined as an asymmetric bidirectional transformation relation where a source knowledge base (*KB*) completely determines a derived (view) *KB*, while the latter may not contain all information contained in the former, but it can still be updated directly. The two operations of crucial importance in realizing a lens relationship are the following:

- *Get* obtains the derived *KB* from the source *KB* that completely determines it, and
- *PutBack* updates the source *KB*, based on the derived view and the previous version of the source (the latter is required as the derived view may not contain all information).

The bidirectional transformation relations between a gold model M_G (containing all assets) and a front model M_F (containing a filtered view) satisfies the definition of a lens. The Get process applies the access control policy for filtering the gold model into the front model. The PutBack process takes a front model updated by the user, and transfers the changes back into the gold model.

Definition 4.1 (Get Transformation). The Get process derives the front model from the gold model in accordance with the read permissions.

$$\textsc{Get} :: (M_G, \text{permission}_{\text{Eff}}) \rightarrow M_F$$

.

Definition 4.2 (PutBack Transformation). The PutBack process enforces the write permissions and derives the updated gold model from the modified front model and the original version of the gold model.

$$\textsc{PutBack} :: (M'_F, M_G, \text{permission}_{\text{Eff}}) \rightarrow M'_G$$

.

The lens concept is illustrated by Figure 4.1. Initially, the 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 readable objects may be hidden as well; finally, some attribute

values of readable objects may be omitted, obfuscated, or hidden altogether. If the user subsequently updates the front model, the PutBack operation checks whether these modifications were allowed by the write access control rules. 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).

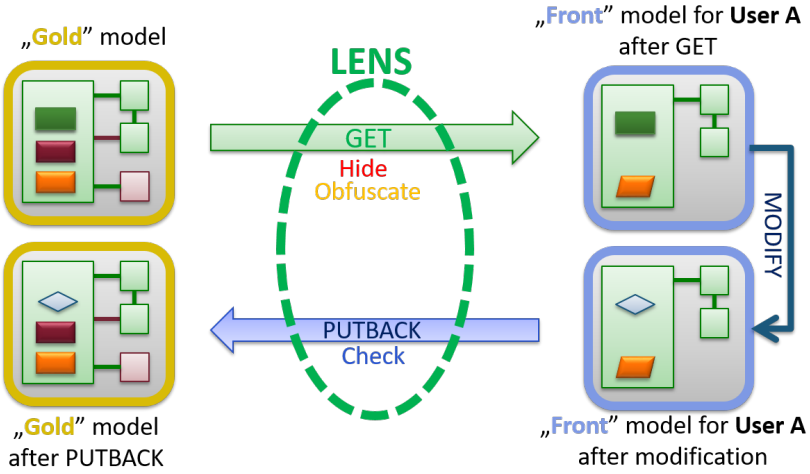
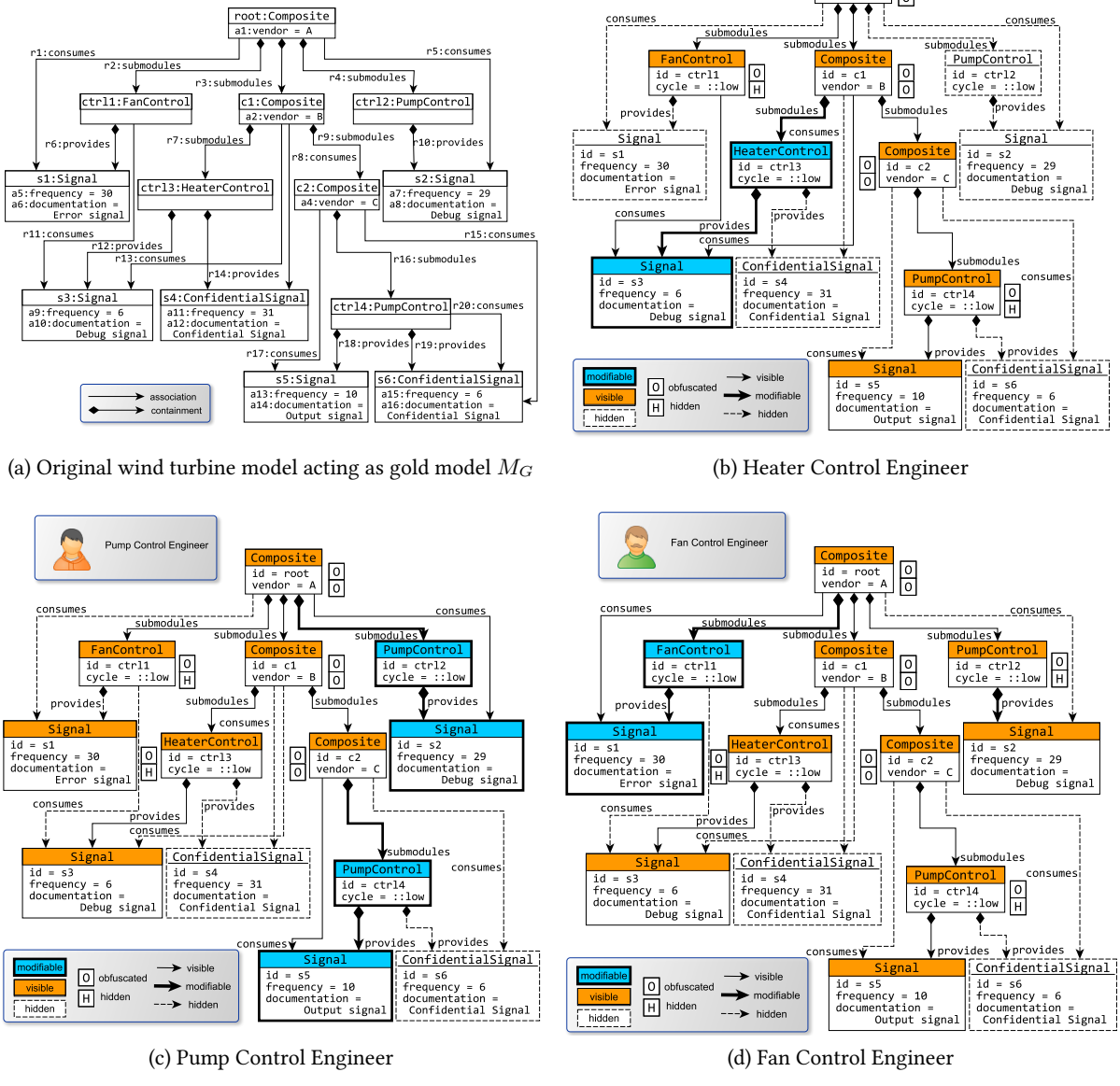


Figure 4.1: Secure Access Control by Bidirectional Lenses

Write access control checks are performed by the PutBack operation as they (a) may prevent a user from writing to the model, and (b) access control rules needs to be evaluated on the gold model.

Access control rules cannot be evaluated directly on the front model since only the gold model contains all information. Thus write access control can only be enforced by taking into account the gold model as well. Therefore, write access control must be combined with the lens transformation. In particular, PutBack must check write permissions; and fail (by rolling back any effects of the commit or operation) if a certain modification cannot be applied to the gold model.



(a) Original wind turbine model acting as gold model M_G

(b) Heater Control Engineer

(c) Pump Control Engineer

(d) Fan Control Engineer

Figure 4.2: Effective Permissions of the Example

Example 13 In our running example, the original model (Figure 4.2(a)) acts as the gold model containing all the information. The GET transformation applies the permissions and produces a front model for each specialist. In Figure 4.2, each front model consists of

- the objects with bold or solid borders;
- the references with solid lines;

but the objects and references with dashed borders and lines are removed. Whereas, the attributes marked with

- an "O" in a square are obfuscated;
- an "H" in a square are removed.

When a *PumpControlEngineer* tries to modify the frequency of the signal `s3` from 6 to 10, the PutBack operation is responsible for declining this change as the access control rules deny the modification (the signal `s3` is readable but not writable).

On the other hand, if a *PumpControlEngineer* tries to modify the frequency of the signal `s2` from 29 to 17 the PutBack operation propagates the change back to the gold model (the signal `s2` is writable) by identifying the signal `s2` in the gold model and setting its frequency attribute.

4.3.2 Model Transformation for Access Control Management

Both Get and PutBack are designed as rule-based model transformations[CH06]. In the terminology of model transformation, gold and front models act as the *source* and *target* models, respectively.

To address challenge **C2.1**, the transformations need to be *reactive* and *incremental* computations in the online collaboration scenario.

- *Reactive* transformations[Ber+15a] follow an *event-driven behavior* where the *events* are triggered by *model manipulations* such as creation/modification/deletion of model assets. The transformation observes these events and reacts to them.
- *Incrementality* [CH06] means that there is no need to re-execute the whole transformation upon a small change introduced into the model. *Source incrementality* is the property of a transformation that only re-evaluates the modified parts of the source model. *Target incrementality*, means that only the necessary parts of the target model are modified by the transformation, there is no need to recreate the new target model from scratch.

According to the process Get and PutBack of the lens, we define Tr_{Get} and $\text{Tr}_{\text{PutBack}}$ transformations, respectively. These transformations consist of four *groups of transformation rules* based on its direction (Get, PutBack) and whether it adds or removes assets from the model (*additive*, *subtractive*):

- In case of Get process:
  - *Additive* adds assets to M_F if no corresponding assets are present in M_G
  - *Subtractive* removes assets from M_F if no corresponding assets are present in M_G
- In case of PutBack process:
  - *Additive* adds assets to M_G if no corresponding assets are present in M_F
  - *Subtractive* removes assets from M_G if no corresponding assets are present in M_F

All four groups consist of one rule for each kind of model asset; we distinguish 3 kinds of model assets (see Section 3.4); this makes twelve transformation rules altogether, described in the tables of Appendices B and C.

The preconditions require to initialize correspondence between front and gold models. For that purpose, we introduce a *trace* function.

Definition 4.3 (Trace Function). The trace function is responsible for associating two object assets with each other:

$$\mathit{trace} :: (o(o_G, t)) \rightarrow o(o_F, t')$$

.

We select three example transformation rules listed in Table 4.2 to describe the key concept of how the access control is managed.

Additive Get Object rule ($\text{rule}_{\text{Additive Get Object}}$)

The *additive* rule of Tr_{Get} related to object assets is responsible for propagating object addition from

rule	Additive GET Object	Priority	4
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{ObjectAsset\}^2$			
$\{o(o_G,t), o(o_F,t') \mid o(o_G,t) \in OA_G, \text{permission}_{\text{Eff}}(o(o_G,t), \text{read}) \neq \text{deny}, \nexists o(o_F,t') \in OA_F : trace(o(o_G,t)) = o(o_F,t'), t = t'\}$			
Action			
$OA_F := OA_F \cup \{o(o_F,t')\}, trace(o(o_G,t)) := o(o_F,t')$			

rule	Additive GET Attribute	Priority	5
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{AttributeAsset\}$			
$\{a(o_F, attr, v') \mid a(o_G, attr, v) \in AA_G, \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) \neq deny,$ $\exists o(o_F,t) : o(o_F,t) \in AA_F, trace(o(o_G,t)) = o(o_F,t), \nexists a(o_F, attr, v') :$ $v' = \begin{cases} v, \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) = allow \\ obf(v), \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) = obfuscate \end{cases}, a(o_F, attr, v') \in AA_F\}$			
Action			
$AA_F := AA_F \cup \{a(o_F, attr, v')\}$			

rule	Subtractive PUTBACK Object	Priority	3
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{ObjectAsset\}$			
$\{o(o_G,t) \mid o(o_G,t) \in OA_G, \text{permission}_{\text{Eff}}(o(o_G,t), \text{read}) \neq \text{deny}, \nexists o(o_F,t') : trace(o(o_G,t)) = o(o_F,t'), t = t'\}$			
Action			
If $\text{permission}_{\text{Eff}}(o(o_G, type), write) \neq deny$ then $OA_G := OA_G \setminus o(o_G,t), trace \setminus o(o_G,t) \mid trace(o(o_G,t)) = o(o_F,t)$ else↙↘			

Table 4.2: *Additive* GET and *Subtractive* PUTBACK rules

the gold model M_G to the front model M_F. A change is recognized in the precondition which selects pairs of $o(o_G,t)$ and $o(o_F,t)$ as follows: an $o(o_G,t)$ in the gold model that has no corresponding $o(o_F,t)$ in the front model, but it should be readable according to the $\text{permission}_{\text{Eff}}$. The action part will create a new $o(o_F,t)$ and establish a correspondence relation between these two objects.

Example 14 A *system administrator* who has unlimited access to the original gold model (depicted in Figure 4.2(a)) adds a new signal object sN_G under the heater control unit $ctrl3$. This change needs to be propagated to the front models as the new signal should be at least readable (also writable for *Heater Control Engineers*).

Transformation TR_{GET} will be executed between M_G and the front model of *Pump Control Engineer* M_F^{Pump} (depicted in Figure 4.2(c)). The precondition of the $\text{RULE}_{\text{ADDITIVE GET OBJECT}}$ selects the $o(sN_G, \texttt{Signal})$ as it has no corresponding $o(sN_F, \texttt{Signal})$ in the front model. The action part creates $o(sN_F, \texttt{Signal})$ and traces it back to $o(sN_G, \texttt{Signal})$. Exactly the same sequence happens in case of the front model of *Fan Control Engineer* M_F^{Fan} (depicted in Figure 4.2(d))[a].

[a] $\text{RULE}_{\text{ADDITIVE GET REFERENCE}}$ takes care of the containment reference between sN_G and $ctrl3$.

ADDITIVE GET ATTRIBUTE RULE ($\text{RULE}_{\text{ADDITIVE GET ATTRIBUTE}}$)

The *additive* rule of TR_{GET} related to attribute assets is responsible for propagating data value insertion on the gold model M_G to the front model M_F. The precondition of the $\text{RULE}_{\text{ADDITIVE GET ATTRIBUTE}}$ selects

$a(o_G, attr, v)$ in the gold model that has no corresponding $a(o_F, attr, v')$ in the front model, but it should be readable according to the permission$_{\text{Eff}}$. The value of v' is calculated in accordance with its read permission (potentially in an obfuscated form).

Example 15 The *system administrator* modifies the frequency attribute of $s1_G$ from 30 to 15. This change needs to be propagated to the front models.

1) Tr$_{\text{Get}}$ will be executed between M_G and the front model of *Pump Control Engineer* M_F^{Pump} (depicted in Figure 4.2(c)). The precondition of the rule$_{\text{Additive Get Attribute}}$ selects the $s1_F$ object from the front model attribute $frequency$ and value 15 as $a(s1_F, frequency, 15)$ does not exist, but it should be readable in F_{pump}. The action part adds the $a(s1_F, frequency, 15)$ to M_F^{Pump}.[a]

2) Tr$_{\text{Get}}$ will be executed between M_G and the front model of *Fan Control Engineer* M_F^{Fan} (depicted in Figure 4.2(d)). But now, the precondition has no match as $s1$ is not readable in M_F^{Fan}

[a] Similarly, rule$_{\text{Subtractive Get Attribute}}$ will handle the removal of the previous attribute asset $a(s1_F, frequency, 30)$ before the addition.

Subtractive PutBack Object (rule$_{\text{Subtractive PutBack object}}$)

The *subtractive* rule of Tr$_{\text{PutBack}}$ related to object assets is responsible for propagating object asset removals from the front model M_F to the gold model M_G. A deletion is recognized in the precondition as follows: there is an $o(o_G, t)$ in the gold model that has no corresponding $o(o_F, t)$ in the front model. The action part checks the write permissions of $o(o_G, t)$. If the removal of the asset is denied, Tr$_{\text{PutBack}}$ terminates after a rollback. Otherwise, it removes the selected object asset.

Example 16 A *PumpControlEngineer*removes $ctrl1_F$ object from his front model M_F^{Pump} (depicted in Figure 4.2(c)). This change needs to be propagated be to the gold model, thus PutBack transformation will be executed between M_F^{Pump} and the gold model M_G (depicted in Figure 4.2(c)). The precondition of the rule selects $o(ctrl1_G, FanControl)$. In the action part, the rule realizes that the permissions do not allow to delete $ctrl1$ object, thus the transformation terminates and rejects the change.

To sum up, Get is responsible for enforcing read permissions in front models, while PutBack takes care of write permissions. If any write permission is violated, the transformation terminates and the front model (*target*) is reverted to its original state.

4.4 Collaboration Scheme

To satisfy our goal **G3**, a general collaboration scheme is required including the bidirectional lens transformation between a server and several clients to enforce access control policies correctly.

The server stores the gold models and clients can download their specific front models. Modifications, executed by a client, can be submitted to the server and downloaded by the other clients. These are the basic actions that nowadays, a version control system (VCS) should provide to a user. In case of various implementations, these actions may be called differently (e.g. *checkout*, *update*, *commit* in SVN or *clone*, *pull*, *push* in Git).

According to the basic actions supported by any VCS, we define the basic operations of the collaboration scheme as follows:

- ***Checkout*** downloads the model from the server-side to the workspace of a specific client who initiated the operation.

- ***Update*** retrieves the model changes from the server-side to the workspace of a specific client who initiated the operation.
- ***Commit*** propagates the changes of a specific client to the server-side.

4.4.1 Formalization of the Collaboration

Figure 4.3 and Figure 4.4 describe the behavior of collaboration scheme as *state machines* for the server and the client.

A state machine consists of *states* (represented by boxes) and *transitions* (denoted by directed edges) between states. Each state-machine has an *initial state* (denoted by arrow from a black circle) and a *current state* that specifies the system at a certain time.

The system can accept *input events* and send *output events* during its process (denoted by labels on the edges where "?" and "!" mean receiving and sending a certain event, respectively, following process-algebraic notation). In the concept of collaboration, each event is assigned to a collaborator using *"."* symbol after the name of event e.g. *input.x/output.y* means, that the collaborator x initiates an *input* and the transition produces an *output* to the collaborator y. A transition will be executed immediately when its input event arrives and during the execution it produces its output event.

Compound state (visualized as boxes containing other states) refines the behavior of a given state by defining its own state-machine where only one state can be active.

Orthogonal regions (divided by dashed borders) separate the behavior of independent states and they are processed concurrently. In each region, only one state can be active at a time.

Two state-machines can *synchronize* on events sending by one and received by the other one.

Server *(Figure 4.3).* Its state machine has three orthogonal regions to handle the *commit*, *update* and *checkout* requests concurrently.

Checkout and Update. In case of receiving *checkout* and *update* requests, our approach rejects them when a user has no access to the model itself[1] by sending an *accessDenied* event followed by a *failure* event. Otherwise, a *success* event is sent.

Upon an *update* request, it is also checked, whether the client's model is up-to-date and then an *upToDate* event is produced followed by a *success* event.

Commit. The process of receiving *commit* requests consists of `Idle`, `Locked`, `Synchronization` and `Unlock` hierarchical states:

`Idle` state accepts commit requests from any collaborator. It produces an *accessDenied* event when the user has no access to the model; or a *needToUpdate* event when the user needs to update his/her model locally to be able to commit the modifications. Both events are followed by a *failure* event. Otherwise, the system locks the model to prevent concurrent processing any other commit requests and activates the `Locked` state.

`Locked` state executes the $\text{TR}_{\text{PUTBACK}}$ in the name of the commit owner (x) and rejects the commit requests from any collaborator (y) by sending an *otherCommitUnderExecution* event with a *failure* event. After the execution of the transformation, a *policyViolated* event is sent to x, if the changes violated the access control policy and the system steps to *Unlock* state. Otherwise, a *putback* event leads the system to the `Synchronization` state.

`Synchronization` state is responsible for sending the *success* event to the owner of the commit and executing TR_{GET} to propagate the changes to other collaborators (denoted by output

[1] Note that we make a distinction between a user having no access to a model at all, and a user having access to the model, but nothing is readable in it.

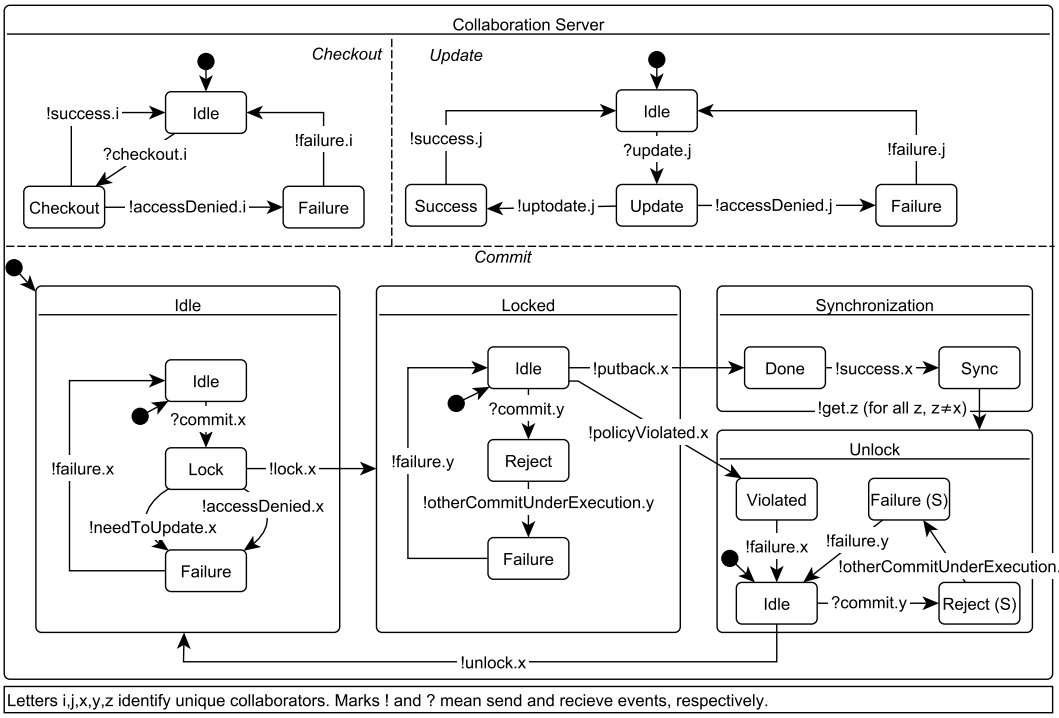


Figure 4.3: State-machine of the Collaboration Server

event *get* for all collaborator except the owner of the commit in the state *Sync*). Then systems moves forward to the *Unlock* state.

`Unlock` state is responsible for unlocking the model in all cases (*unlock* event). If the system is led to this state after a policy violation, the state produces a *failure* event before the unlock. It also rejects any other commit request by sending an *otherCommitUnderExecution* event with a *failure* event.

Client *(Figure 4.4).* Its state machine cooperates with the server using the sending and receiving events that (i) trigger an operation (*commit*,*update*,*checkout*); (ii) indicate failures (*needToUpdate*); and (iii) indicate server responses (*success*, *failure*). It consists of `Checkout`, `Idle` and `Update` hierarchical states:

Checkout State. First, the clients need to checkout their models represented by sending a *checkout* event in the `Checkout` state. Based on the received server response, the clients can move to `Idle` state.

Idle State. In `Idle` state, clients can commit or update their changes by sending *commit* or *update* events. All the events produced by the server can be received, but only the *needToUpdate* event restricts the behavior of the client by moving to `Update` state.

Update State. The clients need to initiate an update request by sending a *update* event to be able to commit their changes again.

4.4.2 Correctness Criteria

To address the challenge **C3.1** we describe the *correctness criteria* that the collaboration scheme needs to satisfy:

Criterion 1. The scheme needs to be deadlock free (i.e. all the locks need to be unlocked during a commit operation).

Criterion 2. The scheme needs to be livelock free (i.e. all the operations need to finish at some point and lead the scheme to an idle state).

Criterion 3. Commit operation shall be rejected while another commit is under execution.

Criterion 4. Commit operation shall propagate the changes to all collaborators.

Criterion 5. Clients need to initiate an update operation when it is required by the server.

Note that Criterion 1. and Criterion 2. are required to ensure that the collaboration can run without any manual intervention. Criterion 3. declines overwriting changes without notification of a commit happened previously. Criterion 5. enforces the clients to avoid conflicting commits.

4.4.3 Proof of Correctness

In accordance with **C3.1**, we formalized our collaboration scheme as *communicating sequential processes (CSP)* [Ros98; Ros10] described in the Appendix A to prove its correctness. *CSP* is a formal specification language of concurrent programs or systems where the communications and interactions are presented in an algebraic style.

The **Server** and **Clients** processes define the behavior of exactly 1 server and n clients, respectively. The collaboration is specified as a concurrent execution (denoted by $||$) of the server and clients where the processes synchronize on a given set of events *SyncEvents*: {*commit, update, checkout, accessDenied, policyViolated, needToUpdate, failure, success*}.

$$\textbf{Collaboration} = \textbf{Server}||_{\mathit{SyncEvents}}\textbf{Clients}[1..n]$$

For the analysis, we used the *FDR4 tool*[Gib+14] to evaluate *assertions* over certain properties of the processes.

Criteria 1. and 2. requires the entire collaboration process (**Collaboration**) to be deadlock and livelock free. To check these properties, the : $[deadlock\ free]$ and : $[divergence\ free]$ built-in structures are used, respectively.

$$assert\ \textbf{Collaboration}\ : [deadlock\ free] \tag{4.1}$$

$$assert\ \textbf{Collaboration}\ : [divergence\ free] \tag{4.2}$$

The rest of the criteria requires to evaluate whether the process formally refines a certain event sequence according to the CSP models, namely the *traces* and *failures* models. For that purpose, we use the $\mathbb{T}$: and $\mathbb{F}$: structures, respectively, where

P $\mathbb{T}$: $< a, b, c >$ means that process **P** must be able to perform the ordered sequence of the events a, b, c.

P $\mathbb{F}$: $< a, b, c >$ means that process **P** must not be able to refuse to perform the ordered sequence of events a, b, c without performing any other event.

Informally, *traces model* $\mathbb{T}$ states what the CSP model can do (e.g. it can perform a then b and finally c, but in the mean time, it can perform anything). On the other hand, *failure model* $\mathbb{F}$ defines what the CSP model must do (e.g. it must perform a then b and finally c).

To check the remaining criteria, we introduce the $\neg$ symbol to negate assertions; the $\setminus$ symbol that hides events from the process and $\mathcal{E}$ denotes the events provided by all processes. The combination of

these symbols allows us to evaluate the processes in the context of certain event, e.g. $\mathbf{P} \setminus (\mathcal{E} \cap \{a, b, c\})$ means that all events are hidden from the process **P** except a, b and c.

Criterion 3. includes that after executing a $\text{Tr}_{\text{PutBack}}$, another $\text{Tr}_{\text{PutBack}}$ cannot be executed without unlocking the model.

$$
\begin{aligned}
&assert\ \mathbf{Server} \setminus (\mathcal{E} \cap \{unlock, putback\}) \\
&\neg \mathbb{T} : < putback.x, putback.y > \\
&x, y \in Int, x \neq y
\end{aligned}
\tag{4.3}
$$

Criterion 4. requires to execute Tr_{Get} for all collaborators other than the owner of the commit after a successfully executed $\text{Tr}_{\text{PutBack}}$ but before unlocking the model. As we start the synchronization with collaborator 1, and then 2, it implies that the collaboration scheme needs to execute it to the last collaborator, namely n.

$$
\begin{aligned}
&assert\ \mathbf{Server} \setminus (\mathcal{E} \cap \{get.n, putback, unlock\}) \\
&\mathbb{T} : < putback.x, get.n, unlock.x > \\
&x, N \in Int, x \neq n
\end{aligned}
\tag{4.4}
$$

To satisfy Criterion 5., after a commit operation rejected by the server with a *need to update* message, the client (i) cannot commit again and (ii) must be able to initiate update operation:

$$
\begin{aligned}
&assert\ \mathbf{Clients} \setminus (\mathcal{E} \cap \{commit, update, needToUpdate\}) \\
&\neg \mathbb{T} : < commit.x, needToUpdate.x, commit.x >
\end{aligned}
\tag{4.5}
$$

$$
\begin{aligned}
&assert\ \mathbf{Clients} \setminus (\mathcal{E} \cap \{commit, update, needToUpdate\}) \\
&\mathbb{F} : < commit.x, needToUpdate.x, update.x > \\
&x \in Int
\end{aligned}
\tag{4.6}
$$

As in the assertions we hide several events, the *FDR4 tool* was able reduce the *state space* and the transitions that needs to be traversed.

We evaluated the assertions[2] for $n = 5$ users. The results are presented in Table 4.3. To check deadlock and livelock properties, all the events and states are required. Hence the tool traversed almost 90000 states and 230000 transitions to prove these properties. To verify the rest of the assertions, at most 500 states and 1100 transitions were enough to traverse. All the assertions are evaluated within less than 0.51 seconds and none of them failed.

However this proof does not formally generalize for more users, it still shows a proof for 5 users which provides sufficient trust for most use cases from software engineering perspective.

Table 4.3: Results of the assertion in *FDR4 tool*

		States	Transitions	time (s)
Assertions	4.1	89233	227591	0.44
	4.2	89233	227591	0.51
	4.3	452	1121	0.13
	4.4	417	1071	0.42
	4.5	10	11	0.12
	4.6	10	11	0.12

[2]The complete formal specification is available at: http://goo.gl/pJzIX1

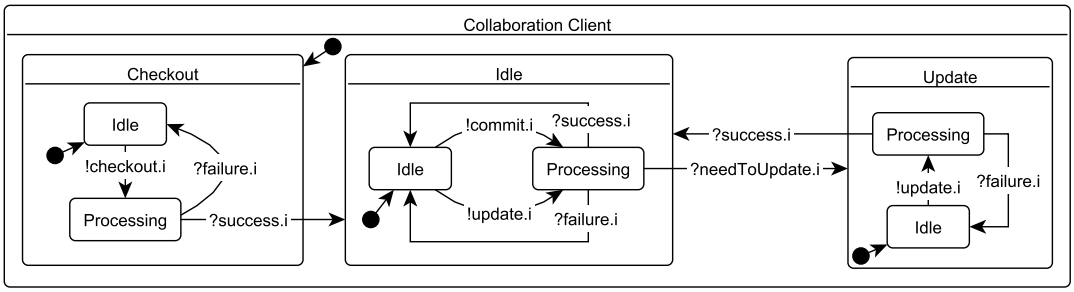


Figure 4.4: State-machine of the Collaboration Client

4.5 Realization of Collaboration Scheme

In accordance with **C3.2**, our goal is to provide tool support[3] for enforcing fine-grained model access control rules in *offline* and *online* scenario realizing the introduced *collaboration scheme* (see Section 4.4).

4.5.1 Offline Collaboration

In the offline scenario, models are serialized (e.g. in an XMI format) and stored in a Version Control System (VCS). Users work on local working copies of the models in long transactions called commits. The goal of our approach is to manage fine-grained access control on the top of existing security layers available in the VCS.

4.5.1.1 Realization

The concept of *gold* and *front* models is extended to the repository level where the two types of repositories are called *gold* and *front* repositories as depicted in Figure 4.5.

The *gold* repository contains complete information about the gold models, but it is not accessible to collaborators. Each user has a *front* repository, containing a full version history of front models. New model versions are first added to the front repository; then changes introduced in these revisions will be interleaved into the gold models using PutBack transformation. Finally, the new gold revision will be propagated to the front repositories of other users using Get transformation. As a result, each collaborator continues to work with a dedicated VCS as before, thus they are unaware that this front repository may contain filtered and obfuscated information only.

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. These regular users can use any compatible VCS client to communicate with their front repository, being unaware of collaboration mechanisms in the background.

This scheme enforces the access control rules even if users access their personal front repositories using standard VCS clients and off-the-shelf modeling tools. Nevertheless, optional client-side collaboration tools may still be used to improve user experience, e.g. for smart model merging, user-friendly lock management(see Chapter 5), or preemptive warning about potential write access violations that greatly enhances the usability and applicability of the offline scenario.

[3]Source codes and more details are at https://tinyurl.com/sosym-access-control-source

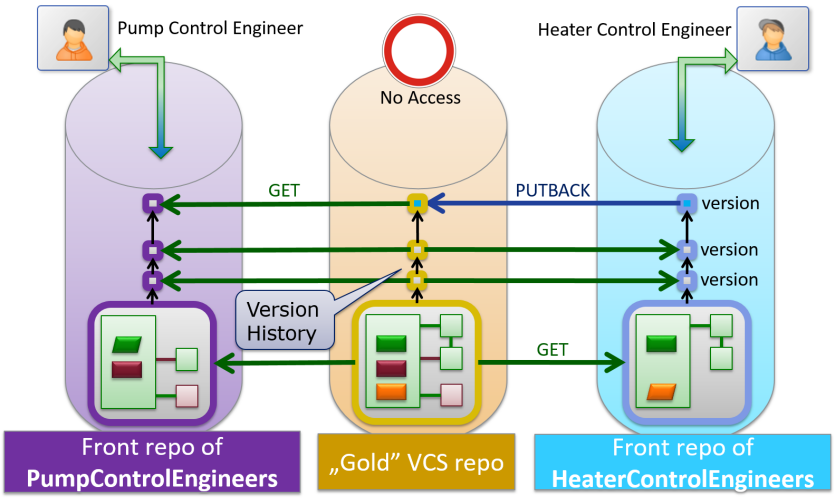


Figure 4.5: The MONDO Offline Collaboration Server: Architecture

4.5.1.2 Realization of the Collaboration Scheme

In the current prototype, our collaboration scheme is realized by extending an off-the-shelf VCS server, namely *Subversion*[SVN]. Subversion provides features of the collaboration scheme by default:

- *File-level access control* is responsible for sending *accessDenied* event to the collaborators whenever their access is denied for a certain file (which contains models).
- *Version-control* allows to the users to download the files by sending a *checkout* event, submit their changes by sending a *commit* event and update the files by sending an *update* event.
- *Version check* checks the version of the files and sends *upToDate* the collaborators whether the files are already up-to-date upon an update or sends *needToUpdate* event they need to update upon a rejected commit.
- *File-level locking* allows users to lock files by sending a *lock* event and reject commits initiated by other users. They can also remove their locks by sending an *unlock* event.
- *Handling Multiple Requests* allows users to initiate multiple requests simultaneously that the server can accept.
- *Final Notification* notifies the users about the result of their requested operations by sending a *success* or a *failure* event.

As checkout and update operations of the collaboration scheme are fully handled by Subversion, we need to integrate the $\text{Tr}_{\text{PutBack}}$ and Tr_{Get} into the commit operation to enforce fine-grained access control and propagate the changes.

Hooks are programs triggered by repository events such as *lock*, *unlock* or *commit*. The hook may be set up to be triggered before such an event (with the possibility of influencing its outcome, e.g. cancelling it upon failure) or directly afterwards (when the event is guaranteed to have happened).The following hook programs will be executed upon a commit operation.

Pre-Commit Hook. $\text{Tr}_{\text{PutBack}}$ is invoked by *pre-commit hook* executing when a user attempts to commit a new revision of a model $M_F^{r'}$ (new revision r' of a model M_F). This hook performs the following steps to enforce access control policies corresponding to Figure 4.6:

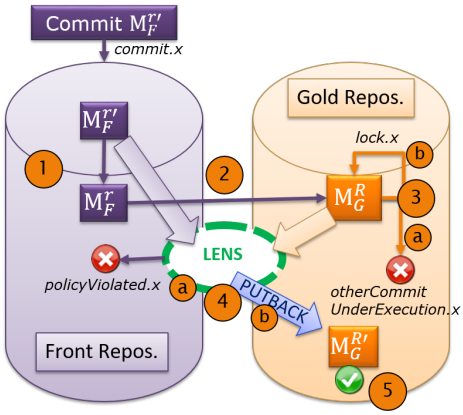


Figure 4.6: Pre-commit hook at the front repository

1. Parent revision M_F^r of $M_F^{r'}$ is identified.
2. Revision M_F^r is traced to the corresponding revision M_G^R in the gold repository.
3. The hook attempts to put a file-level *lock* to M_G in the gold repository.
   a) If the locking attempt fails, the hook terminates sending an *otherCommitUnderExecution* event.
   b) Otherwise, the lock on M_G is activated by sending a *lock* and the hook continues its process.
4. $\text{TR}_{\text{PUTBACK}}$ is executed between $M_F^{r'q}$ and M_G^R in the gold repository, in order to reflect the changes performed in the new commit.
   a) If the $\text{TR}_{\text{PUTBACK}}$ detects any attempts to perform model modifications violating write permissions, then the commit process to the front repository terminates by sending a *policyViolated* event.
   b) Otherwise, the commit is deemed successful, and $M_G^{R"}$ is committed to the gold repository (with metadata such as committer name and commit message copied over from the original front repository commit).
5. Finally, the hook finishes successfully and let the VCS server to handle the request.

Post-Commit Hook. TR_{GET} is invoked by *post-commit hook* synchronizing all front models M_F^r with the new revision of gold model $M_G^{R'}$. This hook is triggered after a commit of $M_G^{R'}$ finished successfully at the gold repository and performs the following steps correspond to Figure 4.7 to propagate the new changes.

1. Parent revision M_G^R of $M_G^{R'}$ is identified.
2. The hook iterates over each front repository and execute the following steps. If the commit to the gold repository is initiated by a front repository then the originating front repository will be skipped.
   a) Revision M_G^R is traced to the corresponding front revision M_F^r in the front repository.
   b) TR_{GET} is executed between $M_G^{R'}$ and M_F^r in order to reflect the changes performed in the commit. If M_F^r does not exist, it is handled as an empty model.
   c) New revision of the model $M_F^{r'}$ is commited to the front repository (with metadata such as committer name and commit message copied over from the original front repository commit).
3. The hook removes the lock from $M_G^{R'}$ by sending an *unlock* event.

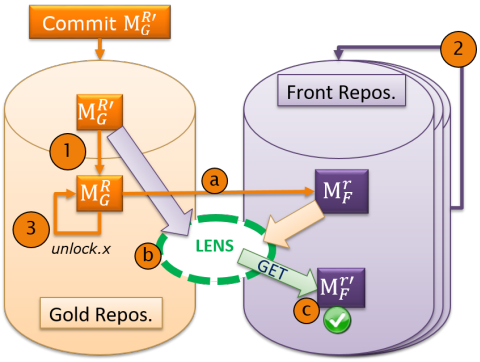


Figure 4.7: Post-commit hook at the gold repository

4. Finally, it finishes successfully and lets the VCS server to handle the request.

4.5.1.3 Discussion

It is worth discussing the following properties of the offline collaboration framework.

Generality. Our solution is general and adaptable to any VCS that supports *checkout*, *update*, *commit* operations (maybe they are named differently).

Server Response. Users get response to their commit right after the collaboration server attempts to propagate back the changes to the gold repository. If any access control rule is violated the pre-commit hook fails. At the last phase of pre-commit, the VCS declines the commit action to the gold repository, if any modified files are locked on the gold repository. Hence, the hook fails again and prevent the VCS specific file level locks. In contrast, if everything goes well the users do not need to wait for synchronizing with the remaining front repositories.

Multiple Models in a Commit. A single commit may update several models at once. In this case, the hooks are invoked for each model in the commit.

Non-blocking Commit. Commit operation does not block update and checkout operations as previous versions still readable in the front repositories.

Models stored among other project files. Our solution supports storing models along with non-model files in the repositories. The hooks can be parameterized with file extensions to determine whether a file needs to be handled as a model. When a file is not a model, it is simply copied from the gold repository to the front repositories.

Correspondence Relation. It is a challenging task to identify the correspondences between model assets of the front and gold models, where the models are stored independently. Our approach currently uses specific attributes to provide permanent identifiers. Such a permanent identifier is preserved across model revisions and lens mappings, and can therefore be used to pre-populate the object

correspondence relation. In our running example, each object has a unique *id* attribute. Note that unlike EMF, some modeling platforms (e.g. IFC [IFC]) automatically provide such permanent identifiers.

While requiring permanent identifiers is a limitation of the approach, it is only relevant for modeling platforms that do not themselves provide this kind of traceability, and only in the offline collaboration scenario. Being able to identify model objects is a relatively low barrier for modeling languages; e.g. the original wind turbine language includes a unique identifier for all model objects.

Authorization Files. 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, but readable by every user; this means that offline clients may evaluate security rules 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 principal. In any way, names and parameters of security rules should not themselves contain sensitive design information.

4.5.2 Online Collaboration

In the online scenario, several users can simultaneously display and edit the same model with short transactions by using a web-based modeling tool where changes are propagated immediately to other users during *collaborative modeling sessions*. In contrast to the offline scenario, where users manipulated local copies of the models, models are kept in a server memory and users access the model directly on the server. The goal of our approach in accordance with *C3.1* is to incrementally enforce fine-grained model access control rules and on-the-fly change propagation between view models of different users.

4.5.2.1 Technical Realization

During a collaborative modeling session, a model kept in server memory for remote access may also be called a *whiteboard* depicted in Figure 4.8. 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 PutBack, and propagated further to the other front models using Get again. In case of online collaboration, these lens operations are continuously and efficiently executed as a *live transformation*[CH06], thus users always see an up-to-date view of the model during the editing session.

Similarly to modern collaborative editing tools (such as Google Sheets [Con08]), 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 demand ("save button"). The *session manager* component enables collaborators to start, join or leave whiteboard sessions and persist models to disk.

4.5.2.2 Realization of the Collaboration Scheme

To achieve challenge **4.2**, we need to discuss how the online collaboration realizes the collaboration scheme.

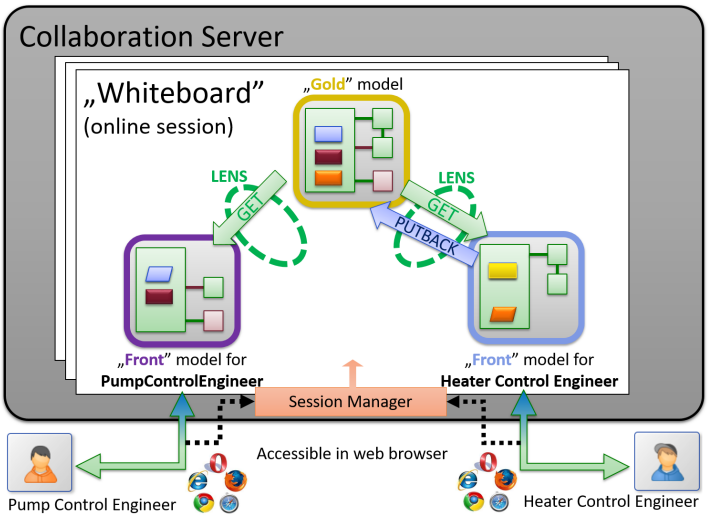


Figure 4.8: Overview of Online Collaboration

1. The *checkout* operation is equivalent to joining the whiteboard session for the first time, except it requires to execute $\mathrm{TR}_{\mathrm{GET}}$ that achieves the front model on which the new user can work.
2. The *update* operation is equivalent to refresh the browser on client side. Via web-based technologies, the collaboration framework notifies and forces the clients' browsers to refresh when new changes are introduced into their front models. However, manual refresh usually results in an *upToDate* event, except when the notification and the manual refresh initiated at the same moment.
3. The *commit* operations are initiated right after users apply modifications on their front models. Other clients need to wait (receiving *otherCommitUnderExecution* event) until the commit finishes, including the execution of PutBack and all Get processes to propagate the changes. After a successful commit, clients receive notification to force them to initiate and update. When a *policyViolated* event occurs, the change is immediately rolled back at the initiator's front model.

4.5.2.3 Discussion

It is worth discussing the following properties of the online collaboration framework.

Conflicts Handling. As the online collaboration operates with short transactions, it has only a small chance that conflict occurs during the session (e.g. a collaborator modifies an object that is deleted by another collaborator and the propagation of the deletion is under execution). However, if a conflict araises, it is resolved by accepting the remote changes. It also implies that the latter changes will be lost.

Blocking Checkout and Update. Version numbers are not considered in online collaboration and only one gold model exists during a session. Hence, checkout and update operations need to wait until the commit operation finishes if these operations were initiated during the execution of a commit.

Correspondence Relation. In the online case, the gold and its front models are initiated for a collaboration session. During a session, these models are stored in the memory and there is no need to reload any of them. Correspondences established during TR_{GET} can be used through the online collaboration. Hence, there is no limitation about the models (unlike in the offline case, where unique identifiers are required).

Integration with Offline Collaboration. Models and authorization files can be persisted to an underlying gold repository provided by a VCS. The online collaboration tool can access them using checkout/update/commit commands. However, if file-level conflicts occur in the underlying VCS, they will need specific user interfaces to resolve them. Instead, we decided that new whiteboard sessions put file-level locks on the resources related to the models to prevent conflicts in the VCS upon persisting.

4.6 Evaluation

We have carried out a scalability measurement in both offline and online scenario over the *Wind Turbine* case study[Bag+14] of the MONDO FP7 project. We state the following research questions in the evaluation:

Online Collaboration

Q1.1 How scalable is our approach to increasing *model size*?

Q1.2 How scalable is our approach to increasing number of active *users*?

Offline Collaboration

Q2.1 How scalable is our approach to increasing *model size*?

Q2.2 How scalable is our approach to increasing number of *front repositories*?

Q2.3 How scalable is our approach to increasing size of committed *changes*?

Finally, Section 4.6.2 will discuss limitations of our solution.

4.6.1 Scalability Evaluation

4.6.1.1 Measurement Setup

For the measurement, we used the simplified metamodel of Figure 2.1 depicted in Figure 4.9 which has slight modifications. The control unit `types` were abstracted to a string attribute, with K different permitted values to provide K different specialist, and the attributes of signals are removed. The corresponding access control rules are similar to our motivating example, with one specialist engineer for each control unit type (each having five access control rules dedicated to them) and an additional *system administrator* user who has read and write permission for the entire model. This means altogether $K + 1$ users and $5K + 5$ access control rules as it is shown in Figure 4.10.

Measurements were performed with gold instance models of various size. The model of size M contains a root `Composite` object, which contains M copies of the structure depicted in Figure 4.11. This means $1 + M$ composite modules, $2M$ control units, $8M$ signals where $3M$ of them are confidential and $14M + M$ references where $4M$ of them are *consumes* cross-references. The copies are not completely identical: the `vendor` attributes are set to a different value in each copy; and `type` as well as `cycle` attributes of control units were chosen randomly from their respective ranges with uniform distribution. However, special care was taken to ensure that all control unit types must occur at least once; this also implies $2M \geq K$.

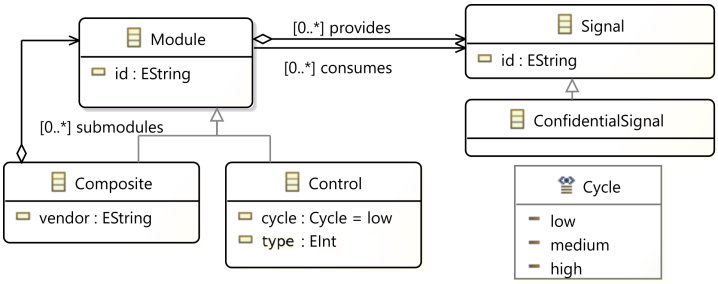


Figure 4.9: Modified Metamodel of Wind Turbine

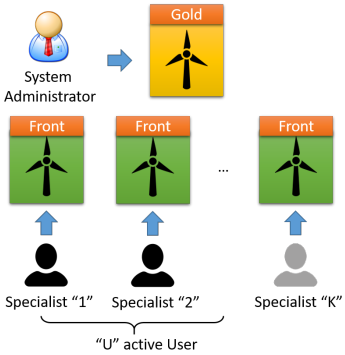


Figure 4.10: Users and active user in the measurement setup

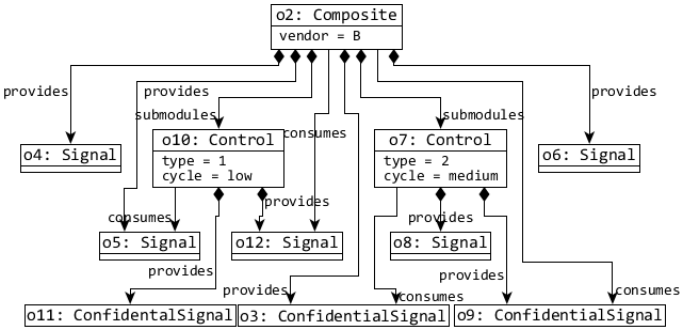


Figure 4.11: Core structure of synthesized models

The measurement was performed with $U \leq K$ specialist users and the system administrator being present (thus in total $U + 1$ front models).

Online case. To test the online behavior of the lens transformation, we measured the time it took the system administrator to perform a complex model manipulation operation on his front model, and to propagate the changes to the front models of all users who can see it. The measured complex operation is a *signal reversal* (depicted in Figure 4.12), which reverts the direction of a communication channels by changing the *provides* and *consumes* to the opposite.

We have selected this representative operation since (a) it involves adding and removing cross-references and a rearrangement of the containment hierarchy; (b) it does not change the size of the

model, thus introduces no bias of this kind; (c) the change is noticeable by all users that can see at least one of the involved modules in their front models; and (d) every access control rule in the policy (except for hiding the vendor attribute) plays a role in determining the impact of the change.

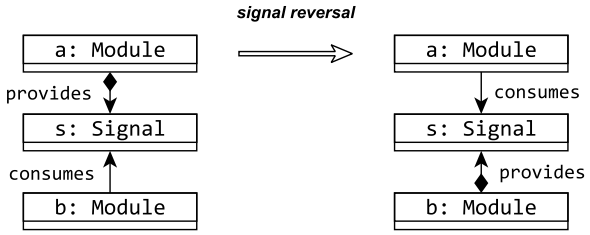


Figure 4.12: Signal Reversal Operation

Offline case. The measurement focuses on the overhead of the collaboration framework required for propagating a change in the front model. We measured the time it took a specific specialist to (1) propagate several number of complex model manipulation operation on her front model to the gold model and (2) from the gold model to the remaining front models of all users who can see the effect of the changes. The former describes the *response time* that a user has to wait for receiving the result (success/failure) of her commit while the latter is the *propogation time* to propogate changes to the other front repositories. The measured complex operation is a *signal addition* (depicted in Figure 4.13), which adds a new signal under the root object.

We have selected this representative operation since (a) it demonstrates that any number of new changes can be introduced into the model; (b) it increases the model size but always with constant-size addition; (c) all the users can see the change in their front model; and (d) every access control rule in the policy (except for hiding the vendor attribute) plays a role in determining the impact of the change.

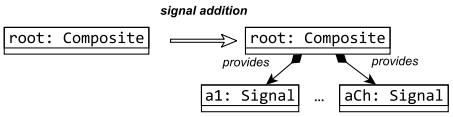


Figure 4.13: Signal Addition Operation

Hardware Configuration All the measurements[4] executed on a personal computer[5]. with maximum a 7GB of Java heap size.

4.6.1.2 Measurements of Online case

In the *model size scalability* series, we used

- fixed number of $K = 50$ control unit types and $U = 10$ present collaborators

[4]Raw data and reproduction instructions at https://tinyurl.com/sosym-access-control

[5]CPU: Intel Core i7-4700MQ@2.40GHz, MEM: 8GB

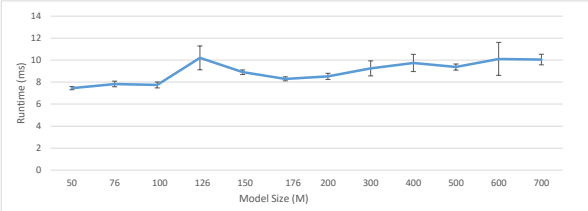


Figure 4.14: Average execution time of an online signal reversal (increasing model size)

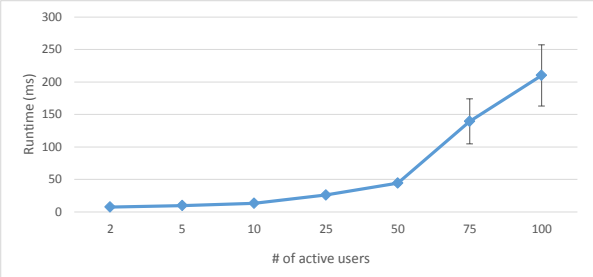


Figure 4.15: Average execution time of an online signal reversal (increasing the number of active users)

- with increasing size of the model ranging from $M = 50$ to $M = 700$ (7701 objects, 10500 references)

In the *active users scalability* series, we used

- fixed number of $K = 100$ control unit types and model of size $M = 200$ (2301 objects, 3100 references),
- with the increasing number of specialist collaborators joining the session ranging from $U = 2$ to $U = 100$.

For accuracy, 100 reversal operations were carried out and their execution times averaged in a single run; we have plotted the median execution time of 10 runs, excluding 2 warm-up runs, with 1 standard deviation error bars.

Addressing Q1.1 The results of the *model size scalability* series are shown in Figure 4.14 . The cost of performing a single reversal model manipulation is low, and seems to be independent from the model size. This confirms that we have achieved incrementality where computation cost is dependent on the size of the change, but not on the size of the entire model.

Addressing Q1.2 The results of the *active users scalability* series are shown in Figure 4.15. It is apparent that when very few users join the session, most signal reversals are not visible to any user other than the principal engineer; but as more and more specialist users join the session, the number of active users starts to dominate the cost of model manipulation. Asymptotically, the cost of model manipulation is proportional to the average number of front models it is propagated to.

Note that it has only a small chance that users concurrently modify their front model, but in that case the operations which arrive later to the server will be rejected. Hence, concurrent modifications have no additional effect on the performance.

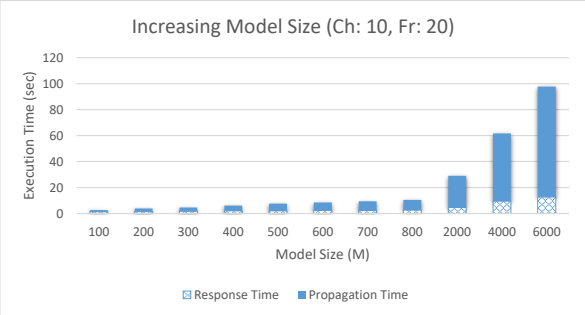


Figure 4.16: Average execution time of an offline signal addition (increasing model size)

4.6.1.3 Measurements of Offline case

In the *model size scalability* series, we used
- fixed number of $K = 100$ control unit types, $Fr = 20$ front repositories and $Ch = 10$ changes,
- where the model is increased from $M = 100$ to $M = 6000$ (34001 objects, 45000 references).

In the *number of front repository scalability* series, we used
- fixed number of $K = 100$ control unit types, model of size $M = 800$ (8801 objects, 12000 references) and $Ch = 10$ changes,
- where the number of front repositories is increased from $Fr = 5$ to $Fr = 100$.

In the *change size scalability* series, we used
- fixed number of $K = 100$ control unit types, model of size $M = 400$ and $Fr = 20$ front repositories,
- where the number of introduced changes is increased from $Ch = 10$ to $Ch = 1000$.

The measurements were executed 10 times with 2 warm-up execution in separate JVM and the results show the median of the measured values.
The charts represent the entire transformation time including the following tasks: (1) loading the EMF models, (2) initializing the lens by building the correspondence tables, (3) loading the additional files such as rules and queries, (4) executing the transformation and (5) finally serializing the results as a committable new version of the models.

The lower part of the bars (denoted by checkered blue background) represents *response time* including the PutBack phase of the transformation. This is the delay experienced by committing users before they receive their response from the server so that they can continue their work. The upper part of the bars (in solid blue color) visualizes *propagation time* of the changes to synchronize with the rest of the front repositories (this happens asynchronously from the committing user's viewpoint).

Addressing Q2.1 The results of *model size scalability* series are shown in Figure 4.16. In case of the largest model, users should wait at most 10 seconds to commit their changes in addition to the default execution time of a commit in the version control system. Response time grows linearly with the size of the model while synchronization is non-linear to the model size.

Addressing Q2.2 The results of *number of front repository scalability* series are shown in Figure 4.17. In case of our special signal addition change, all front repositories had to be updated to propagate the changes and the same number of modification had to be executed on those front models. The results clearly show that the execution time grows linearly to the number of front repositories to which the change has to be propagated.

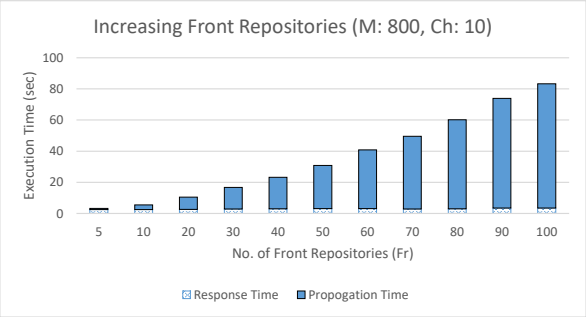


Figure 4.17: Average execution time of an offline signal addition (increasing number of front repositories)

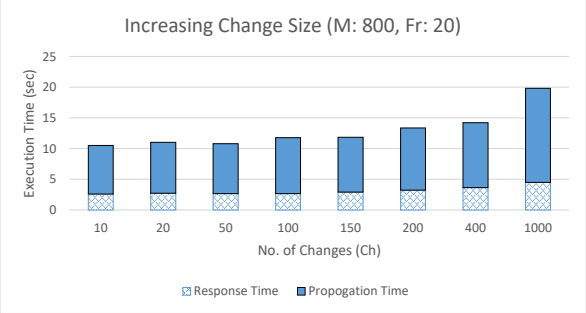


Figure 4.18: Average execution time of an offline signal addition (increasing number of changes)

Addressing Q2.3 The results of the third series are shown in Figure 4.18. It shows that loading the models and building the correspondence table dominate the execution time in case of small changes. However, the execution time grows linearly with the size of changes in case of large commits (e.g. from 1000), for the sole reason that the resulting model size itself is increased due to the addition of so many new signals.

4.6.1.4 Discussion on Performance Findings

As seen from the measurements, the overhead on the commit time experienced by a committer in the offline scenario (see **Q2.1**) is manageable, but can easily reach several seconds for larger models with tens of thousands of elements. This time is still significantly shorter than typical build and test execution times in continuous integration solutions, so our access control service is unlikely to form the bottleneck of developer productivity. Note that while the implementation of our prototype could certainly be improved, it will always have an overhead that is at least proportional to the model size, since the entire new model file has to be read and processed upon each commit (no matter how small the change within that model). This is a characteristic of file-based offline collaboration (required to meet goal G1).

One way to get around this limitation is to use online collaboration instead, where the execution time overhead on model modification is very low, even for a few dozen simultaneous collaborators (see **Q1.1** and **Q1.2**). This can be seen as a space-time tradeoff, as online collaboration uses in-memory

models, putting a limit on the amount of online sessions and participants that a given server can support. We therefore recommend the adaptation of online collaboration whenever possible for models that are currently under very active development, and using the file-based offline interface for other cases, such as accessing rarely updated models, old revisions, side branches, and of course for working on a disconnected computer.

4.6.2 Assumptions and Limitations

Feedback on Write Access Control As discussed before, the means of write access control is the following. In the offline case, the server rejects unauthorized modifications only when the user finally submits them. In the online case, PutBack is a live transformation, and it can immediately reject non-compliant changes. (Note that rejected write attempts offer a *side channel* through which some information on the hidden parts of the gold model may be gained. While it is beside the point here, policy designers are advised to take such unintended effects into account.)

It can be frustrating and unproductive for users to learn about their insufficient permissions by trial and error. This is especially true in the offline case, where the feedback only arrives when modifications are actually committed. In a better system, write restrictions would be readily available to the user; advanced modeling tools may even incorporate this information into their model notation, e.g. to visually show read-only parts of the model as frozen.

However, such a tight feedback loop in the offline case would either require nonstandard communication channels (with their own security risks) to disclose the evaluated permission sets with the client; or alternatively, additional computations such as client-side approximation of the policy queries based on the incomplete information in the front model. Proposing a satisfactory solution is left as future work, e.g. by elaborating initial ideas of [*c*16].

Ordered Lists Some multi-valued references and attributes are *ordered* lists. Model assets introduced in Section 3.4 collectively represent all knowledge contained in a model except for ordering information.

Thus the lack of ordered lists is a limitation of the proposed solution. The core reason is that there is no unique way to provide PutBack for ordered lists that have been filtered; therefore such a lens would necessarily violate at least *undoability*. Finding an acceptable resolution of the problem (e.g. imposing a limitation that, for each user, ordered lists must be read-only unless entirely visible) is left as future work. For now, the proposed solution works properly for unordered collections.

Central Authority Note that both **G1** and **G2** assume a central repository (owned by e.g. a system integrator) where the entire model is available. In a more general case, no single entity would be in possession of complete knowledge. There is an algebra [Dis08] for combining lens transformations in various ways, suggesting a promising path for addressing this issue in future research. However, such a distributed scenario is out of scope; we address the centralized case, which is by far the most common in access control approaches used in model repositories.

4.7 Contributions

In this chapter, we aimed to uniformly enhance secure collaborative modeling by using fine-grained access control policies uniformly for online and offline collaboration scenarios. Each collaborator can access a dedicated copy of the model in accordance with read permissions of the policy. Moreover,

bidirectional transformations are used to synchronize changes between different collaborators and check that write permissions are also respected.

We illustrated our techniques in the context of a Wind Turbine case study from the MONDO European Project, which was also used to assess scalability with models of increasing size, increasing change introduced by collaborators and increasing number of collaborators. In case of online collaboration, the results were promising with close to instant propagation of changes and checking of write permissions. In case of offline collaboration, the results show that the response time is acceptable and the overhead is less than 10 additional seconds for the largest model).

Contribution 2 I formalized the enforcement of high-level fine-grained access control policies and realized provenly secure collaborative architecture the enforces such policies. [*j*1], [*j*2], [*c*7], [*c*6], [*c*9], [*c*10], [*c*14]

- **2.1 Formalization of Bidirectional Rules for Secure Views.** I formalized transformation rules to derive secure front models with respect to the read and write permissions. [*j*2], [*c*14]
- **2.2 Secure Collaboration Scheme.** I formalized a collaboration scheme as *communicating sequential processes* (CSP) to enforce high-level access control policies. I specified correctness criteria and proved the correctness of the scheme. [*j*2]
- **2.3 Realization of Secure Collaboration.** I realized the collaboration scheme in case of offline scenarios by extending an existing version control system to enforce fine-grained access control while collaborators can use off-the-shelf tools. [*j*1], [*c*6]
- **2.4 Evaluation.** I evaluated the scalability of the proposed architecture on a case study of offshore wind turbine controllers. [*j*2], [*c*9], [*c*14]

The bidirectional transformation to enforce access control rules is the contribution of Gábor Bergmann whereas the concept of the common architecture to support both online and offline scenarios is the contribution of István Ráth.

Chapter 5

Conflict Reduction and Handling

5.1 Introduction

Section 1.3.1 discusses the challenge of conflict prevention and resolution in collaborative modeling (**C-II**). The main objective of this chapter is to address **C-II** by proposing a model merge approach [*c*13] to automatically resolve conflicts where well-formedness constraints are also taken into account as well as a realization of *property-based locking* [*c*5] that enables a high degree of collaborative development. In particular, we aim to address the following model merge (MG) and locking (LG) related goals:

MG1 *Automated conflict resolution.* The solution shall provide a way to automatically resolve conflicting changes.

MG2 *Domain-specific knowledge.* The solution shall allow to incorporate domain-specific knowledge into the merge process to provide better solutions.

MG3 *Generic scalability benchmark.* The solution shall provide a generic scalability benchmark for assessing model merge performance for large models and large change sets.

LG1 *Describe locks as properties.* The solution shall provide a way to describe complex properties suitable for specifying locks.

LG2 *Preserve property-based locks.* The solution shall enforce locks described as properties of the models, and only allow modifications that preserve the defining properties of active locks.

LG3 *Support for existing locking strategies.* The solution must provide means to support existing locking strategies (such as traditional fragment and object-based locking).

Structure Rest of the chapter is organized as follows. Additional preliminaries are discussed in Section 5.2. Related work is overviewed in Section 5.3. In Section 5.4, we propose our approach for automated model merge where Section 5.4.1 provides a detailed motivating example for model merge, Section 5.4.2 overviews the concept of our approach, Section 5.4.3 describes the execution process, while we apply the approach to the example in Section 5.4.4. A generic scalability benchmark is proposed in Section 5.4.5 including the evaluation our approach. Property-based locking is discussed in Section 5.5 where Section 5.5.2 introduces our choice for specifying such model queries, Section 5.5.3 follows with the definition of an entire lock. Section 5.5.4.1 and Section 5.5.4.2 show how the proposed formalism is a generalization of the traditional locking approaches. Initial evaluation of property-based locking is discussed in Section 5.5.6. Finally, Section 5.6 concludes the chapter by by a summary and state the contributions achieved in Conflict Reduction and Handling.

5.2 Preliminaries

5.2.1 From Model Comparison To Model Merge

Model comparison refers to identifying the differences between models. It requires reliability, precision and completeness as the merge process frequently relies on the output of this phase to detect conflicts and to resolve the detected conflicts. Altmanninger et al.[ASW09] classifies model comparison methods based on the kind of information available. Only models are provided as input for *state-based* techniques, while *change-based* comparison relies on a list of the performed changes, e.g. op_1, op_2, $\ldots op_n$. In the dissertation, only *state-based* techniques are investigated.

Definition 5.1 (Model Comparison). A function $compare :: M \times M \rightarrow Ch$ takes two models as input (M_1, M_2) and returns a set of changes $compare(M_1, M_2) = Ch$ where a change $change = \langle d, f \rangle$ is tuple formed as follows:

- d is direction from the set of $\{+, -\}$,
- f is a model element of $O_1 \cup O_2 \cup L_1 \cup L_2 \cup S_1 \cup S_2$.

To construct M_2 from M_1, all changes of direction $+$ need to be added to M_1 and all changes of direction $-$ need to be removed from M_1. ▪

Based on the results of model comparison, *model merge* synthesizes a combined model which reconciles the identified differences. This is not always possible due to conflicts between model changes carried out by different collaborators. A merged model is called *syntactically correct* if it corresponds to its metamodel, and *consistent* when additional constraints of the domain are satisfied.

Definition 5.2 (Three-way Model Merge). A function $merge :: M \times Ch \times Ch \rightarrow M$ synthesizes a merged model M_M from models M_O,M_L,M_R, formally

$$merge(M_O, Ch_L, Ch_R) = M_M,$$
$$compare(M_O, M_L) = ChL, compare(M_O, M_R) = ChR$$

5.2.2 Design Space Exploration

Design space exploration (DSE) aims to find optimal *solutions* from the several *design candidates* which satisfy several structural and numeric constraints, and they are reachable from an initial model along a trajectory by applying a sequence of exploration rules. The input of a rule-based DSE[HHV15a] includes (1) the *initial model* used as the start of the exploration; (2) *goals* which need to be satisfied by solutions; (3) the set of *exploration rules*; (4) *constraints* that need to be respected in each exploration state and (5) further *guidance* for the exploration process.

Definition 5.3 (DSE Problem). A *DSE* problem $DSE = \langle M_0, Rules, Goals, Consts, guidance \rangle$ is defined by a tuple as follows:

- M_0 is the *initial model*,
- $Rules$ are a set of *transformation rules*
- $Goals$ are a set of *goals* defined by graph patterns,
- $Consts$ are a set of constraints *constraints* defined by graph patterns,
- $guidence :: Activations \rightarrow A$ selects the next activation from activations. ▪

Definition 5.4 (DSE Solution). A solution $Sol = \{ES\}$ of a $DSE = \langle M_0, Rules, Goals, Consts, guidance\rangle$ is a set of execution sequences ES where

- execution sequence is an ordered set of execution steps denoted as $ES = M_0 \overset{A_0}{\rightarrowtail} M_1 \overset{A_1}{\rightarrowtail} M_2 \dots M_s$
- solution consists of an execution sequences $ES = M_0 \overset{A_0}{\rightarrowtail} M_1 \overset{A_1}{\rightarrowtail} M_2 \dots M_s$ where all goals are satisfied in M_s and none of the constraints are violated in $M_{i \in \{0..s\}}$. ▪

5.2.3 Locking

Locking is a well-known conflict prevention technique where users can request that certain engineering artifacts should be made unmodifiable by all other participants for a duration of time. The goal of locking is to make sure that no concurrent modifications will interfere with activities carried out within the scope of the lock.

In the following, we review the two main approaches in the state of the art of locking in MDE:

Fragment-based (FB) locking[SVN; CS14; Cha14; EMFStore; Kra+06] requires that models are partitioned into storage fragments, e.g. files or projects; in the extreme case the entire model is a single fragment. Entire fragments can be locked at once, including all contained model elements and their features. This solution is often provided by generic file-based collaboration solutions for source code development (e.g. SVN[SVN] or Git[CS14] extended by Gitolite[Cha14]). However, such fragments are inflexible: restructuring an existing model into a different set of fragments may be difficult, thus we may assume that the fragmentation is essentially fixed. Unfortunately, if fragments are too large, then locking a fragment prevents concurrent activities that would otherwise be possible to carry out at the same time. On the other hand, the model persistence or collaboration framework might not support arbitrarily fine-grained storage fragments (e.g. cyclic references between files or projects are frequently disallowed), and a large model blown up into many small files or projects would also be difficult to manage.

Object-based (OB) locking[CDO; ECM11] solves these problems by locking individual model objects (including their attributes and connections). This requires the collaboration framework to be aware of the structure of the model. The object-based approach is more fine-grained than the fragment-based solution, but individual attributes or connections of model elements are still not independently lockable. Furthermore, locking several objects will prevent any modification to them, even if the participant requesting the lock only needs some property of the model to be preserved while carrying out his activities.

While these strategies have been adopted in several collaborative modeling tools, they may limit the degree of concurrent development; and as we demonstrate in this chapter, they do not scale well with the increasing number of collaborators.

5.3 Related work

5.3.1 Model merging

Several approaches address the model merge as depicted in Table 5.1. To position them against our approach, we use several characteristics proposed in a survey on model versioning [ASW09], which also guides the structure of this section.

	Basis	Conflict Detection	Merge Automation	Merge Operations	Objectives	Guidance	Evaluation
EMF Compare[EMF-Comp]	state	static	semi	generic	-	-	scalability
EMF Diff/Merge[EMF-Diff]	state	static	semi	generic	-	-	scalability
Westfechtel[Wes14]	state	runtime	semi	generic	goals	-	preliminary
N-way Merge[RC13]	state	static	semi	generic	-	-	preliminary
AMOR[Bro+09]	state	static	semi	generic, composite	goals	-	precision recall
Dam H.K. et al.[DRE14]	state	static	auto	composite	goals, constraints	repair plan	scalability (closed)
MOMM[Man+15]	operation	runtime	auto	composite	fixed goals	global search prioritized	real data
DSE Merge	state	runtime	auto	generic, composite	goals contraints	local search may/must	scalability (open)

Table 5.1: Comparison of model merge approaches

Comparison Basis. Based on the model comparison technique, the approaches may be classified into *state-based* and *operation-based*. [EMF-Comp; EMF-Diff; RC13; Bro+09; Wes14; DRE14] and DSE Merge are *state-based* as they execute a comparison process between model states. However, [Man+15] uses operations as input where even more complex operations as just the simple add, update, and delete operations are considered.

Conflict Detection. Finding the conflicting changes in the merge process is crucial task for a correct resolution. Most approaches use an initial phase to statically analyze the changes and look for conflicting pairs such as in [EMF-Comp; EMF-Diff; RC13; Bro+09; DRE14]. [Wes14] defines transformation rules for searching conflicts where the satisfied preconditions selects the conflicts in each iteration. [Man+15] uses conflict detection algorithm between operations [Bro+12]. DSE Merge identifies conflicts incrementally as violations of constraints or as deactivations of merge operations, while dependencies between rules and constraints are handled automatically by the underlying DSE engine. This extends [DRE14] where inconsistency constraints are handled incrementally while conflict detection happens as preprocessing.

Merge Automation. Most approaches [EMF-Comp; EMF-Diff; Wes14; RC13; Bro+09] are semi-automated as they use a two-phase process: (i) they apply the non-conflicting operations and then (ii) let the user prioritize and select the operation to apply in case of two conflicting changes. This always results in a single solution due to the manual resolution by the user. In comparison, [DRE14; Man+15] and DSE Merge resolve the conflicts automatically in different ways and offer several solutions.

Merge operations. In this context, merge operations are responsible for applying the changes in the merged model. [EMF-Comp; EMF-Diff; RC13; Wes14] use generic operations for changes. The extension [Bro+11] of [Wes14] adaptively learns resolution patterns from user that can be applied on the models which results in composite operations. [Man+15] applies the input operations which are composite refactorings in their case. [DRE14] uses basic generic operators for conflicts but generates composite operations as repair plans from the description of inconsistency constraints. Our DSE Merge approach allows to combine both generic and domain-specific composite operators in the form of change-driven transformation rules.

Objectives. Quality of the merge model can be improved by objectives that have to be satisfied during (*contraints*) or at the end (*goals*) of the merge process. This is an unsupported feature in [EMF-Comp; EMF-Diff; RC13]. [Man+15] uses two fixed goals which are the base of the conflict resolution. [DRE14] provides support for incrementally detecting violations of inconsistency constraints.

[Bro+09] is connected to an additional model checker component [Bro+11] which allows to check OCL constraints as goals. [Wes14] allows to define well-formedness constraints in OCL that act as goals. DSE Merge let the users to provide additional constraints and goals using graph patterns in addition to a built-in termination condition when no operations are activated.

Guidance. The execution of the merge process can use guidance to find the solution(s) faster. The tool [SUW13] of [Wes14] uses a dedicated fusing algorithm for the model merge phase using a fixed priority strategy of merge operations. [Man+15] bases their tool to a global search genetic algorithm (NSGA-II[Deb+02]) where the operations are also prioritized related to their importance. DSE Merge is built on top of the ViatraDSE framework [HHV15a] using rule-based guided local search exploration. Furthermore, annotating changes with *may*/*must* can further reduce the result set retrieved to the user, which is another key difference wrt [Man+15; DRE14].

Evaluation. [Man+15] provides an empirical evaluation of the tool based on real data, but its scalability is not discussed as their largest model was the same as our smallest. [DRE14] represents an scalability evaluation of its tool with the largest size of 33.000 model element and $1,650$ changes. [RC13] and [SUW13] show a preliminary evaluation which show the relevance of the approach on very small models and change set. [Bro+09] evaluated by [LW13], but scalability is not discussed. For comparing models, [EMF-Comp] has a scalability test presented in [Bar]. DSE Merge is evaluated on an open scalability benchmark [Ujh+15]. As future work, we plan to create an empirical user study from the usability aspect of our tool.

Summary. To summarize the key differences with [DRE14] and [Man+15], we rely on state-based comparison, apply a guided local-search strategy (vs. [Man+15]), detect conflicts at runtime and allow complex generic merge operations (vs. [DRE14]). Internally, we uniquely use incremental and change-driven transformations to derive the merged models. Finally, we report scalability of merge process for models which are at least one order of magnitude larger compared to [DRE14] and [Man+15].

5.3.2 Locking

Approach	Locking mechanism	Type (implicit/explicit)
[Kra+06][Alt+08][MTF][EMFStore]	file-based	explicit
[CDO][Tol16][ECM11]	object-based	implicit and explicit
[Gen][Pin03][Gal+11][Mar+14][Syr+13]	-	-
Our solution	properties	implicit

Table 5.2: Comparison of modeling tools

Modeling Environment. Existing collaborative modeling tools either lack of locking support or implement rigid strategies such as file-based locking, or locking subtrees or elements of a specific type, which hinder effective collaboration.

Most of *offline collaborative modeling tools*, e.g., ModelCVS [Kra+06], AMOR [Alt+08], Eclipse Modeling Team Framework [MTF] or EMFStore [EMFStore], rely on traditional version control systems using file-based locking with contributors committing large deltas of work.

Model repositories such as CDO[CDO], MetaEdit+[Tol16] and Morsa [ECM11], support both implicit and explicit locking of subtrees and sets of elements. These locks can prevent others from modifying elements to avoid conflicts.

Online collaborative modelling frameworks such as GenMyModel[Gen], CoolModes[Pin03], SpacEclipse[Gal+11], WebGME [Mar+14] and ATOMPM [Syr+13], rely on a short transaction model: a single, shared instance of the model is concurrently edited by multiple users, with all changes propagated to all participants instantaneously. These approaches use timestamped operations to resolve conflicts or provide only lightweight lock mechanisms, e.g., explicit locks to certain elements.

Our property-based approach is general and can be used for both implicit locking of subtrees and set of elements or explicit locking of a certain element and its incoming and outgoing references. In addition it extends these lock types with the definition of properties to provide less restrictive locking for the collaborators as highlighted in our experimental evaluation.

Computer-supported Co-operative Work. The most general area of collaboration is the field of computer-supported co-operative work (CSCW) tools that can help people working together including conference calls, screen shares, remote desktops etc. to collaboratively develop artifacts. Beyond implicit and explicit locking several other approach are exist to manage and prevent conflicts in concurrent collaboration.

Timestamped-based operation can be used to order the incoming operation on the server-side[Rek93]. When an operation arrives with earlier timestamp than the latest one due to a delay on the network, it will be rejected, even though the delayed change may not conflicting with the others.

Paul Dourish' pioneering work [Dou96] argues against the inflexibility of locking mechanisms based on the syntax of a collaborative artifact (here, a model). His proposed *Prospero* platform employs the promise-guarantee paradigm, where a user makes a promise concerning the purpose of its changes (the expected behavior, or usage pattern), and the collaborative editing system guarantees consistency of the model, provided that such promise is upheld.

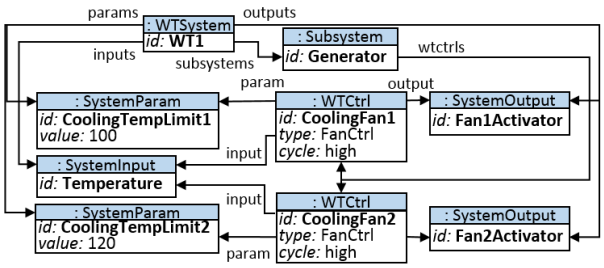
As it is stated in [*c*16], the concept of property-based locking is inspired by the Dourish' framework and aims to adapt it to the field of software/system modeling, where the collaborative artifact is a graph.

Databases. Databases detect *write/write* and *read/write* conflicts, where the former defines modifying the same record concurrently, while the latter is about reading dirty records.

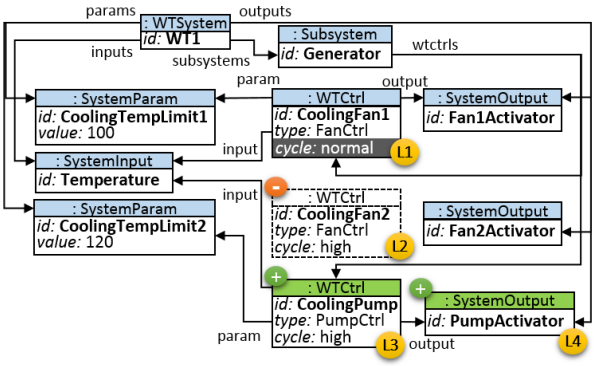
Both relational[Oracle; MS-SQL] and graph-based[Neo4J; Orient; Stardog] databases use transactions to provide atomicity and ordered execution. Thus, these systems usually requires explicit locks before the execution of a transaction.

In database terms, locks can be obtained with a *pessimistic* or *optimistic* strategy. Pessimistic lock prevents the initiation of any modification on the locked records. Optimistic lock allows to execute a transaction, but during the execution it can be aborted when a record became dirty (updated by someone else) and it needs to be rerun.

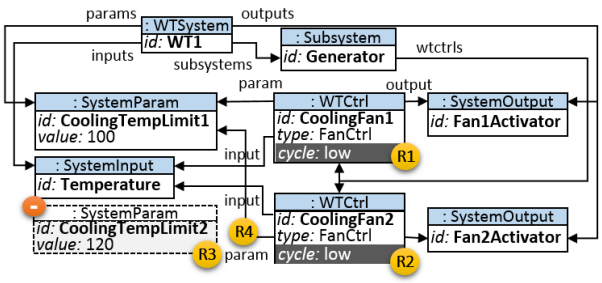
Our property-based solution is an *optimistic* locking strategy to prevent *write/write* conflicts, where the collaborators can introduce any kind of changes until they violate a lock. The approach cannot handle *read/write* as we assume they are handled by the underlying collaboration frameworks (e.g. [SVN], [EMFStore]) due to the atomic transactional executions of changes.



(a) Original model



(b) Local instance (modified by User1)



(c) Remote instance (modified by User2)

Figure 5.1: Local and remote changes for 3-way merge

5.4 Automated Model Merge by Design Space Exploration

5.4.1 A Motivating Model Merge Scenario

The domain of our motivating example describes a more detailed version of *Wind Turbine Control Systems (WTCS)* developed by *IK4-Ikerlan* where different artifacts and algorithms for controlling a wind turbine are specified and connected to sensors and actuators. Models are specified by several collaborators, and consequently modifications could result in merge conflicts.

We introduce a simplified example of a wind turbine (WT1) in Figure 5.1. Real models are obviously larger, sample models of this example contain only artifacts related to the cooling of the `Generator Subsystem`:

- *Inputs*: Wind turbine `WT1` gets data from a temperature sensor specified by the `SystemInput` identified as `Temperature`.
- *Outputs*: `WT1` acts on two fans for cooling the wind turbine generator specified by the `SystemOutputs`: `Fan1Activator` and `Fan2Activator`.
- *Params*: temperature limits for starting generator cooling can be specified

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 with `High` priority cycle with `Temperature` as `SystemInput`, `Fan1Activator` as `SystemOutput`, `CoolingTempLimit1` as `SystemParam`.
- `CoolingFan2`: this control unit (of type `FanCtrl`) specifies the control algorithm for fan #2 with `High` priority cycle with `Temperature` as `SystemInput`, `Fan2Activator` as `SystemOutput` and `CoolingTempLimit2` as `SystemParam`.

Table 5.3: Elements of $\Delta(L)$ and $\Delta(R)$

Δ(L,O)comparison model	Δ(R,O)comparison model
MAY attribute{CoolingFan1,cycle,Normal}	MAY attribute{CoolingFan1,cycle,Low}
MUST delete{CoolingFan2}	MUST attribute{CoolingFan2,cycle,Low}
MUST create{CoolingPump,WTCtrl,WT1,ctrls}	MAY delete{CoolingTempLimit2}
MAY attribute{CoolingPump,type,PumpCtrl}	MAY reference{CoolingFan2,param,
MAY attribute{CoolingPump,cycle,High}	CoolingTempLimit1}
MAY reference{CoolingPump,param,CoolingTempLimit2}	
MUST create{PumpActivator,SystemOutput,WT1,outputs}	
MAY reference{CoolingPump,output,PumpActivator}	

As a running example, we investigate the following scenario:

Local changes. The first expert creates a Local version of the model with the following changes: (L1) the `cycle` attribute of `CoolingFan1` is changed to `Normal`, (L2) `CoolingFan2` instance is deleted. (L3) A new control unit (`WTCtrl`) is created with `CoolingPump` id. The new control unit is of type `PumpCtrl` with `High` cycle. Its input references the existing `Temperature` and its `param` references the existing `CoolingTempLimit2`. In contrast, (L4) its output references a new `SystemOutput` instance identified as `PumpActivator`.

Remote changes. Another expert also remotely modified and already committed the model (before the first expert working on the local version managed to commit the model) to introduce the following remote changes: (R1) the cycle attribute of `CoolingFan1` is changed to `Low`, (R2) the cycle attribute of `CoolingFan2` is changed to `Low`, (R3) deletes `SystemParam` instance identified as `CoolingTempLimit2` and (R4) changes `param` reference of control unit identified as `CoolingFan2` to `SystemParam` instance identified as `CoolingTempLimit1`.

Model comparison. Table 5.3 shows the result of model comparison between the different versions of the model calculated by using existing tools (using e.g. EMF Compare or Diff/Merge [EMF-Diff]). The differences between the local and the original model is denoted with $\Delta(L, O)$ (or shortly ΔL), while $\Delta(R, O)$ (or ΔR) represents the differences between the remote and the original model.

Change annotation. After the comparison, the local collaborator annotates local changes $L2$, $L3$ and $L4$ and remote change $R2$ as *must* which prescribes that all such changes have to be present in the merged model unless some of them are in a conflict. In such a case, the merged model should contain as many (non-conflicting) *must* changes as possible, while some (conflicting) *must* changes might be omitted from the merged model. All other changes are marked as *may* to denote that the corresponding change may be included in the merged model.

Challenges. The following challenges need to be addressed for our example:

- Calculate *merged models* automatically as a maximal subset of non-conflicting changes from the local and remote change set. When there is a large number of possible combination of changes where some of them are selected from the local and the others from the remote branch, a merged model may be restricted to solutions compliant with *must* and *may* change annotations.
- Use *domain-specific goals and constraints* to restrict merged models to consistent ones (to ensure that all inputs and parameters are referenced by at least one control unit and each output is referenced by different control unit).
- Specify *domain-specific composite operations* to guide the merge process into a consistent solution (e.g. to remove inputs, parameters and outputs not referenced by any control unit).

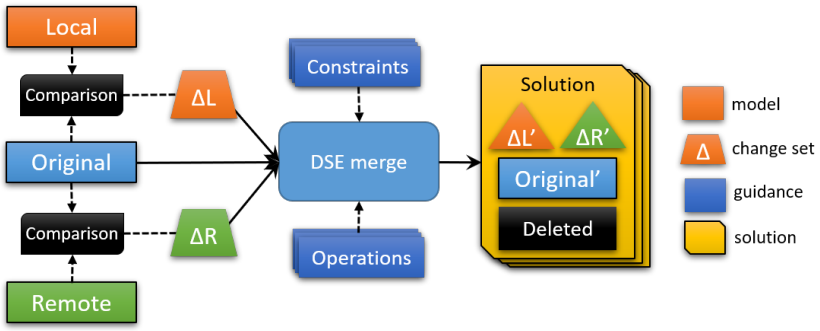


Figure 5.2: Architecture of DSE Merge

5.4.2 Conceptual Overview

To achieve **MG1**, we propose to exploit guided rule-based *design space exploration (DSE)* [**hegedus2011model**] for automated model merge with an architecture depicted in Figure 5.2. Rule-based DSE aims at finding optimal *solutions* from the several *design candidates* which satisfy several structural and numeric constraints, and they are reachable from an initial model along a trajectory by applying a sequence of exploration rules. The input of a rule-based DSE includes (1) the *initial model* used as the start of the exploration; (2) *goals* which need to be satisfied by solutions; (3) the set of *exploration rules*; (4) *constraints* that need to be respected in each exploration state and (5) further *guidance* for the exploration process.

We applied three-way model merge to a DSE problem as follows:

1) the **initial model** contains the *original* model O and two *difference models* (ΔL and ΔR)
2) the main **goal** is that there are no executable changes left in ΔL and ΔR along a specific exploration path.
3) the **operations** are defined by change driven transformation rules to process generic change objects (*create*, *delete*, *set*, *add*, *remove*) of the difference models, and potentially composite (domain-specific) operators;
4) **constraints** may identify inconsistencies and conflicts to eliminate certain trajectories;
5) as main **exploration strategy**, any changes annotated as *must* are tried to be merged before resolving *may* changes.

Input. Our model merge approach takes three models as input: the original model O and the *difference models* between local and original models ΔL as well as the remote and original models ΔR. These together constitute the initial model for DSE. The calculation of the difference models ΔL and ΔR is carried out by an external comparison tool such as EMF Compare or Diff/Merge. Furthermore, in order to derive efficient state encoding for the exploration process, we assume that each element in the original model has some unique identifier.

Output. The output of the merge process automatically derived by DSE is a *set of solutions* where each solution consists of (i) the merged model M derived by applying a (non-extensible and non-conflicting) subset of local and remote changes on the original model O; (ii) the set of non-executed changes $\Delta L'$,$\Delta R'$; and (iii) the collection of the $deleted$ objects.

5.4.3 Key Aspects of Exploration Process

Each solution is derived along a trajectory from the initial state to a solution state by *applying generic and domain-specific operations*. Along this trajectory, we transform the original model O into the

merged model M, and the change models ΔL and ΔR are gradually reduced to $\Delta L'$ and $\Delta R'$. In each exploration step, *conflicts are detected and resolved* by incrementally tracking the matches (activations) of operations and constraints. Finally, a solution state is identified if all *goals* are satisfied without violating a *constraint* along the trajectory.

Operations. We incorporate two kinds of operations in the exploration based model merge: *generic merge operations* [Wes14] and (domain-specific) *composite operations*[Man+15; DRE14] (such as refactorings, or repair rules) to achieve **MG2**. Each operation is captured by (graph) transformation rules (see Def. 2.12), which consist of a *precondition* described as a graph pattern and an *action* part which captures model manipulations.

Generic merge operations are *change-driven transformations* [Ber+12], which consume or produce change models as additional input or output. The precondition selects an applicable change c from the deltas $\Delta L \cup \Delta R$ and may require the existence of certain model elements in the origin model O. The action part of a generic merge operation (1) modifies the original model O to apply a change, (2) moves the change c from the difference set $\Delta L \cup \Delta R$ into a completed set $Comp$ to prevent the application of the change multiple times. Thus such change-driven rules transform state-based merging into operation-based merging [Bro+12].

By default, domain-specific *composite operations* only manipulate the model O without consuming the deltas. Therefore, they need to be complemented with generic change-driven rules which identify the model-level changes carried out by them and record them as difference models in the completed set. In most cases, domain experts are responsible for capturing complex (domain-specific) operations only at the preparation of the merge tool for the specific domain. Collaborating engineers only use them as part of the merge process.

Conflict detection and resolution. A local change $l \in \Delta L$ and a remote change $r \in \Delta R$ may be conflicting, i.e. it is impossible to obtain a consistent merged model M by applying both l and r. Alternatively, in an operation-based interpretation, a conflict denotes a pair of operations o_1 and o_2, whereas one operation masks the effect of the other (i.e., they do not commute) or one operation disables the applicability of the other [Man+15].

Instead of static (a priori) detection of conflicts as proposed in [Men02; Ste+96; Fea89], we detect conflicts *on-the-fly* during the exploration process by relying upon the *incremental book-keeping of rule activations and constraints*. In each state of the DSE, we investigate one by one all (enabled) activations of transformation rules, and try to find a solution by firing them. In case of a conflict, (1) firing one rule may prevent the application of another activation, or (2) both rules are fireable, but the result state violates a constraint. When two operations are confluent (i.e. they can be applied in arbitrary order), state encoding of DSE [HHV15a] helps identify that an already traversed state is reached. Hence applying operations in a different order has no impact on the results.

Activations of rules and constraints are continuously and efficiently maintained when firing an operation (either generic or composite), thus disabled rules and violated constraints are immediately identified. For that purpose, we rely upon the reactive VIATRA framework [Ber+15b] and incremental model queries. The technicalities of conflict detection will be illustrated in Section 5.4.4.

Conflict resolution by exploration strategy. In case of a conflict between two operations, DSE will investigate both trajectories as possible resolutions and derive two separate solutions correspondingly. Thus a merged model M derived automatically as a solution contains no conflicts by definition.

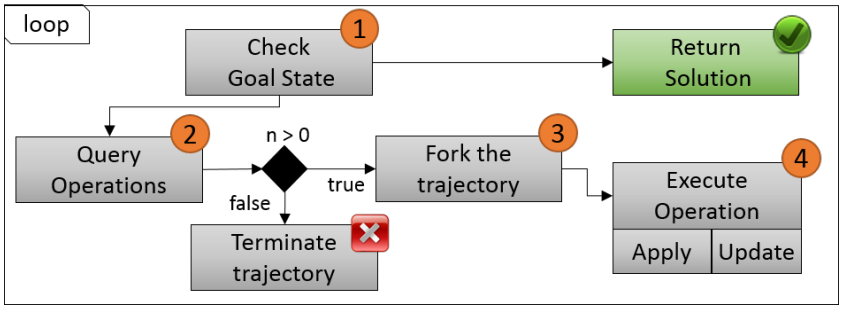


Figure 5.3: Execution loop of the merge process

In case of many conflicts, the result set can too large to be presented to experts. Therefore, in order to reduce the number of solutions retrieved by DSE and guide the exploration in case of conflicts, model changes can be prioritized by the collaborators as *may* and *must* (see Table 5.3) prior to executing merge.

- If a change c_1 with *must* priority is in conflict with another change c_2 of *may* priority, then the merge will always select the former (c_1).
- If two conflicting changes c_1 and c_2 are both annotated with *may* than the merge will randomly select one.
- However, if two changes c_1 and c_2 of *must* priority are in conflict, then the merge process will enumerate both of them separately (in different solutions).

Goals. In generic, we aim to apply as many changes in ΔL and ΔR as possible to derive the merged model M. When extending a trajectory by any of the remaining changes in $\Delta L'$ or $\Delta R'$ would cause a conflict with some already applied change, a solution state of the DSE is reached. Technically, it is detected by the termination of the rule system, i.e. no operations are activated. Additionally, domain experts can provide domain-specific goals that act as heuristics for the exploration and provide consistent solutions.

Altogether, we define a *fully automated model merge approach* where all possible resolutions of conflicts are calculated, and all consistent merged models are prompted to experts, which was claimed to be beneficial in [Wie+13]. Representation of solutions contains several layouts (e.g. tree, graph) and metrics (e.g. number of executed changes) which help experts select the best solution for their purpose.

The high level execution algorithm of the model merge by DSE is described in Alg. algorithm 4. The `traverse` procedure is responsible for searching different design candidates and storing them in a (global) variable `solutions`. Inputs of the procedure are the current state of the `model`, current set the `changes` and the `trajectory` of already executed operations. Initially, `model` is the original model, `changes` are $\Delta(L) \cup \Delta(R)$, while `trajectory` is empty, while the result will be stored in the `solutions` set.

1. *Check for solution*: First, the algorithm checks (Line 3) if a solution is found or not. In the context of model merge, this means that (a) no additional change in *changes* is executable on *model* (without introducing a conflict) and (b) all domain specific constraints are also satisfied (i.e. the corresponding goal patterns do not have a match). If a solution is found, the `trajectory` is put into `solutions` and the process backtracks (Line 4).
2. *Query operations*: Otherwise, the algorithm queries and calculates the executable `operations` (Line 6). An operation is executable if its precondition is satisfied by the current version of the model, which is efficiently tracked by incremental graph query techniques. Furthermore,

Algorithm 4 Merge Strategy

```
1:  solutions ← Set()
2:  procedure TRAVERSE(model, changes, trajectory)
3:      if CHECKIFSOLUTION(model, changes) then
4:          return PUT(solutions, trajectory)
5:      else
6:          operations ← QUERYOPERATIONS(trajectory)
7:          for o ∈ operations do
8:              model' ←APPLY(COPY(model), o)
9:              changes' ←REMOVE(COPY(changes), o)
10:             trajectory' ←ADD(COPY(trajectory), o)
11:             if not ALREADYTRAVERSED(model') then
12:                 TRAVERSE(model',changes',trajectory')
13:             end if
14:         end for
15:     end if
16: end procedure
```

priority is given to operations processing `must` changes, i.e. operations for `may` changes are only retrieved if no further `must` changes can be handled.

3. *Execute operations*: Each operation (`o`) is then applied on a copy of current model (`model'`) in Line 8, and then we remove the change imposed by operation `o` from (a copy of) the current set of changes (`changes'`) in Line 9. Then the operation is appended to (a fresh copy of) the current trajectory (`trajectory'`) in Line 10.
4. *Check if already traversed*: We have to guarantee that each solution is unique, i.e. two solutions yielding the same merged model by applying operations in a different order are treated identical. Therefore, each traversed state is stored globally (using a hash function) and the algorithm finally checks if new state identified by `model'` is not yet traversed (Line 11), and calls itself recursively with the updated model, changes and trajectory variables (Line 12).

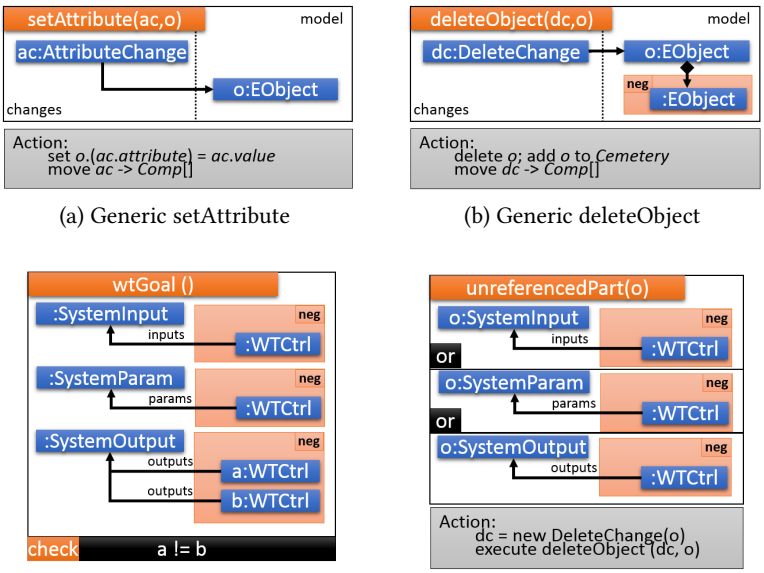
Note that this algorithm represents our basic approach, while the real implementation uses a more sophisticated exploration strategy based on the VIATRA-DSE [HHV15a] engine.

First of all, 1) the strategy checks whether the current model satisfies all goal constraints or not. In the former case, the `trajectory` is put into `solutions` and the process backtracks. Otherwise (2) the procedure queries the available `operations`. 3) Each operation (`o`) is applied on a copy of current model (`model'`), removed from the copy of current set of changes (`changes'`) and also added to a copy of current trajectory (`trajectory'`). Next, 4) it checks that this state is not yet traversed and calls itself recursively with the modified model, changes and trajectory variables. The output of our algorithm is the `solutions` set.

5.4.4 Elaboration of Model Merge on an Example

5.4.4.1 Operations and goals

Change-driven rules for generic operations. We defined the following generic operations in the merge process for *creating/deleting object*, *setting/adding/removing attribute* and *setting/adding/re-*



(a) Generic setAttribute

(b) Generic deleteObject

(c) Domain-specific goal

(d) Domain-specific operation

Figure 5.4: Operations and goal

moving reference. For space considerations, we only discuss operations for setting an attribute (*setAttribute*) and deleting an object (*deleteObject*) in details (depicted in Figure 5.4).

- *setAttribute(ac,o)*: The precondition prescribes that an attribute change ac is available in change set $\Delta L' \cup \Delta R'$ and its object o exists in the current model. Its action sets (i) attribute $ac.attribute$ of object o to the given value $ac.value$, and (ii) moves the change ac to the completed set $Comp$.
- *deleteObject(dc,o)*: The precondition states that a delete change dc is available in the current change set $\Delta L' \cup \Delta R'$ and its referred object o exists in the current state of the model where o is a leaf in the containment hierarchy. The action part (i) deletes the object o from current state, (ii) puts it into the deleted set $Deleted$ and (iii) moves the change dc to the completed set $Comp$.

Domain-specific goals and operations. Our example introduced in Section 5.4.1 requires to extend model merge with domain-specific knowledge to guarantee the consistency of solutions. In the *Wind Turbine Control System (WTCS)* domain, it is mandatory that all `SystemInput` and `SystemParam` instances should be referenced by at least one control unit and each `SystemOutput` has to be referenced by a unique control unit. Model merge needs to respect such domain specific knowledge, which can be captured by additional goals specified as constraints and depicted in a graphical representation in 5.4(c).

A domain-specific operation called *unreferencedPart* can be defined to eliminate unreferenced `SystemInput`, `SystemOutput` and `SystemParam` instances (see 5.4(d)). Here the precondition selects the unreferenced object o while the action part (i) initiates a new *delete change* independently from the current change set and (ii) executes the action part of the generic *delete* operation.

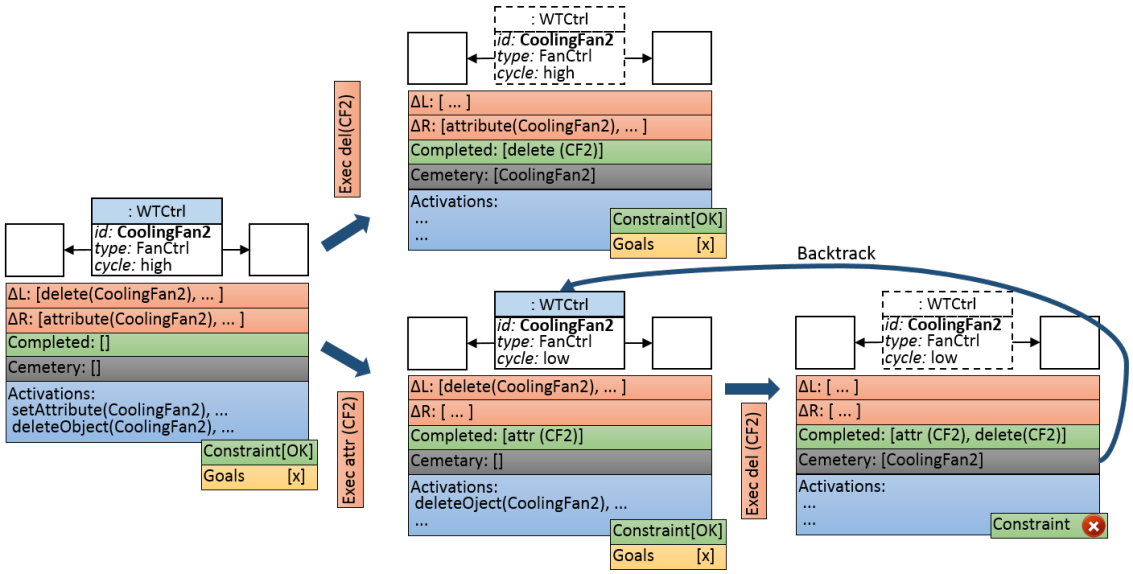


Figure 5.5: Conflict resolution with incrementally tracking constraints and operations

5.4.4.2 Conflict detection in a sample exploration step

Conflict detection and resolution is carried out during exploration by incrementally tracking rule activations and special constraints. We illustrate this step in the context of our running example (see Figure 5.5, which is an extract of *iteration* 3 *and* 9 of merge session from subsubsection 5.4.4.3). It demostrates a delete/use conflict: simultaneously setting the cycle attribute of `CoolingFan2` and deleting `CoolingFan2`. Any solution of model merge may only contain one of the two changes.

1. In the beginning, both operations have an activation (left in Figure 5.5) in the context of object `CoolingFan2`. Initially, all changes are located in ΔL or ΔR, *deleted* set and completed changes are empty. In this state, all constraints are satisfied, but goals are violated which means this state is not a solution.
2. Our merge process first selects and executes the `deleteObject` operation (top branch of Figure 5.5) which removes `CoolingFan2` from the model, moves `CoolingFan2` to the *deleted* set, and the corresponding change is moved from ΔL to the completed set $Comp$. As a side effect, operation `setAttribute` loses its activation in the context of `CoolingFan2` since its precondition is no longer be satisfied in the new state. This fact is immediately identified by the underlying reactive transformation engine [Ber+15a]. In the new state, the exploration incrementally checks that all constraints are satisfied and goals are violated, and then selects another enabled (activated) operation for execution.
3. Later, after backtracking to the first state, operation `setAttribute` is scheduled for execution on object `CoolingFan2` (bottom branch of Figure 5.5). As a result, *deleted* set remains empty, the change is moved to the completed set, all goals are violated, and all constraints are satisfied. As a main conceptual difference, the activation of `deleteObject` is not disabled on `CoolingFan2` as the corresponding object still exists, hence its precondition is satisfied.
4. Next, the process selects and executes `deleteObject` operation. As a result, `CoolingFan2` is moved to the *deleted* set and the change is moved from ΔR to the completed set $Comp$. We detect this conflict by (incrementally) checking a generic merge constraint: there are two changes in the

completed-set $Comp$ which modifies the same object. In this case, exploration has to backtrack and finds another executable operation.

Obviously, the first type of constraint could also be detected by using similar constraints as for the second type. However, lost activations reduce the number of states to be traversed, thus they are preferred. Furthermore, note that when two operations are applicable in both order with a confluent result, the state encoding of DSE identifies that the same model is reached as a state.

5.4.4.3 A merge scenario on the motivating example

A possible execution of the DSE Merge is depicted in Figure 5.6 which displays the completed changes for two solutions. In each iteration, one change is processed.

- Itr. 1-2: all *must* changes are available and the algorithm randomly picked the `createObject` of `CoolingPump` and `PumpActivator`.
- Itr. 3: at this point only two conflicting transitions have activation; the algorithm picked `deleteObject` for `CoolingFan2` non-deterministically. This leads to a state where the precondition of `setAttribute` operation cannot be satisfied any longer, thus it is disabled.
- Itr. 4-5: only *may* operations have activation where a `setAttribute` operation is selected that set the `cycle` attribute of `CoolingFan1` to `normal`. Because of the generic constraint, the other `setAttribute` related to the same object (`CoolingFan1`) is disabled. The same happens when executing `deleteObject` for `CoolingTempLimit2` that disables the `setReference` operation which should connect `CoolingPump` and `CoolingTempLimit2`.
- Itr. 6: this (aggregated) step is composed of all iterations that execution of operation `setAttribute` related to the newly created `CoolingPump`.
- Itr. 7: on this trajectory, deletion of `CoolingFan2` leads the model into a state where the `Fan2Activator` output is not referenced by any control unit. Thus our domain-specific (composite) operation (`unreferencedPart`) has an activation that is executed on the model. After this iteration, there are no more activations and all goals are satisfied, so *Solution #1* is found.
- Itr. 8: after the solution, the strategy backtracks until it finds an activation for a *must* operation that should lead the model into a partially traversed state and forks the trajectory. Only the `setAttribute` operation related to *CoolingFan2* can be executed. After the execution, *deleteObject* of `CoolingFan2` could have activation, but it is disabled by the generic constraint.
- Itr. 9-11: The same activations are available as for the 4th iteration except the domain-specific operation. The algorithm randomly executes these operations and finds *Solution #2*.

Resolved conflicts. In iteration 3 and 8, two conflicting operations marked with *must* are executed which forks the exploration into two separate solutions to resolve the conflicts. At iterations of 4 and 9, two operations with *may* mark are in conflict. In each trajectory, only one of them is selected. Similar happens in iteration 5 and 10, but this time the same operation is selected in each branch.

Solution. There are two solutions in the output of the merge process. We discuss solution #1 in details where the merged model is depicted in Figure 5.7. It also displays in dashed line the deleted objects, namely, `CoolingTempLimit2`, `CoolingFan2` and `Fan2Activator`. There are four non-executed changes as shown in the bottom left corner of Figure 5.7.

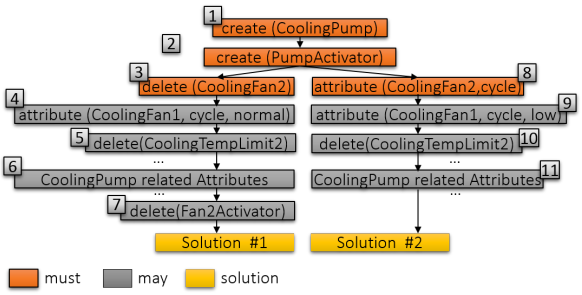


Figure 5.6: Possible execution of the process

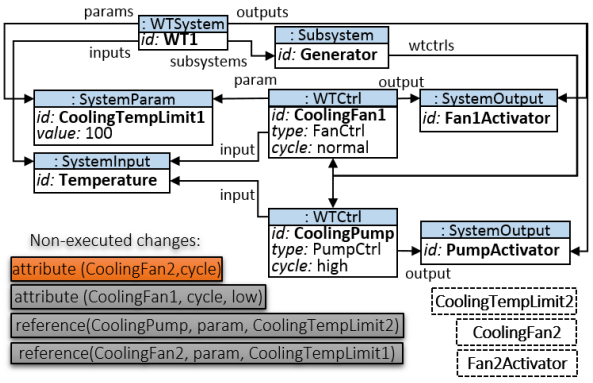


Figure 5.7: Merged Model from Solution #1

5.4.5 Scalability Benchmark of Model Merge and Evaluation

In order to evaluate DSE Merge, we have carried out an initial evaluation to demonstrate the scalability of our approach on the Train benchmark case study [Ujh+15] (an accepted model querying performance benchmark). We evaluate our DSE-based automated merge approach to assess its scalability where we investigate the scalability of the approach by measuring execution time for model comparison (carried out by EMF Compare) and model merge.

Our measurements aim to address the following questions:

Q1 *What is the influence of the size of models on the merge process?*
Q2 *What is the influence of the size of change set on the merge process?*
Q3 *What is the influence of the number of changes in conflict on the merge process?*

5.4.5.1 Scalability Benchmark

As the state-of-the-art of model merge still lacks well-accepted benchmarks to measure *scalability* of model merging components (e.g. [LW13] measures precision and recall), we propose a new scalability benchmark (**MG3**) for model merge by adapting of the Train Benchmark [Ujh+15], which is an existing performance benchmark for model queries and well-formedness constraints (and also a case of the TTC 2015 contest [Szá+15]). The benchmark uses a domain-specific model of a railway system originating from the MOGENTES project [Mog].

From the existing benchmark, we reuse (1) the *model generator* to derive models of different size conforming to a railway metamodel, (2) the *fault injector* which changes the generated model (e.g. by changing structural features, and creating or deleting objects) to violate predefined well-formedness constraints, and (3) *repair actions* which pseudo-randomly resolve such violations in accordance with to a random seed value.

Based upon these components, we summarize how synthetic models are generated that contain conflicts serving as input for model comparison and model merge: (1) First, we generate a well-formed model. (2) Next, we inject several faults into the generated model. The result of this phase acts as *original (O)* model. (3) Then, local and remote changes are simulated by repairing these violations either in the local model *(L)* or remote model *(R)* or in both of them with different random seeds. In the latter case, the framework repairs the same problems in both cases by using different values, which leads to a conflict between two models. (4) We calculate the differences between the two with

Size	Δ	Diff (sec)	Merge (sec) 0% conflict	50% conflict	100% conflict	Size	Δ	Diff (sec)	Merge (sec) 0% conflict	50% conflict	100% conflict
11710	120	4.672	1.265	2.095	3.477	87396	120	28.302	10.654	13.556	22.913
	240	7.329	2.241	3.345	4.109		240	30.711	20.285	24.377	37.501
	480	12.951	3.923	4.650	8.813		480	36.378	38.154	48.655	76.703
	960	23.323	8.853	12.008	21.842		960	49.382	75.567	92.797	153.234
	1920	26.368	11.352	19.766	29.948		1920	80.934	162.845	205.423	367.357
23180	120	7.233	2.686	2.924	6.262	175754	120	59.236	21.332	27.699	43.492
	240	7.569	4.355	5.106	8.596		240	79.068	42.308	50.843	79.492
	480	13.695	9.433	14.127	17.796		480	93.395	80.130	95.332	162.106
	960	23.383	18.219	22.474	40.589		960	97.313	157.720	185.030	279.367
	1920	41.857	34.181	57.207	96.806		1920	118.439	311.525	362.841	626.946
46728	120	17.258	6.679	8.156	12.567	354762	120	176.200	47.410	57.695	89.101
	240	18.592	10.625	12.623	20.047		240	177.280	84.678	104.739	166.990
	480	27.410	19.063	24.210	39.855		480	188.028	156.568	198.307	317.629
	960	40.915	37.961	51.924	90.295		960	209.440	307.878	406.879	636.156
	1920	69.344	165.203	180.534	217.343		1920	257.355	1,342.081	1,401.882	1,535.091

Table 5.4: Scalability measurement results

an existing comparison tool (EMF Compare). (5) Finally, these two model have to be merged with *may* annotations for changes using our merge tool.

5.4.5.2 Measurement Setup

For the evaluation, we generated models where the number of model elements is from $10,000$ to $350,000$, the number of faults injected into the models (i.e. size of the change set) is from 10 to 2000 while the number of conflicts are set to 0%, 50% and 100% of the total number of changes.

Analysis of results. To address **Q1**, **Q2** and **Q3**, measurement results are summarized in Table 5.4 taking the average of 5 separate runs.

As expected, merge time is linear in model and change size, and also proportional to comparison time. Furthermore, fewer conflicts imply faster merge time. Our results also show that runtime of merge is lower than compare time in case of smaller change sets (120,240), and gradually outgrows it as the change set increases. However, change sets of an average commit in real projects are even smaller than our smallest case (see also the evaluation in [Man+15]), which means that our scalability results represent a pessimistic setup.

5.4.5.3 Usability Evaluation

In [*c*4], we report how we systematically evaluated the efficiency of the DSE Merge technique from the user point of view using a experimental software engineering approach. The empirical tests included the involvement of the intended end users (i.e. engineers), namely undergraduate students, which were expected to confirm the impact of design decisions. In particular, we asked users to merge the different versions of the same model using DSE Merge when compared to using Diff Merge.

The main contributor of the experiment was Ankica Barisic, hence detailed descriptions and results are omitted from the thesis, however the experiment showed that to use DSE Merge participant required lower cognitive effort, and expressed their preference and satisfaction with it.

5.5 Property-based Locking

Once granted to a user, a *property-based lock* forbids all other users to modify the model in a way that would violate the given property of the model. In order to address **LG1** and capture exactly

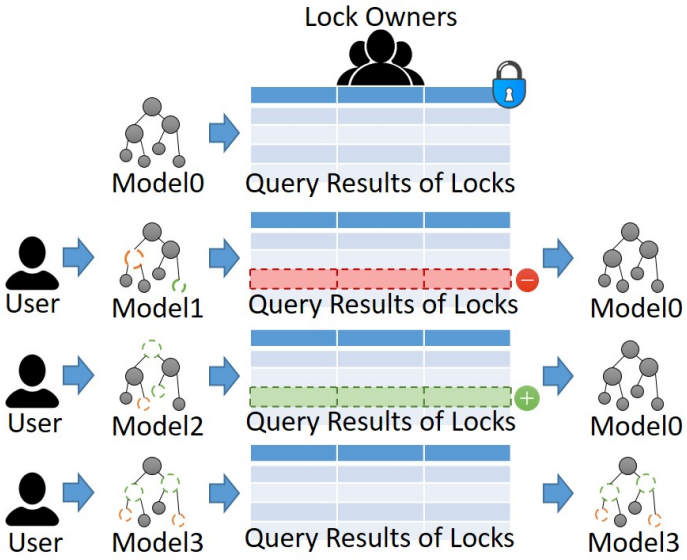


Figure 5.8: Behavior of Property-Based Locks (in case of invariant property)

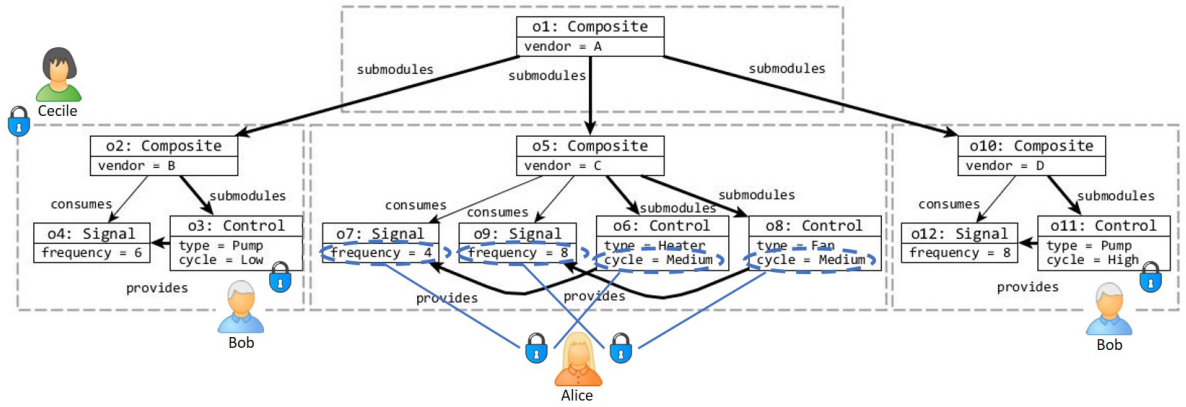


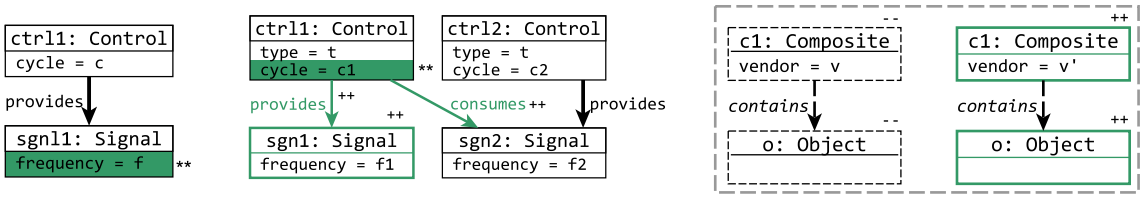
Figure 5.9: Sample Wind Turbine Instance Model (stored in separate fragments denoted by dashed border)

which changes violate the property, we adapt the concepts of a *change query* [Ber+12]. Each lock is associated with a *model query* that can be evaluated on different snapshots of the model. Only those modifications are allowed that do not change the result set of this query (in case of an *invariant property*), or change it only in a given *direction* (e.g. new matches may appear, or existing matches may disappear) as it is depicted in Figure 5.8.

5.5.1 A Motivating Locking Scenario

5.5.1.1 Operations

Several collaborators may work concurrently on wind turbine models to test and fine-tune them. Each collaborator attempts to a execute *maintenance operation (M)* to update certain signals, a *testing*



(a) Maintenance Operation **(M)**

(b) Testing Operation **(T)**

(c) Replacement Operation **(R)**

Figure 5.10: Sample Operations (++: creation, –: deletion, **: update)

operation (T) to add debug signals as outputs to certain control units and input signals to start their operation, or a *replacement operation (R)* to replace parts of specific vendors with new parts. (Note, that we omitted `protectedIP` attribute of composites as we assume access restrictions are handled correctly.)

Operations. Each operation is visualized in Figure 5.10 where `++`, `--` and `**` define creation of new objects, deletion of existing objects and update of certain attributes, respectively.

(M) changes the frequency f attribute of all signals contained by control units with a certain cycle c depicted in 5.10(a).

(T) creates new signals below control units of a certain type t, updates their cycle attribute from $c1$ and make the control unit to consume another signal provided by a control unit of the same type t depicted in 5.10(b).

(R) replaces each composite provided by a vendor v containing controls and signals transitively with the same structure but provided by another vendor v', see 5.10(c).

Example 17 Operation **M** can double the frequency of $o7$ and $o9$ if $c = medium$; operation **T** can create signals below $o3$ and $o11$, set their cycle speed to $high$ and create consumes references between the pairs of $\langle$o3,o12$\rangle$ and $\langle$o11,o4$\rangle$ if $t = pump$; while operation **R** can delete $o2$, $o3$, $o4$ if $v =$ "B" and replace them with new $o2'$, $o3'$, $o4'$ objects where the vendor of $o2'$ would be E if $v' =$ "E".

5.5.1.2 Usage Scenario

The wind turbine control model can be hosted on a collaboration server[CDO; EMFStore][c6] where it is stored, versioned etc. Users can connect to the model to modify the model by executing operations like the testing operations of the example.

To prevent the execution of conflicting operations, collaborators may lock certain properties of the model. If another collaborator attempts to execute an operation that violates a lock, the operation will be rejected.

Example 18 As an example, $Alice$, Bob and $Cecile$ are collaborators developing the example model. $Alice$ attempts to execute **M** operation with $c = medium$ and assigns 10 to the frequency of all selected signals ($o7$, $o8$) denoted by **M**($medium$, 10). Her interest is to prevent modification that would create or remove pairs of signals and controls where $c = medium$.

Bob tries to execute **T** operation with $t = pump$, changes the cycle attribute to $medium$,

creates testing signals under the controls and makes controls consume a signal denoted by **T(**$pump, medium$**)**. His interest is to lock the controls $o3$ and $o11$ to prevent any modification on these objects.

Finally, $Cecile$ replaces the components of $v = B$ to the components of provider E denoted by **R(**B, E**)**. She needs to prevent any kind of modifications in the fragment whose root is $o2$.

To prevent conflicts and preserve their intentions, they need to lock parts of the model of their interest as they are visualized with *lock icons* in Figure 5.9.

5.5.1.3 Effectiveness of Locking Strategies.

We discuss how FB and OB locks could be used in case of operations **T**, **M** and **R**. Unfortunately, traditional locking strategies will not be effective in all cases.

***Operation* T:** OB locks must be put on each existing control unit of type t, as all of them have some attributes and references modified when **T** is executed.

The FB approach has to lock all fragments containing control units of type t; this is an example of *overlocking*, as the rest of the objects (signals, composite modules) in those fragments can no longer be modified.

Both approaches suffer from *underlocking* (i.e. conflicting edits caused by too few locks) when collaborators can concurrently create new control units of type t (in different fragments in case of FB) which violates the atomicity of **T**, as the cycle and signals of these new elements will remain unadjusted.

***Operation* M:** OB has to put locks on all control units and signals for the given cycle value c. This is *overlocking* as nobody can modify these objects, their attributes and references, even though many of them (attribute *type* or reference *consumes*) are irrelevant from the viewpoint of **M**.

The *overlocking* is more severe when a FB lock is used: each fragment must be locked if it contains a control unit with cycle value c, and no other collaborator can modify these fragments at all (e.g. modifying objects, attributes or references that are irrelevant from the view of **M**).

Since other collaborators can concurrently create new instances with the cycle value c, breaking the atomicity of **M**, it also results in *underlocking*.

***Operation* R:** FB locks must be put on each fragments containing composites provided by vendor v. In case of OB, it requires to lock all objects in the selected fragments.

While *overlocking* is not an issue here, *underlocking* is still problematic as other collaborators can concurrently create new fragments containing composites of vendor v.

Example 19 If $Alice$ requests a FB lock, no other collaborator can modify the fragment with root $o5$. In case of an OB locking strategy, $Alice$ needs to put locks on control units $o6$ and $o8$ to prevent the change of their cycle attributes, and their signals $o7$ and $o9$.

However, these locks are unnecessary as no one can modify the selected fragment if FB lock is used, and no one can modify any attributes and references of the selected objects when OB lock is used.

On the other hand, other collaborators can create new controls or modify existing ones (in other fragments) and set their cycle attribute to $medium$ which would violate the intention of $Alice$.

In case of operation **T**, FB is ineffective, while, in case of operation **M**, neither of FB and OB locking strategies are effective. Hence, a more fine-grained approach would be beneficial to lock only the necessary context of the operations.

We propose to associate an *invariant property* with each operation; a *PB lock* can be requested to prevent the violation of such a property, ensuring that users other than the lock owner will not interfere with the execution of the operation.

***Invariant Property of* M** preserves the set of signals provided by control units with cycle c, and their frequencies.

***Invariant Property of* T** preserves control units of type t, their attributes and their references.

***Invariant Property of* R** preserves the set of objects transitively contained by composites provided by vendor v and all attributes and references of these objects.

Example 20 If a PB lock is granted for $Alice$ to perform operation **M** with $c = medium$, other collaborators can modify any part of the model as long as they do not

- modify the frequency of a signal contained by a control unit with $medium$ cycle;
- delete/create/move a signal contained by a control unit with $medium$ cycle;
- change the cycle attribute of a control unit containing signals from/to $medium$.

In our example, operation **M** is best protected by a fine-grained PB lock, operation **T** ideally requires an OB lock whereas a FB lock is most suitable for operation **R**.

5.5.2 Properties Captured by Graph Patterns

Model queries are formulae over models, declarative descriptions of a read-only computation. We have chosen *graph patterns* as the query formalism, since we found them helpful in capturing the structural relationships between objects.

In particular, our prototype implementation uses the VIATRA QUERY framework[Ber+15a], due to its expressive power (first-order formulae with extensions such as transitive closures) and incremental evaluation capabilities. However, the concepts are general and technology independent and they are likely to be adaptable to other model query languages (as well as the more general case of change queries).

Example 21 Listing 5.1 displays a graph pattern in the declarative VIATRA QUERY[Ber+11] syntax, expressing a property of the wind turbine model. The pattern `signals` (pattern name after the `pattern` keyword) selects triples of ⟨`sig`,`frq`,`cycle`⟩ (pattern parameters between parentheses) where `sig` is an instance of the class `Signal` with its frequency attribute set to `frq` (2), and there is a `Control` instance `ctrl` that provides `sig` (3) and has its cycle attribute set to `cycle` (4). This pattern will be used to express the invariant property of the PB lock for operation **M**.

```
1 pattern signals(sig:Signal, frq, cycle) {
2   Signal.frequency(sig, frq);
3   Control.provides(ctrl, sig);
4   Control.cycle(ctrl, cycle);
5 }
```

Listing 5.1: Graph Pattern to Lock Signals

Example 22 The $\mathbb{MS}_{\texttt{signals}}$ of pattern `signals` (Listing 5.1) contains the tuples ⟨`o4,6,low`⟩, ⟨`o7,4,medium`⟩, ⟨`o9,8,medium`⟩ and ⟨`o12,8,high`⟩. If the *cycle* attribute is bound to *medium*, the match set is reduced to ⟨`o7,4,medium`⟩ and ⟨`o9,8,medium`⟩.

After *Bob* executes his operation, the cycle attributes of $o3$ and $o11$ are set to *medium*. Hence, pattern `signals` has two new matches: ⟨`o4,6,medium`⟩ and ⟨`o12,8,medium`⟩. Then, when *Cecile* applies her operation which removes objects $o3$ and $o4$, a match of pattern `signals` disappears: ⟨`o4,6,medium`⟩.

5.5.3 Definitions of Property-based Locks

A PB lock asserts that no user, except for the lock owner, is allowed to perform modifications matching a change query. We use simple change queries [Ber+12] defined as a graph pattern, possibly restricted by parameter bindings to the context of a specific model elements; and the *directions* of change sensitivity, i.e. to forbid disappearance / deletion (d) and/or appearance / addition (a) of pattern matches:

Definition 5.5 (Property-based Lock). PB *Lock* is a tuple formed of an *owner* and a *changeQuery* where a *changeQuery* consists of a graph pattern GP, bindings B and a set of change direction.

$$
\begin{aligned}
&directions \subseteq \{d, a\} \\
&changeQuery = \langle \text{GP}, \text{B}, directions \rangle \\
&lock = \langle owner, changeQuery \rangle
\end{aligned}
$$

Informally, a PB lock guards the MS specified by the GP and its parameter bindings B. A model modification violates a lock iff the MS is extended by a new match with $a \in directions$, or an existing match is removed with $d \in directions$.

Example 23 *Alice* needs to define and put the following lock on the model using the pattern Listing 5.1 to prevent certain modifications described in subsubsection 5.5.1.2:

$$lock\langle Alice, \langle \texttt{signals}, \{cycle = medium\}, \{d, a\} \rangle \rangle \tag{5.1}$$

The $\text{MS}_{\texttt{signals}}$ of pattern `signals` consists of ⟨`o7,4,medium`⟩ and ⟨`o9,8,medium`⟩.

After *Bob* executes his operation, the cycle attributes of $o3$ and $o11$ are set to *medium*. Hence, $\text{MS}_{\texttt{signals}}$ is extended with new matches: ⟨`o3,6,medium`⟩ and ⟨`o11,8,medium`⟩ which violates the lock of *Alice*. When *Cecile* applies her operation which removes $o3$ and $o4$ objects, the match of ⟨`o4,6,medium`⟩ disappears from the $\text{MS}_{\texttt{signals}}$ which likewise results in lock violation.

Note that this is an invariant property, since the lock forbids changes in both directions $\{d, a\}$. It is possible to e.g. use only $\{d\}$, in which case the lock would allow the appearance of new signals of medium-cycled control units, only a removal would be prevented.

We believe that *PB locking* provides a flexible architecture and technical realization of various locking strategies. As such (to address G3), we describe how traditional locking strategies can be implemented on top of *PB locking*.

5.5.4 Support for Traditional Locking Strategies

5.5.4.1 Support for Object-based Locking

Object-based locks prevent the modification of certain objects in the model including the change of their attributes and their references. To describe OB locks using PB locks, a graph pattern needs to be generated to detect when an object is deleted or created. To detect changes of references and attributes of an object, additional patterns are required that separately select the values of each feature.

Example 24 *Bob* requires to lock the object $o3$ and $o11$. Thus, a `control` pattern is generated to observe the deletion and creation of `Control` objects in the model while `type`, `cycle`, `provides` and `consumes` patterns are generated to observe attribute and reference changes of control objects. These locks can be used by Bob where the parameter $ctrl$ is bound to $o3$ and $o11$.

```
1  pattern control(ctrl)
2  { Control(ctrl); }
3
4  pattern type(ctrl, val)
5  { Control.type(ctrl,val); }
6
7  pattern cycle(ctrl, val)
8  { Control.cycle(ctrl,val); }
9
10 pattern provides(ctrl, sig)
11 { Control.provides(ctrl,sig); }
12
13 pattern consumes(ctrl, sig)
14 { Control.consumes(ctrl,sig); }
```

Listing 5.2: Graph Patterns to Select Controls and Their Feature Values

$$lock\langle Bob, \langle \texttt{control}, ctrl = o3, \{d, a\}\rangle\rangle \tag{5.1}$$
$$lock\langle Bob, \langle \texttt{control}, ctrl = o11, \{d, a\}\rangle\rangle \tag{5.2}$$
$$lock\langle Bob, \langle \texttt{type}, ctrl = o3, \{d, a\}\rangle\rangle \tag{5.3}$$
$$lock\langle Bob, \langle \texttt{type}, ctrl = o11, \{d, a\}\rangle\rangle \tag{5.4}$$
$$lock\langle Bob, \langle \texttt{cycle}, ctrl = o3, \{d, a\}\rangle\rangle \tag{5.5}$$
$$lock\langle Bob, \langle \texttt{cycle}, ctrl = o11, \{d, a\}\rangle\rangle \tag{5.6}$$
$$lock\langle Bob, \langle \texttt{provides}, ctrl = o3, \{d, a\}\rangle\rangle \tag{5.7}$$
$$lock\langle Bob, \langle \texttt{provides}, ctrl = o11, \{d, a\}\rangle\rangle \tag{5.8}$$
$$lock\langle Bob, \langle \texttt{consumes}, ctrl = o3, \{d, a\}\rangle\rangle \tag{5.9}$$
$$lock\langle Bob, \langle \texttt{consumes}, ctrl = o11, \{d, a\}\rangle\rangle \tag{5.10}$$

(1) The $MS_{control}$ of pattern `control` has one match $\langle$`o3`$\rangle$. If the object $o3$ is deleted, $MS_{control}$ becomes empty.

(3) The MS_{type} consists of the following tuples: $\langle$`o3,Pump`$\rangle$. If the attribute *type* of $o3$ is changed to Fan, $\langle$`o3,Pump`$\rangle$ disappears from MS_{type} and $\langle$`o3,Fan`$\rangle$ appears.

5.5.4.2 Support for Fragment-based Locking

Fragment-based locks prevent any kind of modifications in a certain fragment including deletion or creation of objects and update features of existing objects. To describe FB locks using PB locks, graph patterns are required to collect all objects in a fragment and their features. These patterns are

extensions of the patterns generated for OB where the objects of the patterns are need to be contained by a root of certain fragment.

Example 25 $Cecile$ requests to lock the fragment whose root is $o2$. Listing 5.3 describes the patterns that are generated for FB locks. Pattern `containedBy` is a helper pattern to select object pairs in parent and child relation using the containment edges of the modeling language. Pattern `fragment` queries each object obj transitively contained by an object $root$ and the $root$ itself. When $o2$ is bound to $root$, all objects contained by $o2$ are selected by the pattern which covers the entire fragment. Pattern `cycleInFragment` selects the cycle attribute of each control contained by the root object. Similar patterns are generated for all features of each class in the modeling language.

```
1 pattern containedBy(parent,child) {
2   Module.submodule(parent,child);
3 } or {
4   Module.provides(parent,child); }
5
6 pattern fragment(root, obj) {
7   find containedBy+(root,obj);
8 } or {
9   root == obj }
10
11 pattern cycleInFragment(root, ctrl, val) {
12   find containedBy+(root, ctrl);
13   Control.cycle(ctrl, val); }
```

Listing 5.3: Fragment-based Lock Patterns

These patterns need to be used with $root = o2$ binding for the definition of $Ceclie$'s locks.

$$lock\langle Cecile, \langle \texttt{fragment}, root = o2, \{d, a\}\rangle\rangle \tag{5.1}$$

$$lock\langle Cecile, \langle \texttt{cycleInFragment}, root = o2, \{d, a\}\rangle\rangle \tag{5.2}$$

...

The MS_{fragment} of pattern `fragment` has the following matches ⟨`o2,Composite`⟩, ⟨`o3,Control`⟩, ⟨`o4,Signal`⟩. If the signal $o4$ is deleted, the match ⟨o4,Signal⟩ disappears. When a new signal o' is created under $o3$, a new match ⟨`o',Signal`⟩ appears.

The $MS_{\text{cycleInFragment}}$ of pattern `cycleInFragment` consists of tuples selecting the cycle attribute of the control $o3$. If the cycle value of control $o3$ is changed to $high$, the match ⟨`o3,low`⟩ disappears and a new match ⟨`o3. high`⟩ appears.

5.5.5 Enforcing Property-based Locks

Here we focus on the enforcement of *lock-operation compatibility* (whether an operation is acceptable given the set of active locks) only; see subsubsection 5.5.6.8 for discussion of *lock-lock compatibility* (whether a new lock request should be granted or rejected given the set of active locks) which could cause deadlock situations.

5.5.5.1 Algorithm to Enforce Locks

When an attempt is made to modify the model, the current set of active locks needs to be checked; the changes need to be rejected if a lock is violated. The rejection requires the ability to revert the model to its original state. Hence, our approach is built on the well-known concept of transactions

where the operations are wrapped into a single transaction that can be executed. After the execution it can be rolled back to regain the unmodified model.

We introduce the function TRANSACTION that takes $operations$ as input and returns an undoable $transaction$. The function EXECUTE attempts to execute the $transaction$ on a certain $model$. If it fails due to access control violation or conflicts, it returns $false$, otherwise it returns $true$ and applies the $operations$ in the $transaction$ on the $model$. The ROLLBACK function is responsible for undoing a certain $transaction$ on a given $model$.

Now we present an algorithm depicted in algorithm 5 to enforce PB locks as it is address by goal **LG2**. The novelty of the algorithm is the decision process to determine when a transaction needs to be rolled back due to lock violation.

The ENFORCELOCKS function takes a $model$, a $user$, a set of $Locks$ and a sequence of $Operations$ as input where the user attempts to execute his/her operations on the model but the locks cannot be violated. If a lock is violated the operations are rejected and the model remains untouched.

The algorithm consists of 3 main parts:

Phase 1: The algorithm iterates through all the locks and calculates the MS_{lock} of each lock owned by a collaborator other than the $user$.

Phase 2: The operations are attempted to be executed as a $transaction$ to provide the ability of roll back. If the execution fails, the function terminates.

Phase 3: The MS'_{lock} of each lock is reevaluated and compared to MS_{lock} calculated in Phase 1. The $transaction$ is rolled back and the function terminates if MS_{lock} is modified in a direction that is disallowed by the lock.

Note that the practical execution of *Phase 1* and *Phase 2* can be simplified using incremental evaluation techniques [Ujh+15]. Additionally, we assume the algorithm is executed by a collaboration framework (see Section 4.5) on the server side that ensures any other submission attempt including lock addition will be rejected while the algorithm is under execution.

5.5.5.2 Correctness Criteria of the Algorithm

We identify 4 correctness criteria of the presented algorithm and their proofs are sketched in the followings:

Theorem 1. (Termination) *Let $\overrightarrow{Op}$ be an operation sequence leading the model $M \xrightarrow{\overrightarrow{Op}} M'$ executed by a user $u \in Users$ and let $Locks$ be the currently active locks.*

The termination of the algorithm is guaranteed.

Proof. — (Sketch) *Phase 1* iterates over the finite set of $Locks$ and we can assume the function PM terminates. *Phase 2* executes the ordered and finite number of operations in a sequence. Hence, the application of the operations terminates. If conflict or access control violation occurs the algorithm terminates. *Phase 3* iterates over the finite set of $Locks$ and the algorithm eventually halt after the iteration. ∎

Algorithm 5 Enforcement of Property-based Locks

```
function EnforceLocks(model, user, Locks, Operations)
    ▷ Phase 1: Evaluate match sets of relevant locks
    Map⟨lock, MS⟩ relevantMap ← ∅
    for all lock in Locks do
        if lock.owner ≠ user then
            cq ← lock.changeQuery
            MS_lock ← PM(cq.pattern, cq.bindings, model)
            relevantMap.put(lock, MS_lock)
        end if
    end for
    ▷ Phase 2: Operations are wrapped into a transaction
      and the transaction is attempted to be executed.
    transaction ← Transaction(Operations)
    if Execute(transaction, model) fails then
        ▷ The execution can fail if access control rules
        ▷ are violated or conflicts are introduced
        return
    end if
    ▷ Phase 3: Reevaluate the match sets, check violations
      and roll back the transaction if it is required
    for all ⟨lock, MS_lock⟩ in relevantMap do
        ▷ Reevaluate the match sets of relevant locks
        cq ← lock.changeQuery
        MS'_lock ← PM(cq.pattern, cq.bindings, model)
        if MS'_lock \ MS_lock ≠ ∅ and a ∈ ctx.directions or
           MS_lock \ MS'_lock ≠ ∅ and d ∈ ctx.directions then
            Rollback(transaction, model)
            return
        end if
    end for
    return
end function
```

Theorem 2. (Correctness) *Let $\overrightarrow{Op}$ be an operation sequence evolving the M to M' (denoted as $M \xrightarrow{\overrightarrow{Op}} M'$) executed by a user $u \in Users$ and let $Locks'$ be the active locks owned by other collaborators $\subseteq (Users \setminus \{u\})$.*

$$\exists l \in Locks' : l \text{ is violated, after } M \xrightarrow{\overrightarrow{Op}} M'$$
$$\Rightarrow \overrightarrow{Op} \text{ is rejected}$$

Proof. — (Sketch) *Phase 3* is responsible for deciding whether a lock is violated. If a lock is violated, the operations are rolled back and the algorithm terminates where the termination ensures that the operations cannot be executed again during the process. ∎

Theorem 3. (Completeness) *Let $\overrightarrow{Op}$ be an operation sequence evolving the M to M' (denoted as $M \xrightarrow{\overrightarrow{Op}} M'$) executed by a user $u \in Users$ and let $Locks'$ be the active locks owned by other collaborators $\subseteq (Users \setminus \{u\})$. Additionally, let AC violation be true if an access control rule is violated by $\overrightarrow{Op}$ and let Conflict be true, if another $\overrightarrow{Op'}$ has already evolved the M to M''.*

$$\overrightarrow{Op} \text{ is rejected} \wedge \neg \text{Conflict} \wedge \neg \text{AC violation}$$
$$\Rightarrow \exists l \in Locks' : l \text{ is violated, after } M \xrightarrow{\overrightarrow{Op}} M'$$

Proof. — (Sketch) *Phase 2* is responsible for executing the operations on the model and it checks if conflicts are introduced or access control rules are violated. When none of them occurs, the algorithm continues with *Phase 3*. In *Phase 3*, the operations are rolled back if a lock is violated and algorithm terminates. If there is no lock violation, the algorithm terminates without rollback. ∎

Theorem 4. (Determinism) *Let $\overrightarrow{Op}_1$ and $\overrightarrow{Op}_2$ be operation sequences evolving the model M to M' (denoted as $M \xrightarrow{\overrightarrow{Op_1}} M'$ and $M \xrightarrow{\overrightarrow{Op_2}} M'$, respectively) executed by the same user $u \in Users$ and let $Locks'$ be the active locks owned by other collaborators $\subseteq (Users \setminus \{u\})$.*

$$\exists l \in Locks' : l \text{ is violated, after } M \xrightarrow{\overrightarrow{Op_1}} M'$$
$$\Rightarrow l \text{ is violated, after } M \xrightarrow{\overrightarrow{Op_2}} M'$$

Proof. — (Sketch) As l is evaluated, we can assume that $\overrightarrow{Op}_1$ and $\overrightarrow{Op}_2$ do not introduce conflicts and do not violate access control rules (which is discussed in Chapter 3 in details, but out of scope for the current chapter). *Phase 3* is independent from the operations when the locks are evaluated as the match sets are calculated at the two states of the model (M and M'). Hence, the lock violation is associated to a state of the model. Thus, two operation sequences leading a model to the same states will violate the same locks. ∎

5.5.6 Evaluation

We have carried out an initial evaluation to compare the effectiveness of three different locking strategies for conflict prevention with respect to overlocking. Although the FB and OB locking approaches are widely used in the industry, we found no systematic comparison to evaluate the effectiveness of these strategies. Hence, our comparison is an additional contribution.

The evaluation is based on stochastic simulation of collaborators modeled as a Discrete Event System Specification (DEVS) [CZ88]. Each collaborator attempts to lock and modify the same auto-generated semi-synthetic models. The simulation investigates the following research questions:

Q1: How do the success rates of the locking strategies vary with increasing *model size* i.e. increasing number and size of fragments?

Q2: How do the success rates of the locking strategies vary with increasing *number of collaborators*?

To answer these questions, we calculate the success rates $\mathcal{SR}$ of locking strategies by counting the number of modification attempts accepted by all existing locks and dividing it with the total number of attempts:

$$\mathcal{SR} = \frac{\text{modification attempts accepted by all locks}}{\text{all modification attempts}}$$

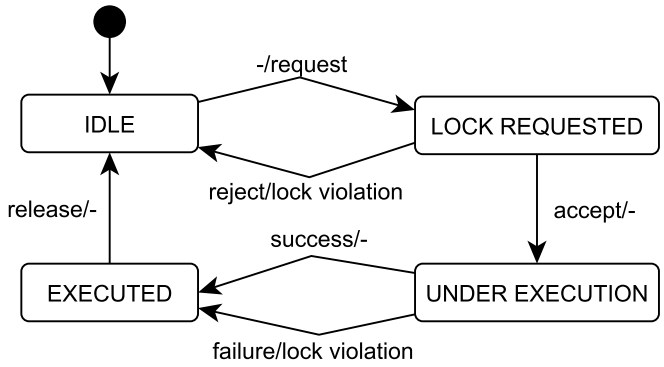


Figure 5.11: Behavior of a Collaborator

5.5.6.1 Behavior of Collaborators

The behavior of each collaborator is modeled as a state machine depicted in Figure 5.11. At the beginning, each collaborator is in the `WAITING` state. Then the collaborator *request*s to put his/her lock on the model. Until receiving the result of the lock request, the collaborator stays in the `LOCK REQUESTED` state. If the lock request is *rejected*, a *lock violation* event is produced and the collaborator steps back to `WAITING` state. Otherwise, the lock is *accepted* and the collaborator executes his/her modifications in the state `UNDER EXECUTION`. The execution can terminate successfully (*success* event) or with failures (*failure* event). The latter case means lock violation, thus a *lock violation* event is produced. Both cases lead the state machine to the `EXECUTED` state. The last step is to *release* the acquired lock and step back to the `WAITING` state.

During the simulation, our task is to collect the number of produced *lock violation* events and subtract it from the number of all attempts in order to calculate the number of accepted attempts.

5.5.6.2 Operations of Collaborators

In order to obtain repeatable (cyclic) behavior for stochastic simulation, the operations introduced in subsubsection 5.5.1.1 are replicated with corresponding revert operations that remove the objects and references inserted by the original operations and reset the attributes modified by the original operations. Each collaborator alternately executes an original operation and its revert operation to preserve the basic structure of the model while maintaining the context of pattern matches. To compare the locking strategies with respect to *overlocking*, locks are defined for each operation using FB, OB and PB approaches.

5.5.6.3 Model Synthesis

For the simulation, we used the example wind turbine metamodel with slight modifications: the *type* and *cycle* attributes of control units were abstracted to string values.

The evaluation was performed with synthesized models of various size. Each model has a root *Composite* object containing F model fragments to increase the horizontal dimension. Each model fragment contains a copy of the example structure depicted in Figure 5.9 consists of 4 *Composite*s, *Signal*s and *Control* units. Each copy of the structure contains at most one other

Table 5.5: Means and Rates of Waiting and Execution times

	λ_{WAIT}	$E_{\text{WAIT}}(x)$	λ_{EXEC}	$E_{\text{EXEC}}(x)$
M	$\lambda^{\mathbf{R}}_{\text{WAIT}} = \frac{1}{24}$	$24h$	$\lambda^{\mathbf{R}}_{\text{EXEC}} = \frac{1}{3}$	$3h$
T	$\lambda^{\mathbf{T}}_{\text{WAIT}} = \frac{1}{12}$	$12h$	$\lambda^{\mathbf{T}}_{\text{EXEC}} = \frac{1}{2}$	$2h$
R	$\lambda^{\mathbf{M}}_{\text{WAIT}} = \frac{1}{4}$	$4h$	$\lambda^{\mathbf{M}}_{\text{EXEC}} = \frac{1}{1}$	$1h$

copy connected to the structure's root $o1$. Hence, each fragment has D copies of the structure in the hierarchy to increase depth of the fragments. Altogether, each model has 1 root $+$ $(4$ composites $+$ 4 signals $+$ 4 controls$)$ $\times$ F fragments $\times$ D depth objects with references of $(1$ submodules from root $+$ 7 submodules $+$ 4 consumes $+$ 4 provides in the structures$) \times F \times D +$ $F \times D$ submodules to connect the substructures under each other.

The *vendor*, *type* and *cycle* attributes were limited to U_{max} different values chosen with uniform probability where U_{max} is the maximal number of collaborators working on the model.

5.5.6.4 Timing of Simulation

We assume that granting, rejecting and releasing a lock can be processed instantaneously (as an atomic operation), but the *waiting time* before requesting a lock and the *execution time* of an operation need to be handled to simulate overlapping lock requests and executions. Exponential distribution was used to approximate the behavior of human collaborators to simulate *off-line collaboration scenarios* (like SVN or Git). Hence, each type of operations (**R**,**T**,**M**) has various rates of waiting time ($\lambda^{\mathbf{R}}_{\text{WAIT}}$, $\lambda^{\mathbf{T}}_{\text{WAIT}}$, $\lambda^{\mathbf{M}}_{\text{WAIT}}$) and execution time ($\lambda^{\mathbf{R}}_{\text{EXEC}}$, $\lambda^{\mathbf{T}}_{\text{EXEC}}$, $\lambda^{\mathbf{M}}_{\text{EXEC}}$). Table 5.5 shows the sample rate values, where we assume that replacement operations **T** are executed once a day, testing operations **T** are introduced twice a day while maintenance operations **M** are scheduled 6 times a day in average — but the actual execution and waiting times are random variables with exponential distribution.

5.5.6.5 Coupled DEVS Model

The coupled DEVS model[CZ88] consists of one atomic server model and N atomic collaborator models (N is the number of collaborators in simulation). Each collaborator model has exactly one channel to the server model.

For the collaborator model, the states, the internal and external transitions are depicted as a state machine in Figure 5.11. The time advances (ta) of `Idle` and `Under Execution` states are parametrized for each type of collaborators (shown in Table 5.5) while all other ta values are treated as zeros. Our rationale is to handle the request and release of locks as atomic operations with negligible execution time.

The server model (not detailed in the paper) has a single state with several self-loop transitions to react to the collaborator's requests by providing inputs for the collaborator model. The ta values in the server are handled as zero. This setup means that locks are immediately evaluated and placed.

5.5.6.6 Simulation Setup

In the evaluation, we simulated collaborators of $U = 9$ and $U = 27$ where U is the number of active collaborators. Each operation type (**R**, **T**, **M**) were executed by $\frac{U}{3}$ number of users. We assigned a

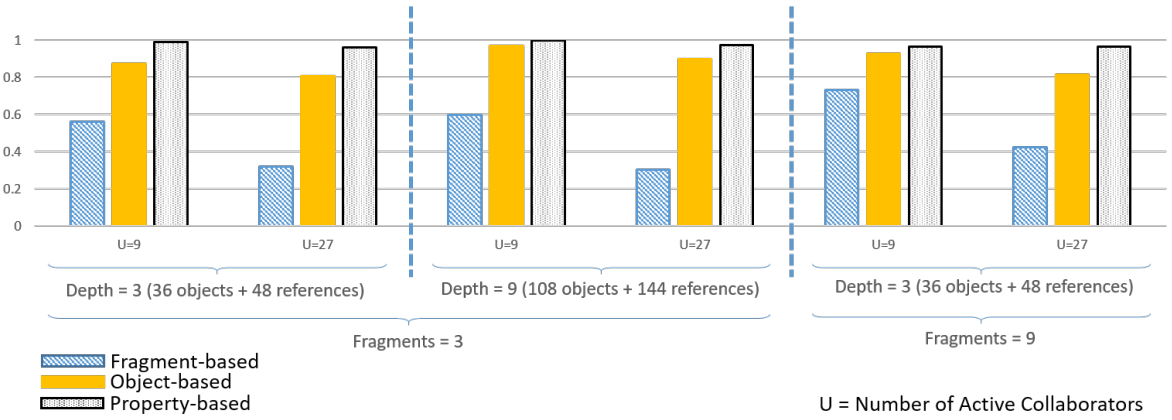


Figure 5.12: Success Rates of the Simulation - Property-based (filled), Object-based (crossed) and Fragment-based (dotted)

unique operation for each user by setting unique values to c, t and v parameters to avoid regular conflicts by ensuring that multiple users cannot modify exactly the same part of the model. Each simulation ran until $(1\ \mathbf{R} + 2\ \mathbf{T} + 6\ \mathbf{M}) \times (\frac{U}{3}) \times Days$ execution were attempted where $Days = 10$. This calculation gives the expected number of execution attempts after 10 days. Finally, the size of fragments was increased from $F = 3$ to $F = 9$ and the depth of each fragments was increased from $D = 3$ (including 36 objects and 48 references) to $D = 9$ (including 108 objects and 144 references) to cover small and large number of *collaborators*, *fragments* and *sizes of each fragment*.

5.5.6.7 Evaluation of Results

We executed the simulations[1] 10 times for each locking strategy and the parameter combinations of (U, F and D). The results visually summarized in Figure 5.12 show the median of success rates: the higher the value, the better the strategy performs. Each cluster depicts the success rates for all three locking strategies with a certain parameter setting where the checkered, solid and dotted columns are associated to the FB, OB and PB locking strategies, respectively.

Related to research question **Q1**, we can observe that

- FB locking shows better results when the number of fragments (F) is increased, but its efficiency decreases when the sizes of the fragments (D) are growing but the number of fragments remains — with 73.3% success rate in best case, 30.2% in worst case.
- OB locking is insensitive to which dimension of the model increases, larger models increase its efficiency — 97% in best case, 81% in worst case.
- The same applies to PB locking, but success rates are consistently better than for OB and FB locking in all simulated cases — 99.9%: best case, 96.1%: worst case.

As for research question **Q2**, an increasing number of collaborators results in a decreased success rate (thus decrease scalability) both for OB and, especially, for FB strategies where OB locking shows significantly better results compared to FB. However, success rate is not compromised by *property-based locking* even when the number of collaborators grow.

[1] Raw data and reproduction instructions are at http://tinyurl.com/models17-locking

Therefore, we may conclude that PB locking can successfully provide technological foundation for different locking strategies and it enables to define fine-grained locks that increase the effectiveness of collaboration.

5.5.6.8 Discussion of Limitations

User Experience. We believe that the proposed approach of property-based locking can provide a helpful underlying lock enforcement infrastructure to realize low-conflict collaboration systems. However, manually defining complex properties using graph patterns can be a demanding task. Therefore we expect that modeling tools will have built-in definitions for such lock properties in case of both elementary model manipulation commands and more complex domain-specific refactoring actions. Additionally, domain experts could provide a library of frequently needed lock properties, from which the collaborators can easily select and parameterize a lock to request. The detailed evaluation of user experience is out of scope for the current paper.

Lock-lock Compatibility. We have so far only considered lock-operation compatibility. Unlike the other two approaches, PB locks only impose a restriction on what other users are allowed to do, but reveal very little information on how the lock owner is intending to change the model. Therefore, when the lock management service has to decide between accepting and rejecting a lock request, it is not possible to tell whether the owners of other locks would likely attempt any operations in the future that would violate this lock if granted. Contrast with e.g. FB, where a lock request on a fragment is rejected if someone else has already locked that fragment, so that the owner of the earlier lock can carry on undisturbed.

As a consequence, PB has no general way of rejecting locks at the time the request is made (the corresponding *reject* transition of Figure 5.11 is not used), and incompatibility will only be detected later, when an actual violating operation is performed. However, as discussed above, PB is seen as a common platform for enforcing locks in a variety of scenarios; in some of these special cases (such as when PB lock definitions are mapped from FB or OB lock requests, we do have a notion of lock-lock compatibility, which a sufficiently careful implementation could enforce.

Deadlock. Due to *lock-lock compatibility* issues of PB, deadlock situation can occur when two or more locks prevent the modification of certain model parts by the lock owners. For instance, (a) a property locks all the Composite objects of vendor C while (b) an other property locks all the submodules of connected to a Composite provided by the vendor A. In this case, neither of the lock owners are able to modify O5 as the owner of (a) is prevented by lock (b) and the owner of (b) is prevented by the lock (a). A naive approach would be the following: when the modifications of a lock owner is rejected due other lock violation, the lock owner should remove his/her locks and wait until the other lock owners remove their locks as well. However, resolving such deadlocks requires further investigation in the future to provide better strategies.

5.6 Contributions

In this chapter, we presented an automated technique for three-way model merge exploiting design space exploration in the background and an underlying infrastructure to realize a property-based locking strategy as a common generalization of existing fragment-based and object-based locking approaches.

Our model merge technique automatically derives consistent and semantically correct merged models in all possible ways and also highlights the remaining (unresolved thus conflicting) model differences.

Our property-based locking approach enables to define fine-grained locks as properties described as graph patterns allowing only modifications that preserve the property.

Contribution 3 I proposed a fine-grained property-based locking technique to avoid conflicts and an automated three-way model merge technique to resolve conflicts. [*c*4], [*j*1], [*c*5], [*c*7], [*c*6], [*c*9], [*c*13], [*c*16]

- **3.1 Fine-grained Property-based Locking.** I proposed a property-based locking technique as generalization of traditional fragment-based and object-based locking techniques which captures fine-grained locks as graph patterns and exploits incremental query engines to maintain and evaluate locks. [*j*1], [*c*5], [*c*7], [*c*6], [*c*16]
- **3.2 Automated Model Merge using DSE.** I proposed an automated three way model merge technique by adapting rule-based design space exploration to derive consistent and semantically correct merged models. [*c*4], [*c*9], [*c*6], [*c*13]
- **3.3 Generic Scalability Benchmark.** I proposed a scalability benchmark for model merge by adapting an existing performance benchmark for model queries. [*c*13]
- **3.4 Evaluation.** I evaluated the scalability of the automated model merge and I compared the effectiveness of fine-grained property-based locking and traditional locking strategies for conflict prevention on a case study of offshore wind turbine controllers. [*c*5], [*c*13]

The novel concept of property-based locking has been carried out in an international collaborative work [*c*16] with Marsha Chechik, Fabiano Dalpiaz, Jennifer Horkoff and Rick Salay where my contributions are the first adaption and implementation in a real practical setting and its evaluation in the context of MONDO EU FP7 research project. The concept of using DSE for merging purposes is the contribution of István Ráth, whereas Viatra DSE implementation, on which DSE Merge relies, is the contribution of Ákos Horváth[**horvath2014phd**], Ábel Hegedüs[Heg14] and András Szabolcs Nagy[HHV15b]. Systematic evaluation of the efficiency of the DSE Merge technique from the user point of view is the contribution of Ankica Barisic[Bar+18] supported by MPM4CPS EU COST Action.

Chapter 6

Synchronization of View Models

6.1 Introduction

The main objective of this chapter is to address the challenge of bidirectional sychnronization of view models **C-III** introduced in Section 1.3.1 to provide views of the detailed system models to collaborators.

Usage of view models in collaborative modeling increases the precision of the modification introduced by collaborators as they can work on the model only in their terminology. It also eliminates the chance to introduce invalid changes unrelated to their specialty.

We propose a declarative approach deriving view models by unidirectional transformations and a technique to automatically calculate possible source model candidates for a set of changes in different view models. In particular, we aim to address the following goals:

G1 *Lightweight specification of multiple views and their chain.* The solution shall propose a lightweight mechanism to specify multiple view models for the same source model and chains of dependent views.

G2 *Incremental maintain of views.* The solution shall automatically and incrementally maintain view models for the same source model and a change in the source model needs to be propagated transitively to all dependent views.

G3 *Automatically calculate valid candidates.* The solution shall automatically calculate well-formed source model candidates for a set of changes in different view models.

G4 *Impact of a view modification.* The solution shall identify the possibly affected partition of the source model to restrict the impact of a view modification.

Structure Rest of the chapter organized as follows. Related work is overviewed in Section 6.2. Section 6.3 introduces a motivating example in the context of telecare systems. Section 6.4 overviews the concepts of view models and deriving view models by query-based unidirectional transformations. Maintenance of view models (from source model to view models) is discussed in Section 6.5, then our backward change propagation approach (from view models to source models) is presented in Section 6.6. An initial experimental evaluation is provided in Section 6.7. Finally, Section 6.8 concludes the chapter by a summary and state the contributions achieved in the topic of *View Model Synchronization.*

Approach	Logic Solver	Traceability	WF const.	Partial Model	Interoperability
ATL[Xio+07]	-	-	-	-	+
TGG[Sch94]	-	+	-	-	+
QVT-R[QVT]	-	+	+	-	+
QVT-R with Alloy[MC13]	SAT Solver	-	+	-	+
JTL[Cic+10]	ASP	-	-	-	+
MTE with MILP[CK13]	MILP	-	-	-	+
EMF Views [Bru+15]	-	+	-	+	+
F-Alloy [GK15]	SAT Solver	-	+	-	+
QueST[Gho+15]	SAT solver	+	+	-	+
Our solution	SAT solver	+	+	+	+

Table 6.1: Comparison of related approaches

6.2 Related work

Most existing view model synchronization techniques use bijective transformations where transformations can be executed in both the forward and reverse directions such in lenses [Fos+07], injective functions [MHT04] or ATL [Xio+07]. Triple Graph Grammars (TTG) [Sch94] are a well-know approach for model synchronization [Her+15] where the forward and reverse transformation rules are derived automatically from a bidirectional rule definition. A special class of TGG is View TGG [Anj+14] which is specialized for efficient update propagation. As a fundamental difference, our approach uses patterns to define the well-formedness constraints and the view instead of generative graph grammars.

Most closely related approaches for view synchronization are listed in Table 6.1. To compare them to our approach, we use several characteristics to guide the structure of this section.

Using Logic Solvers. Using logic solvers for generating possible source and target candidates is common part of several approaches. [Cic+10] uses Answer Set Programming, [CK13] maps the problem to Mixed Integer Linear Programming. Those approaches use solvers to select model elements which may alter the matches related to view model changes (similar to the calculation of the affected part). As a difference, our approach takes the whole DSL specification into the account, to change source model elements which are only implicitly related to the view changes, caused by the interaction of the metamodel, the WF constraints and other view models.

[MC13] uses Alloy to generate change operations on the source model which leads to a modified source model which is (i) well-formed and (i) consistent with the changed target model. As a difference, our solution creates the changed partition (and not the change operations). [Gho+15] and [GK15] converts the transformation to Alloy similarly, but do not handle WF constraints of the source model, and changes the whole source model. By selecting the affected part, our solution likely has better scalability as it has to manage less objects: [GK15] scales up to 20 objects in the source and target models in total, and no other measurement is given. This technique is also suggested in the future work of [MC13].

Traceability Links. For the backward propagation of changes, use of traceability links is a well-accepted approach to define which part of the source model has to be updated upon a change on the target model. In [Sch94], these links are stored as a *correspondence model* where their maintenance is derived from the TGG rules. [QVT] also specifies *trace* classes to facilitate and maintain traceability links. [Gho+15] stores traceability links in Alloy[Jac] as a bijective mapping. [Bru+15] uses a *weaving* model that stores the traces of links between different models in the view, however all objects in the view model act as proxies to an object in the source model. Our solution builds and maintains a

traceability during the forward propagation of changes. Moreover, we reuse the information stored in the traces to improve the affected part calculation for changes in the view.

Well-formedness constraints. To avoid the calculation of *ill-formedness* source models after the backward propagation of a view model change, well-formedness constraints should be taken into account. This property is supported in the specification of [QVT] and by [MC13] and in our approach.

Partial synchronization. [Het10] defines *partial synchronization* to apply the changes of target model only to the relevant part of the source model. This reduces the number of possible source candidates. Our approach identifies the fixed part of the source model, that cannot be changed, and selects the complement of this part which will be the basis of constraints. While [Het10] defines a formal framework for model synchronization, our solution can be interpreted as an efficient and view-model specific realization of it.

Partial models have certain similarity to *uncertain models*, which offer a rich specification language [FSC12; SC15] amenable to analysis. Uncertain models provide a more expressive language (called MAVO annotations) compared to partial snapshots (which implements only annotation V and O from MAVO) but without handling additional WF constraints. Such models document semantic variation points generically by annotations on a regular instance model, which are gradually resolved during the generation of concrete models. An uncertain model is more complex (or informative) than a concrete one, thus an a priori upper bound exists for the derivation, which is not an assumption in our case.

Concrete models compliant with an uncertain model can synthesized by the Alloy Analyzer [SFC12], or refined by graph transformation rules [Sal+15]. Each concrete model is derived in a single step, thus their approach is not iterative like ours. Scalability analysis is omitted from the respective papers, but refinement of uncertain models is always decidable.

We believe that our contribution is novel in the context of view model synchronization in the sense that the effects of uni-directional and non-injective forward rules are reversed by mapping models, WF constraints and rules into first-order logic and then using iterative calls to a SAT-solver. Furthermore, the consistency criteria for the derived source models is stricter as it includes WF constraints of the source language (and not only consistency constraints of the transformation). Moreover, a fix partial model is specified upon a change in the target model using traceability links maintained during the forward propagation. Finally, iterative and incremental calls to logic solvers scales better then a full model generation run.

6.3 Motivating scenario

Our change propagation technique will be illustrated on a case study of a remote health care system developed in the Concerto project [Conc], which develops an environment for pulse and blood pressure measurement controlled by a smart phone, which is illustrated in the upper left part (1.) of Figure 6.1. Measurements of pulse and blood pressure is measured by the sensors of a mobile phone, which are executed periodically triggered daily by the phone timer. The completion event of measurements triggers the processing of sensor data: pressureDone and pulseDone. The result of the measurement is collected in reports pulseReport and pressureReport, and sent to the different hosts. In our case study, the blood pressure is sent to the general practitioner (gp) of the patient for logging, and signs of hearth failure is sent to hospitals (modeled by emergency).

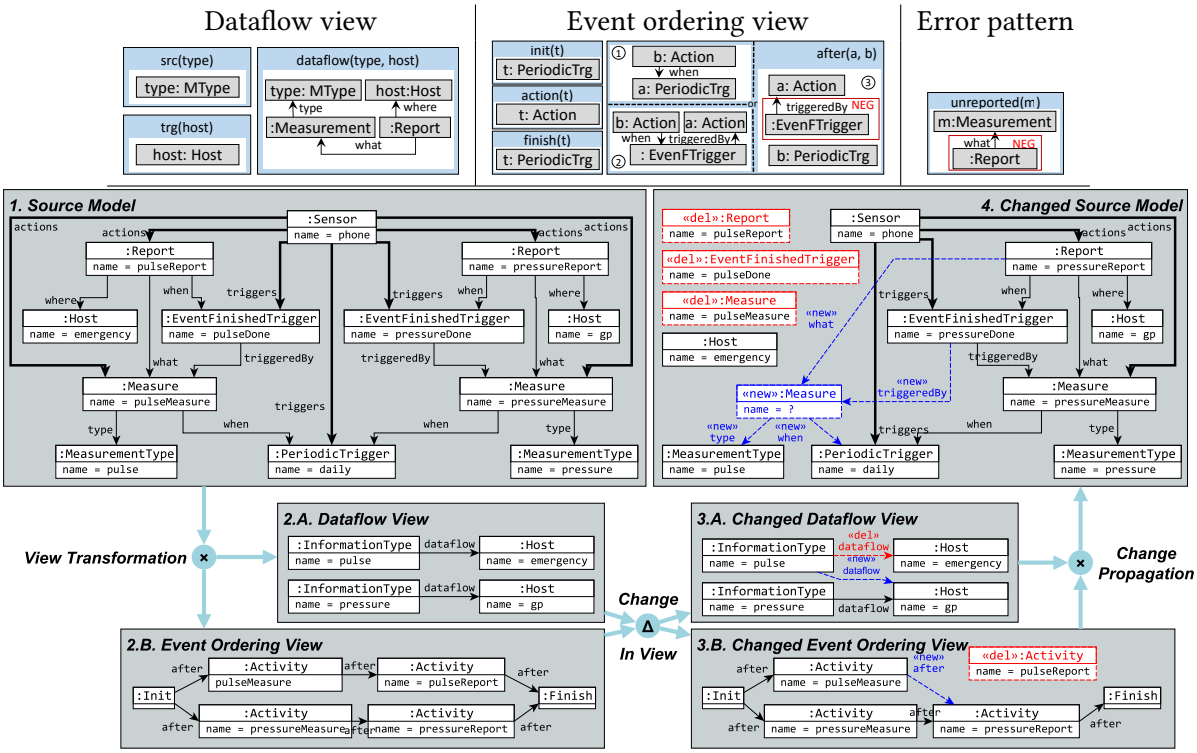


Figure 6.1: Motivating scenario on a healthcare example

Example 26 Two view models are derived from this source model in our telecare example which are maintained as the source model changes. The *Dataflow* view ($2.A$) shows which Hosts will be notified about each InformationTypes, while *Event* layout ($2.B$) describes event sequences represented by Activation nodes with *after* links between them leading from an Init node to a Finish node.

Let us now assume that changes are made in views illustrated in $3.A$ and $3.B$: $3.A$ represents a change where dataflow from Pulse to Emergency is redirected to the General Practitioner (denoted by «del» and «new»). In $3.B$, the action dedicated to report the pulse is removed from the view, but the remaining report waits for the completion of both measurement (denoted similarly). Our technique will allow to automatically generate valid and well-formed source model candidates like ($4.$) that conforms to the current state of the view model.

6.4 View Models

In a domain-specific modeling tool, the underlying domain model is presented to the engineers in different views. These views are frequently represented as models themselves (called view models and denoted by M_V in the sequel), which are populated from the underlying domain model (called source model, M_S). One source model may populate multiple view models. In a general setting, view models can be detached from the source model to such an extent that they correspond to a different language, thus they need to be compliant with a *view metamodel* MM_V and satisfy *view-specific well-formedness constraints* WF_V.

Definition 6.1 (View Model). A view model is derived from the source model by a unidirectional *forward transformation.*

$$M_V = \mathsf{fwd}(M_S)$$

.

This is a restricted class of model transformations where query-based declarative techniques are especially suitable [Gho+15][*c*18]. Efficient live maintenance of a view model upon changes of the source model can be carried out by incremental transformation techniques [HLR06][*c*18] even for multiple view models ($M_{Vi} = \mathsf{fwd}^i(M_S)$) or chains of view models ($M_V = \mathsf{fwd}^2(\mathsf{fwd}^1(M_S))$).

A forward transformation frequently creates and maintains a trace model $T = T_{obj} \cup T_{fea}$ between the source M_S and view M_V models. An *object trace* T_{obj} is a relation which connects activations of rules (queries) in the source model M_S to objects of the view model M_V. Similarly, a *feature trace* T_{fea} (i.e. link or slot trace) is a relation which connects rule activations in the source model M_S to features in the view model M_V.

While view models may immediately reflect live changes in the source model, view models were immutable by the engineers, which restricts the use of view models in an industrial setting. Hence, we allow view models $M_{V1}, \dots M_{VN}$ to be changed directly to $M'_{V1}, \dots M'_{Vn}$ and present an approach for backward change propagation for view models to an updated source model M'_S using logic solvers.

6.4.1 Definition of view models

We propose to use declarative queries as derivation rules to (i) specify new view model elements in the target M_V, and (ii) maintain a trace model between the source M_S and view M_V models based on the matching queries (patterns). A graph pattern describes structural conditions on a model with a combination of path and type expressions equivalent to first order logic predicate. A derivation rule consists of a pattern predicate p and an action part where for each activation m of predicate p, the action part is fired.

Definition 6.2 (Lookup Functions). To help navigation along traces, two (injective partial) lookup functions are introduced: $lookup_{VS}(o)$ maps a view object o to a predicate p with its activation m over the source model and $lookup_{SV}(p(m))$ maps an activation to a view object when $(p(m), o) \in T$.

.

Definition 6.3 (Derivation rules). The following actions are used in derivation rules:

- AddObj(class:*Class*) Activation m of precondition p creates an entry $(p(m), o)$ in the trace with a unique view object o in the view model with the corresponding type class, where $o = lookup_{SV}(p(m))$. For each activation m of precondition p, there exists a unique $o \in O_V$ view object, where

$$M_S \models p(m) \Leftrightarrow T \models (p(m), o) \Leftrightarrow M_V \models \mathsf{class}(o)$$

- AddRef(ref: *Ref*, sp: *src.pre*, sm: *src.match*, tp: *trg.pre*, tm: *trg.match*)
  Activation m of precondition p creates an entry $(p(m), (o_s, o_t))$ in the trace with a link ref in the view model from the source o_s to the target o_t, where $o_s = lookup_T(\mathsf{sp}, \mathsf{sm}))$ and $o_t = lookup_T(\mathsf{tp}, \mathsf{tm}))$. For each activation m of precondition p

exists $o_s, o_t \in O_V$:

$$M_S \models p(m) \Leftrightarrow T \models (p(m), (o_s, o_t)) \Leftrightarrow M_V \models \mathsf{ref}(o_s, o_t)$$

- AddAtt(attr: Att, sp: $src.pre$, sm: $src.match$, val: $value$)
  Activation m of precondition p creates an entry $(p(m), (o_s, val)$ in the trace by setting the slot attr of o_s view object to val, where $o_s = lookup_T(sp, sm))$. For each activation m of precondition p exists $o_s \in O_V$:

$$M_S \models p(m) \Leftrightarrow T \models (p(m), (o, \mathsf{val})) \Leftrightarrow M_V \models \mathsf{attr}(o, \mathsf{val})$$

.

As a result, we obtain a declarative formalism for defining view models with execution semantics compliant with incremental and live graph transformations [Rát+08]: when a new activation of a forward rule is detected, the corresponding view elements are created and when a previously existing activation of a forward rule disappears the related view elements are removed. However, this is still a restricted subclass of model transformations since (1) each rule creates exactly one new element (object or link) in the view model, and (2) the transformation is monotonic in the sense that a view element always depends on the existence of a match of a positive pattern (i.e. we disregard from cases when a view element is created when a pattern cannot be matched).

Example 27 Queries used for defining the views of our motivating example are depicted in the top of Figure 6.1. For the *Dataflow* view, src query selects all the `MeasurementType` to create `InformationType` instances in the view, while trg is responsible for creating `Host` instances from the `Host` objects in the source model. The dataflow pattern has two parameters and selects all the type and host pairs that are connected to each other via a `Measurement` and a `Report` objects. The action part of the rule will create an edge between an `InformationType` and a `Host` associated with the two parameters upon a match appears for the pattern. Similarly, the *after* pattern is responsible for setting the after edge between view model objects. In case of (1.), the edge will go from an `Init` object to an `Activity`, (2.) describes the connection between two `Activity`, while (3.) activates when an `Action` (common ancestor of `Report` and `Measurement`) is not triggered by any `EventFinishedTrigger`. The init and finish queries create `Init` and `Finish` objects in the view for each `PeriodicTrigger` objects in the source model. Finally, the action query builds `Activity` instances from all `Action` objects.

6.4.2 Characterization of query-based transformation of view models

S. Hidaka et al. [Hid+15] classifies bidirectional transformation approaches based on their features. According to it, our work is a *syntactic approach* for *forward functional* transformation of *MDE artifacts*. View models contain *no complement information*, hence they are regular models without additional annotations. These models are *total targets* as the full view model is specified by the consistency relations. Definition of a view model is *unidirectional*, however the expressiveness of definition is *Turing incomplete*. Forward propagation of the *operation-based* changes are *live*, *incremental* and executed *automatically* that also maintains *explicit traces*. However, that approach has *incomplete change support*, thus only the modification of the source model is supported.

6.4.3 Derivation rules by query annotations

In our context, *view models* are conceptually equivalent to regular models. Their structure is defined by a metamodel, which thus consists of (view) classes, features (e.g., links and slots). However, the lifecycle (i.e. existence and non-existence) of view model elements is strongly tied to certain elements of the underlying source model as being specified by query-based *derivation rules*.

In principle, an arbitrary model transformation language could be used to specify how to synthesize view models. To address **G1**, our approach uses *a lightweight mechanism exploiting annotations* of the previous query language to capture derivation rules, instead of relying on a full-fledged model transformation language. As a result, we obtain a declarative formalism compliant with the execution semantics of incremental and non-deleting graph transformations (GT), where the LHS of the GT-rule is defined by the query itself and the creation rules are captured by its annotations.

Note that unlike bidirectional model synchronization approaches like triple graph grammars [Lau+12; GW09], derivation rules define a unidirectional transformation from the source model to the view model, where model changes are also propagated only in this direction but not vice versa.

A derivation rule is obtained when a query is extended by one or more of the following annotations:

- `@QueryBasedObject(eClass = "ClassName")`:
  The `QueryBasedObject` annotation adds a new view object of type `ClassName` and creates a `Trace` element into the *Traceability* model where parameters of the pattern will be the source n-ary, and the created object will be in the target.
- `@QueryBasedFeature(src = source, trg = target, feature = "FeatureName")`:
  The `QueryBasedFeature` annotation, as defined in [RHV12], is used to define links and slots initialized by queries. It sets the pattern parameter `trg` as the value of `FeatureName` feature of the pattern parameter `src` where `src` has to be a view model object.

Additionally, to support the explicit traceability, another annotation is defined to select the corresponding elements from the created view model.

- `@TraceLookup(src = source1, source2, trg = target, type = "TargetClass")`
  The `TraceLookup` annotation is used to declare a new variable with its "trg" slot to use in other annotations, find the proper target element in the view model which is created from the collection of "src", and select the target elements of type "TargetClass". (Note that the type is only an optional filter.)

Example 28 In the example, queries are annotated in the VIATRA Query syntax as follows:

```
1  // Dataflow View
2  @QueryBasedObject(eClass = "InformationType")
3  pattern src(type) {...}
4  @QueryBasedObject(eClass = "Host")
5  pattern trg(host) {...}
6
7  @QueryBasedFeature(src = type, trg = host, feature = "dataflow")
8  @TraceLookup(src = {type}, trg = t, type = "InformationType")
9  @TraceLookup(src = {host}, trg = h, type = "Host")
10 pattern dataflow(type,host) {...}
11
12 // Event Ordering View
13 @QueryBasedObject(eClass = "Init")
14 pattern init(t) {...}
15 @QueryBasedObject(eClass = "Activity")
16 pattern action(t) {...}
17 @QueryBasedObject(eClass = "Finish")
```

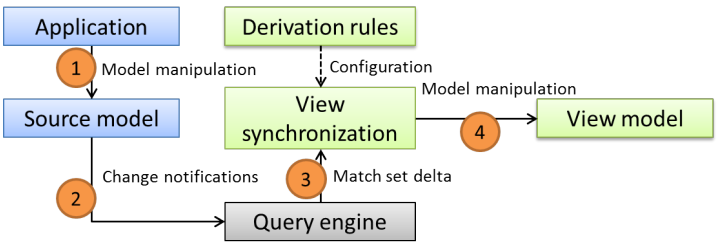


Figure 6.2: Overview of integration architecture

```
18 pattern finish(t) {...}
19
20 @QueryBasedFeature(src = pre, trg = post, feature = "after")
21 @TraceLookup(src = {a}, trg = pre, type = "Object")
22 @TraceLookup(src = {b}, trg = post, type = "Object")
23 pattern after(a,b) {...}
```

Listing 6.1: Example Pattern with Annotations syntax

6.5 Updating view models by incremental query evaluation

Now we outline the mechanism of handling changes in the source models in accordance with **G2** and show how derivation rules are used to update the target model continuously.

6.5.1 Incremental evaluation of queries

The key to update any target model synchronously is incremental query evaluation, where query result changes caused by model manipulation are also provided without complete reevaluation. We use the VIATRA Query framework [Ujh+15] that supports incremental query evaluation over EMF instance models by constructing a *Rete* rule network that processes *change notifications* sent by the EMF notification API. This Rete network remains in operation as long as the query is needed: it continues to receive elementary change notifications and propagates them to produce *query result deltas*. Even though this imposes a slight performance overhead on model manipulation, and a memory cost proportional to the cache size (approx. the size of match sets), VIATRA Querycan evaluate very complex queries over large instance models very efficiently.

The framework also provides an interface to notify any observer object when a new match appears or an old one disappears. With this functionality, our query-based approach has the ability to react on these events by firing all the required derivation rules.

6.5.2 Integration architecture of view synchronization

The view models are constructed and updated incrementally by translating the match set deltas from the incremental query evaluation into model modifications. The integration architecture for our framework is shown in Figure 6.2. After an initial `configuration` of the `view synchronization` using the `derivation rules`, the `source model` is loaded by EMF and the initial query results from the `query engine` are used for constructing the `view model`.

Once the view model is created, the synchronization is a reactive process with the following main steps: (1) `model manipulations` are carried out by an `application` (e.g. graphical editor or

user interface) on the source model, (2) these changes are propagated to the query engine as `change notifications`, (3) a `match set delta` is sent to the view synchronization, and finally, (4) `model manipulation` is performed on the view model based on the derivation rules.

6.5.3 From match set changes to view model synchronization

Initial setup of derivation rules The view models are created entirely based on the derivation rules specified as model queries with annotations (see Section 6.4.3). Additionally, the metamodels for view models are also selected. These metamodels must include the classes and feature types (attributes and references) defined by the derivation rules.

The view synchronization groups the derivation rules into two sets, based on the annotation type (query-based object, query-based feature). For each rule, a matcher for the model query is initialized in the query engine and handlers are attached to the matcher to receive deltas. These handlers are responsible for modifying the objects or feature settings in the view model.

Query result deltas As described in Section 6.5.1, the Rete network processes change notifications and provides query result deltas, that can be considered as higher level notifications. A delta is a structure $Delta = (Found, Lost)$, where $Found$ is the set of new matches that appeared and $Lost$ is the set of old matches that disappeared since the last delta.

The initial construction is executed similarly as incremental updates are handled, since the initial query results can be considered as a delta containing the initial matches in $Found$.

Source-view traceability During the synchronization process, internal traceability links are stored for identifying the view model elements corresponding to appearing and disappearing matches in deltas.

The `traceability` structures are created for each view model object created by object derivation rules. The traceability structure stores the `source` match for the `view` model element. This explicit traceability model is managed by the view model synchronization and is used for finding the container and target objects of view model links and slots.

Example 29 The `query results` contain all matches for each model query that is used by derivation rules in the example. The matches are tuples of pattern parameter assignments, where each assignment is a direct link to an object in the source model, or a slot value.

Between *dataflow view* and source model, the following traceability is created: $\langle \text{PM}_{src}\langle \texttt{pulse} \rangle, \texttt{pulse} \rangle$, $\langle \text{PM}_{trg}\langle \texttt{emergency} \rangle, \texttt{emergency} \rangle$, $\langle \text{PM}_{dataflow}\langle \texttt{pulse} \rangle, \texttt{emergency} \rangle$, $\langle \text{PM}_{src}\langle \texttt{pressure} \rangle, \texttt{pressure} \rangle$, $\langle \text{PM}_{trg}\langle \texttt{gp} \rangle, \texttt{gp} \rangle$, $\langle \text{PM}_{dataflow}\langle \texttt{pressure} \rangle, \texttt{gp} \rangle$,

Similarly, between *event ordering view* and the source model, the following structure is constructed: $\langle \text{PM}_{init}\langle \texttt{daily} \rangle, \texttt{:Init} \rangle$, $\langle \text{PM}_{finish}\langle \texttt{daily} \rangle, \texttt{:Finish} \rangle$, $\langle \text{PM}_{activity}\langle \texttt{pulseMeasure} \rangle, \texttt{pulseMeasure} \rangle$, $\langle \text{PM}_{activity}\langle \texttt{pressureMeasure} \rangle, \texttt{pressureMeasure} \rangle$, $\langle \text{PM}_{activity}\langle \texttt{pulseReport} \rangle, \texttt{pulseReport} \rangle$, $\langle \text{PM}_{activity}\langle \texttt{pressureReport} \rangle, \texttt{pressureReport} \rangle$,

Incremental view maintenance in the example In this short paragraph, we describe how the incremental view maintenance behaves on the motivating example.

Example 30 When the new `Measure` object with its links are added to the source model the following steps are performed: (1) a delta is received with $Found = \text{PM}_{dataflow}\langle pulse, gp \rangle$ for

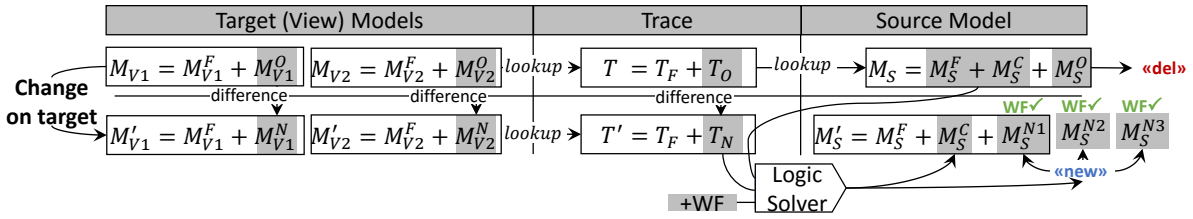


Figure 6.3: Overview of backward change propagation

the `dataflow` query, (2) the handler of the feature derivation rule creates a `dataflow` link between `pulse` and gp in the view model.

Similarly, when the `pulseReport` object is deleted from the source model the following steps are performed: (1) a delta is received with $Lost = \text{PM}_{action}\langle pulseReport\rangle$ for the `action` query, (2) the handler of the object derivation rule deletes the `Trace` from the traceability model and (3) `removes` the related `Activity` from the view model.

Derivation rule priority Since the traceability links are used for identifying view elements and their corresponding matches in the delta, the order of handler execution is not arbitrary. For *Found* matches, object derivation rules are executed first, even if there are link and slot rules defined for the same query. Links and slots are set after the objects are created. However, for *Lost* matches, links and slots are handled before objects to ensure that the source and target objects are still available. Derivation rules of a given query can depend on objects created by rules for other queries, therefore deltas are not handled separately, but combined based on both rule types (query-based object and feature) and events (found, lost).

6.6 Backward Change Propagation by Logic Solvers

6.6.1 Overview of the Approach

To address **G3**, we present a novel approach to back-propagate view model changes into a consistent source model by using logic solvers. An overview of our approach is depicted in Figure 6.3 where *target (view) models* $M_{V1}, M_{V2}, \ldots$ are derived from a *source model* (M_S) based on the matches of the view definition queries (in the source model), and a *traceability model* T is built and maintained during the forward transformation. Now the engineer makes changes to the view models, which leads to changed view models $M'_{V1}, M'_{V2}, \ldots$. The goal of our approach is to calculate (one or more) source model $M_S{}'$ which corresponds to the change, maintain T', and *satisfy additional constraints of the source model.*

A change in the view model can be separated into two partitions: the fixed model partition M^F_{Vi} denotes a partial model which remains unchanged, while M^O_{Vi} is updated to a new M^N_{Vi} (cross-links are included in M^O_{Vi} and M^N_{Vi}). The change is propagated consequently to the trace model: T_O contains the invalidated trace links, T_N symbolizes the new links to be created and T_F contains the remaining trace links which are not affected by the change. Along the traces, the change can be propagated back to the source model by identifying unchanged, newly activated and deactivated matches of queries in the source model.By analyzing the impact of changing matches in T_N, the source model can be partitioned into three partial models: a fix part ${M_S}^F$ contains the elements which cannot be changed, a changing part ${M_S}^C$ contains the object which can be modified, and a obsolete part ${M_S}^O$ which

objects can be deleted. The changing trace T_N declaratively specifies structural constraints on the ${M_S}^C \cup {M_S}^O$ model which have to be satisfied in order to ensure the consistency of the forward transformation.

A possible solution M'_S for the changed parts ${M_S}^C \cup {M_S}^{Ni}$ has to (1) match the fixed part M^F_{Vi}, (2) the requirements defined by the changed matches defined in T_N, and (3) additional domain-specific *WF* (of the source model). All these constraints are transformed into a first-order logic problem to be solved by a *logic (SAT/SMT) solver* following [SVV16]. The solver provides several (but not necessarily all) possible valid solutions for ${M_S}^C$ and it may create new object in ${M_S}^{Ni}$ from which from the model developer may choose the most appropriate one ${M_S}^{Ni}$.

As a summary, our approach integrates two novel techniques into an interactive workflow: an inverse impact analysis by *change partitioning* (subsection 6.6.2) separates the affected and unaffected parts of the source model, while *partial model generation* creates candidate source models that correspond to the new target views (subsection 6.6.3) and source constraints.

6.6.2 Change Partitioning

The rationale of change partitioning in accordance with **G4** is as follows: (i) if a view model element does not change, the associated traces cannot be changed; (ii) otherwise, if the change of a source object may induce a valid view model it has to be selected. However (iii) unnecessary source changes should not happen. To ensure these conditions, an impact analysis of the changes has to be conducted to identify the affected part of source model which can be modified.

6.6.2.1 Affected parts of view model changes

Table 6.2 contains the calculation of affected parts in case of modifying a view model M_V. When a view object o is removed, the affected parts are determined by its defining pattern p and its activation m that is stored in the trace model T. When a link ref is removed from M_V, the affected part of the source o_s and the target o_t are returned. When a new view element is created, then there are obviously no activations of forward rules.

6.6.2.2 Affected parts of pattern activations

A pattern predicate p with its activation m marks the union of the bodies defined in Table 6.3. Each body consists of several conditions $const$ that may introduce additional internal variables $Params_i$. Hence, an activation of a $body$ is extended with m_i all the possible bindings of internal variables. The affected part of a body $body$ is the union of the affected parts of each constraints $const$.

6.6.2.3 Affected parts of source constraints

Affected parts of source constraints are defined in Table 6.4. A *class* (class) condition returns the object that is bound to its parameter along activation m. Similarly, *slot*(attr) and *link*(ref) conditions (together feature conditions *feature*(feat) select respective parameters (x and both x, y) from m. A *path*($\text{feat}_1 \ldots \text{feat}_n$) condition can be split into several *feature*(feat) to calculate its affected part. For *equal*($=$) and *not equal*($\neq$), the bound objects of parameters from both side of the operators are returned.

A pattern p may call another pattern p' (*find*$[p']$ or transitively with *find*+$[p']$) by mapping symbolic parameters $Params_{p'}$ of the called pattern to concrete values of the caller. Thus given a

$$\begin{array}{c} -\mathsf{class}(o) \rightarrow \mathsf{affected}(p(m)) : lookup_{VS}(o) = p(m) \\ \hline -\mathsf{ref}(o_1, o_2) \rightarrow \mathsf{affected}(p(m)) : lookup_{T}(o_1, o_2) = p(m) \\ \hline -\mathsf{attr}(o, val) \rightarrow \mathsf{affected}(p(m)) : lookup_{T}(o_1, o_2) = p(m) \end{array}$$

Table 6.2: Affected changes of view model

$$\begin{array}{rl} p(m) & \rightarrow \bigcup \mathsf{affected}(body_i, m) \\ \hline body(m) & \rightarrow \bigcup \mathsf{affected}(cond_i(m + m_i)) \end{array}$$

Table 6.3: Affected activations

$$\begin{array}{r} \mathsf{class}[obj], (m) \rightarrow \{o_{obj} | m(obj) = o_{obj}\} \\ \hline \mathsf{attr}[x, val], (m) \rightarrow \{o_x | m(x) = o_x\} \\ \hline \mathsf{ref}[x, y], (m) \rightarrow \{o_x, o_y | m(x) = o_x, m(y) = o_y\} \\ \hline \mathsf{feat}(m) \rightarrow \begin{cases} \mathsf{affected}(\mathsf{attr}(m)) \text{ if feat is attr} \\ \mathsf{affected}(\mathsf{ref}(m)) \text{ if feat is ref} \end{cases} \\ \hline \mathsf{feat}_1 \ldots \mathsf{feat}_n[x, y](m) \rightarrow \bigcup \mathsf{affected}(\mathsf{feat}_i(m)) \\ \hline \begin{array}{r} x = y, (m) \\ x \neq y, (m) \end{array} \rightarrow \{o_x, o_y | m(x) = o_x, m(y) = o_y\} \\ \hline \mathit{find}[p'](m) \rightarrow \mathsf{affected}(p'(m')), m' \circ m \\ \hline \mathit{find}^+[p'](m) \rightarrow \bigcup \mathsf{affected}(p'(m')), m' \circ m \\ \hline \mathit{neg\ find}[p'](m) \rightarrow \begin{cases} \{o_x | m(x) = o_x, \}, \text{ if no inner var} \\ \{o_i | o_i.type \in \text{inner types of } p'\} \end{cases} \end{array}$$

Table 6.4: Affected changes of source constraints

binding : $Params_{p'} \rightarrow Params_p$, a match m' is composed from m by getting objects from the original activation m, formally:

$$m' \circ m \Rightarrow m' : m(\mathsf{binding}(var')) \rightarrow O_S, var' \in Params'_{p'}$$

However, a *negative application condition* ($neg\ \ find[p']$) is separated into two cases: if the sub predicate p' does not introduce any internal variable, the affected part returns the linkd objects from the activation m. Otherwise, we restrict the affected part to all the objects that has the same type as the introduced internal variables have.

6.6.2.4 Categorization of affected source model objects

Definition 6.4 (Categories of Affected Objects). The affected objects of the source model can be categorized into three groups:

- ${M_S}^F$: *neither changeable nor removable*: It includes all objects of M_S which are not in the affected part of the change from the view Δ_{view}.

$$ {M_S}^F = M_S - \mathsf{affected}(\Delta_{view}) $$

- ${M_S}^C$: *changeable but non-removable objects*: It includes all objects in the affected part of the change from the view Δ_{view} which are responsible for the existence of other activations.

$$ {M_S}^C = \mathsf{affected}(\Delta_{view}) - \{o | o \text{ referred by } T_{obj}\} $$

- ${M_S}^O$: *changeable and removable objects*: All objects in the affected part of the change from the view Δ_{view} which are not responsible for the existence of any other activations.

$$ {M_S}^O = \mathsf{affected}(\Delta_{view}) - {M_S}^C $$

.

Example 31 In our example of Figure 6.1, the affected part for the deletion of `dataflow` edge from the view model is calculated as follows. The trace model T_F stores that the existence of `dataflow` edge is related to the dataflow pattern where the activation binds pattern parameters `emergency:Host` and `pulse:MeasurementType` to objects in the source model M_S. The affected part of the pattern includes all objects related to the match {*emergency,pulse*}, and the affected part of the constraints includes internal variables {*pulseReport, pulseMeasure*}. However, *pulseMeasure* is also responsible for an after edge in the other view model, thus it can be changed but not allowed to be deleted from the source model. At this stage, the user may manually move objects between these categories (${M_S}^F$,${M_S}^C$,${M_S}^O$) to refine his/her intention on source candidates.

$$ {M_S}^C = \{pulseMeasure\} $$
$$ {M_S}^O = \{emergency, pulse, pulseReport\} $$
$$ {M_S}^F = \{\text{rest of the objects}\} $$

6.6.3 Model Generation by Logic Solvers

Logic solver based model generation for a domain specific language is an actively researched area. Instance models can be created to provide models that satisfies required properties, test cases or to create counterexamples for false language properties [Sem+15], and incremental model generation techniques [SVV16; SFC12] are able to take advantage nearly finished partial instance models.

6.6.3.1 Logic Representation of View Models

In general, solver-based model generation takes the logic representation of the metamodel $Meta$ to synthesize conforming instance models M with $M \models Meta$. Each class C_i is represented by a predi-

cate over the objects of the model $C_i \subseteq o_M$, each link R_i is mapped to a relation over pairs of objects $R_i \subseteq o_M \times o_M$, and slots are modeled by $A_i \subseteq o_M \times Type_i$, where $Type_i$ is the domain of slot i. Additionally, $Meta$ contains basic structural constraints of the metamodel as axioms (e.g. multiplicity and containment hierarchy, see complete [Sem+15] for full details), so the interpretation of relations represents structurally correct models. The logic problem can extended to include well-formedness constraints WF [Sem+15] defined either as graph patterns or OCL invariants to generate valid solutions.

A model query P_i with can be represented as a predicate over objects of the target model $P_i : o_M \times \ldots \times o_M \rightarrow \{true, false\}$ which is evaluated to true only if some objects satisfy the translated query specification. A match m_i is represented as a tuple of objects: $m_i = (c_i^1, \ldots, c_i^n)$, where $c_i^j \in o_M$. Pattern matches are controlled by two formulae: (1) a pattern P_i has a match $m_i = (c_i^1, \ldots, c_i^n)$ if and only if the constants satisfies the associated logic predicate $P_i(c_i^1, \ldots, c_i^n)$; (2) each match of a pattern has to be unique: for all matches m_i and m_j there is at difference $\exists_{x \in 1..n}(c_i^x \neq c_i^x)$. Consistency with the view model is ensured by a formula set $View$, which controls the matches of query predicates. Therefore, the generation of a valid and consistent view model is specified as $M \models Meta \wedge WF \wedge View$.

Example 32 In the logic equivalent of our running example the `type`, `what` and `where` links are modeled by relations R_{type}, R_{what}, R_{where}. The specification of `dataflow(t,h)` pattern can be represented by the following predicate:

$$P_{DF}(t, h) \Leftrightarrow \exists i_1, i_2[R_{type}(i_1, t) \wedge R_{what}(i_2, i_1) \wedge R_{where}(i_2, h)].$$

Here t and h has to be connected by a specific path of relations. In our example the changed dataflow model has two matches: $m_1 = (c_1^1, c_1^2)$ and $m_2 = (c_2^1, c_2^2)$, where $c_1^2 = c_2^2$ (as both types are forwarded to the general practitioner). With the following axioms added to the logic problem it can be ensured that there are exactly two matches of the pattern (as defined by the dataflow view), and each match is unique:

$$\forall h, t[P_{DF}(t, h) \Leftrightarrow (t = c_1^1 \wedge h = c_1^2) \vee (t = c_2^1 \wedge h = c_2^2)]$$

$$(c_1^1 = c_2^1 \vee c_1^2 = c_2^2)$$

6.6.3.2 Incremental Transformation of View Models

In case of view models, the affected part ${M_S}^C \cup {M_S}^O$ typically remains proportional to the change, thus ${M_S}^F$ explicitly defines most of the generated models. Incremental model generation techniques like [SVV16] are able to take advantage of fully specified model fragments, and encode the graph generation problem in a way that the problem is proportional to the newly created fragment. In the following, we give a brief description of the mapping technique.

- **Objects**: the object set is partitioned into three subsets: ${M_S}^F$ is mapped to the fix objects O_F, ${M_S}^C$ stands for the changing objects and finally O_N replaces the objects which are removed ${M_S}^O$. In general, predicates dealing with ${M_S}^F$ are interpreted, thus the solver already knows its truth evaluation.
- **Classes**: Each class predicate C is also separated into three subsets: a fully interpreted C_F defined over O_F, a fully interpreted C_C defined on the changing objects O_C, and the uninterpreted C_N over O_N. In summary, the solver has to interpret only relations of C_N.

- **References**: Reference predicates are also separated to multiple smaller relations: R_{OO} represents the interpreted relation between fixed objects, R_CC the link between changing objects, and R_{NN} represents the link between new objects. Additionally, uninterpreted cross-links have to be added for links connecting these regions: $R_{OC}, R_{ON}, R_{CO}, R_{CN}, R_{NO}, R_{NC}$. While only R_{OO} is interpreted from the nine new relations, it contains the most links.
- **Attributes**: Attribute predicates are also separated into three partitions for O_F, O_C and O_N.
- **Model Queries and Matches**: Model queries are separated into multiple queries, each parameter can be bound to O_F, O_C and O_N. This might add several relations to the logic problem, but the unchanged matches are already interpreted. In the construction of the constraint set $View$, the uniqueness of non-interpreted matches needs to be ensured

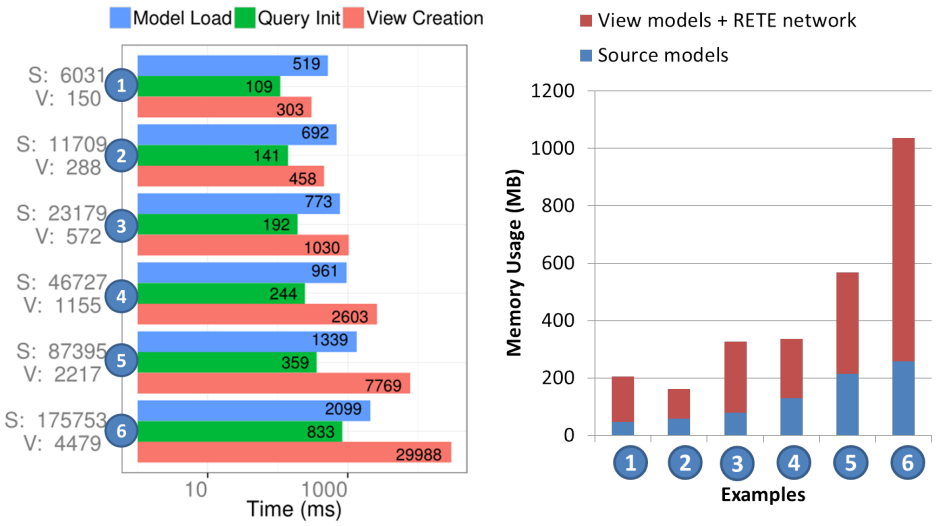
Compared to solving the model generation problem as a whole, our incremental approach enables the logic solver to handle much fewer variables as a large fragment of the model is already interpreted (prior to calling the solver). The downside is that constraints become more complex as they have to be separated into those groups above. However, we expect that most predicates remain interpreted, which is beneficial for the solver.

6.6.4 Properties of Our Approach

Our approach has the following properties (based on [Hid+15]):

1. *Full operation support on views:* View models can be edited as regular models, while the technique ensures consistency between source and view models.
2. *Implicit backward consistency:* View models derived from a source model candidate $M'_S = \text{fwd}^i(S')$ are isomorphic to the view models $V'_1, \dots V'_n$.
3. *Delta-based and offline back-propagation:* After making changes on the target models, our approach generates source model candidates from a stable state of the views and the changes in the traces. Upon a sequence of (possibly concurrent) view changes is applied it leads the view models to a new stable state. Then the difference between the previous and current state of the views can be propagated back even if some changes are contradictory or inconsistent by themselves.
4. *Interactive execution:* There might be several source candidates for a view model change on which the solver can iterate, starting from the smallest solution. The developer or a selection strategy can select the most suitable one from the sequence of valid solutions.
5. *Well-formedness:* S' satisfies the well-formedness constraints of the source domain $S' \models WF$.
6. *Incrementality:* A view change can affect only the affected part of the source model. Additionally, if a view model element is not changed, it should not be changed by the effect of back propagation (even if the result would create isomorphic). Thus all matches m which $T \models (p(m), v)$ and $v \in M'_V$ should remain in $T' \models (p(m), v)$.
7. *Hippocraticness of unaffected partition:* If a view model element is not changed, the associated source model part has to remain unchanged.

Conditions 1-4 are related to usability and they connect the new backward propagation technique to our previous forward transformation approach. Some form of consistency is ensured in several approaches (e.g. [Cic+10]) but we also incorporate WF constraints of the source language. Hippocratic behavior defined in [Ste08] states that a backward or forward transformation must not modify the source or the target model if they are already consistent. In Property 6. and 7. we define a stronger requirement which states that consistent partitions of the source and target models should not be modified. This constraint simultaneously keeps most of the source model untouched and makes the deduction phase more efficient by limiting the task to partial models.



(a) Runtime performance

(b) Memory Usage

Figure 6.4: Measurement Results

6.7 Evaluation

6.7.1 Performance evaluation of forward synchronization

In order to evaluate the scalability aspect of the forward synchronization approach, we have carried out an initial evaluation to demonstrate the scalability of our approach on the Train benchmark case study [Ujh+15] (an accepted model querying performance benchmark). It was selected as in our industrial case study we only had moderate size models containing only a few hundred elements and their execution in all cases were smaller than 500 milliseconds.

Our measurements aim to address the following questions:

Q1 *What is the influence of the size of the source model on the runtime of the forward synchronization?*

Q2 *What is the influence of the size of the source model on the memory usage of the forward synchronization?*

6.7.1.1 Measurement Setup

Based on the Train benchmark, we defined 3 queries including advanced features such as transitive closure computation to define the view models. For the different cases we selected 6 model sizes for the source model ranging between 6000–175000 elements that produced view models of 150–4500 elements.[1]

Hardware Configuration All measurements were executed on a developer PC with a 3.5 GHz Core i7 processor, 16 GB RAM, Windows 7 and Java 7. Measurements were repeated ten times, and the time and memory results were averaged (since the recorded deviation was negligible). To avoid interference between different executions, each time a new JVM was created with a 2 GB heap limit. The time to start up and shut down the JVM was not included in the measurement results.

[1]More details are available at https://incquery.net/publications/query-based-synchronization

6.7.1.2 Measurement Result

Runtime results To address **Q1**, 6.4(a) depicts the runtime results: each row shows the time required to (1) load a source model, (2) initialize the query engine and (3) derive the view model. Additionally, for each instance model, the number of model elements of the source (S) and view (V) models are presented.

Memory usage results To address **Q2**, 6.4(b) highlights the overall memory usage for the different model instances, where the blue section shows the memory consumption of the source model alone and the (upper) red part depicts the memory needed for both the Rete network of the cached queries and the generated view-models. As expected the overall memory consumption for the view models are 2-3 times larger than the source models themselves due to the Rete caching mechanism [Ujh+15] and the quadratic nature of transitive closure. However, this intensive caching provides the fast incremental update in case of small source model changes [Ujh+15].

To sum up, we were able to demonstrate that our view model based approach is capable of scaling up to source models with more than 100 000 elements within acceptable timeframe and memory consumption, which is in line with our previous experiments [Ujh+15].

6.7.2 Experimental evaluation of backward propagation

In order to evaluate the performance of the key step of our backward change propagation technique we have conducted initial measurements on a prototype implementation in the context of the running example as case study (taken from the CONCERTO project). The measurement scenarios and the results are available on GitHub[2]. Our measurements aim to address the following questions:

- Q1 *What is the influence of the size of the source model, the size of the target change and the scope of the source model (i.e. the number of newly created source objects) on the runtime of the solver?*
- Q2 *What is the difference between incremental model generation and full model generation (like in [Gho+15]) with respect to performance and the quality of results?*
- Q3 *What are the (solver-specific) limitations of our backward change propagation technique?*

Our evaluation exclusively focuses on assessing the performance of the model generation step for the source model, and excludes the performance evaluation of change partitioning. In our initial experiments, we experienced that both the change partitioning time (i.e. the selection of affected source elements) and the forward transformation time is negligible (less than 1 second for the largest problems we measured) compared to the time required to solve the logic problem by Alloy. Thus performance limitations are dominated by the latter.

6.7.2.1 Change propagation problem generator

In order to measure the performance of our technique we extended the running example visible in Figure 6.1 to a change propagation benchmark, which can be parametrized and scaled by the *size (s)* of the source model and *the number of changes (c)* in the views. We have created valid health care models illustrated in Figure 6.5 using a *change propagation problem generator* discussed in [c15].

As a result, we obtain non-trivial source and view models, while the random changes of the view model remain semantically meaningful.

[2] https://github.com/FTSRG/publication-pages/wiki/incremental_backward_change_propagation_of_view_models_by_logic_solvers

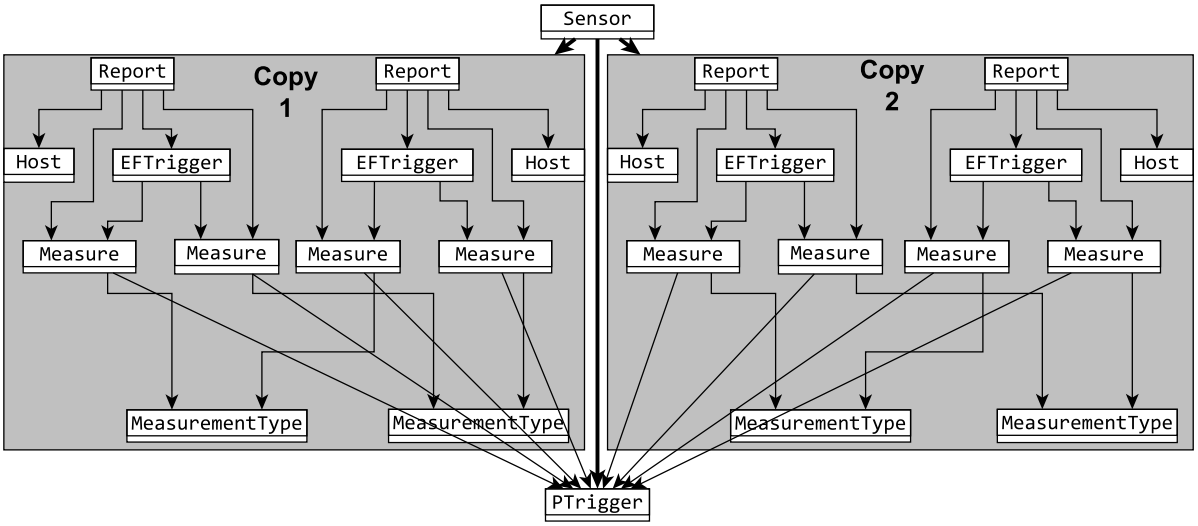


Figure 6.5: Structure of the generated source models

6.7.2.2 Measurement Setup

Each model generation task was executed on the generated healthcare change propagation problems using the Alloy Analyzer (with SAT4j-solver). Each change propagation problem is solved with our incremental solution which generates only the affected part in the source model. As a baseline, we compare it to a solution conceptually similar to [Gho+15] which generates full models for a view model. The full model generation is achieved as a corner case of the incremental generation where the changed view models are interpreted as if they were newly created, and no part of the source model is preserved.

We measured the runtime of Alloy Analyzer, which consists of an initial conversion to a conjunctive normal form and then solving the SAT-problem by the back-end solver. We executed each measurement 5 times, then the average of the execution times was calculated. The measurements were executed with a 120 second timeout on an average personal computer[3]. The memory usage of the solver was always below 2 GB.

6.7.2.3 Measurement Result

The execution times of our measurements (see Figure 6.6– Figure 6.9) are given in seconds.

- $|S|$ denotes the number of objects in the original source model (M_S);
- $|\Delta V|$ denotes the size of the target change, i.e. the sum of removed and newly created matches in the view model (M_V^O and M_V^N);
- Finally, $|N|$ denotes the source scope, i.e. the number of new objects in the changed source model candidates (the number of objects in ${M_S}^N$). It is equivalent to the scope of model generation in Alloy when incremental technique is used.

In case of full model generation where the each model object is newly created, the size of the original model is subtracted from this value, so the two techniques are comparable with respect to the number of objects.

[3]CPU: Intel Core-i5-m310M, MEM: 16GB, OS: Windows 10, Reasoner: Alloy Analyzer 4.2 with sat4j

6.7.2.4 Increasing model size

First, Figure 6.6 displays the runtime results of cases where only one change is performed ($c = 1$, $\Delta V = 8..9$). Eleven different series are measured, where the change propagation is solved with source scope of $|N|$ new elements (ranging from 0 to 10) with increasing the size of source models $|S|$ up to 123 objects. With 0 new elements, the problem was unsatisfiable, otherwise valid solutions were created.

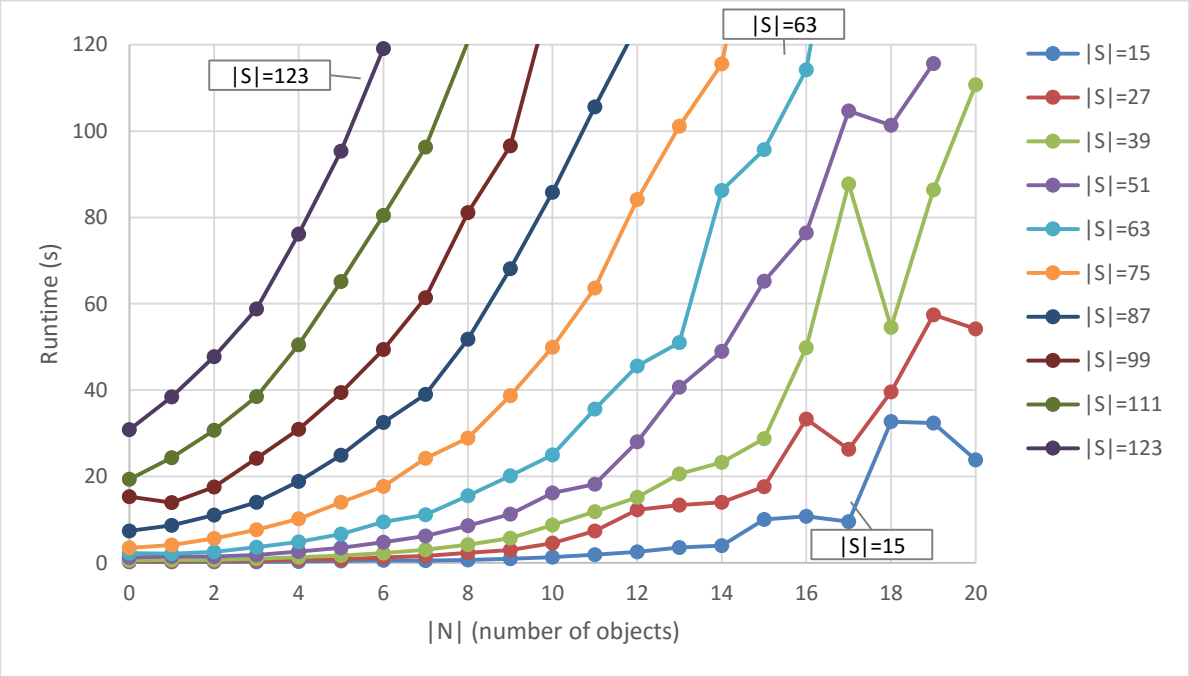


Figure 6.6: Runtime with increasing source model size (vertical axis: size of source model, series: size of source scope, change size: 1)

The results show a polynomial (cubic) increase of run-times in the size of the original source model, and it always harder to solve the problem with increasing source scope size. Additionally, the solver has a similar characteristics regardless a problem is satisfiable or unsatisfiable. Fortunately, smaller solutions can be retrieved more efficiently, which is typically preferred over solutions with unnecessary elements.

We also depicted the run-times in Figure 6.7 while further increasing the size of source scope. This shows that the source scope needs to be decreased when the size of source models increases in order to obtain the same run-time.

6.7.2.5 Increasing target change size

Figure 6.8 displays the results with larger target change sizes. Here, the vertical axis displays the source scope, i.e the number of new objects added to the source model by the solver, and different series shows the run-times of target change size ranging from 8 up to 43 changes, and the measurement is repeated for two model sizes. For the smallest change one new element is needed to successfully create solutions, two new object is needed for the next change size, and so on, so the largest changes needed at least 5 new objects.

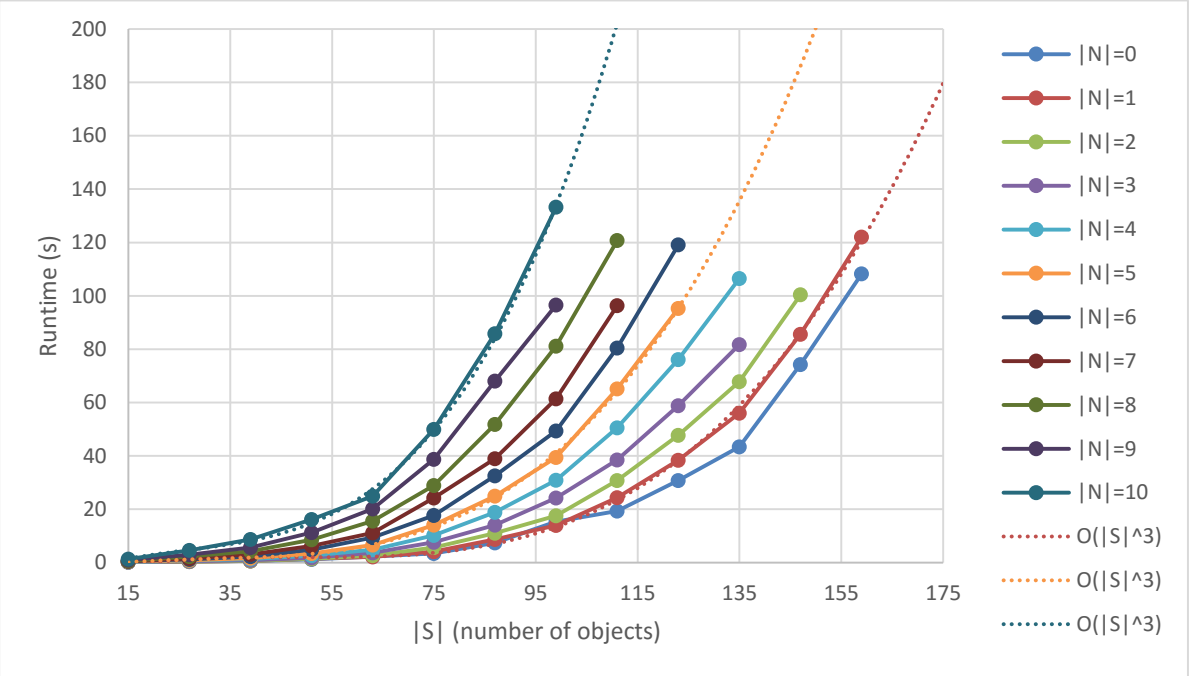


Figure 6.7: Runtime with increasing source scope size (vertical axis: size of source scope, series: size of source model, change size: 1)

The results shows that the run-time is not directly affected by the size of the change, different change sizes have the same complexity. However, a larger target change size usually implies a larger source scope, which, of course, influences the run-time of the solver. In other terms, if the consequences of a large target changes are attempted to be propagated back to the source model with a small scope size, then the logic solver will conclude that the problem is unsatisfiable. While if we also increase the scope size, then the problem might become satisfiable - but the logic solver may fail to find a solution due to its performance limits.

Therefore, based on our measurements of increasing model and change size, our first question can be addressed as follows:

A1 *The runtime of the solver is a polynomial function of the original size of the model and the source scope size (i.e number of newly added elements). The size of the target change increases the size of the source scope required for a solution.*

6.7.2.6 Incremental vs. full generation

In Figure 6.9 the runtime of the incremental and full model generation is depicted with respect to the size of the source model size and the source scope. Only results of two small source models (15 and 27) are presented as on all larger cases, the full model generation technique was unable to provide a solution.

In comparison, the incremental model generation technique performed much better: it was able to solve some nontrivial problems, and in general, it was orders of magnitude faster than full generation.

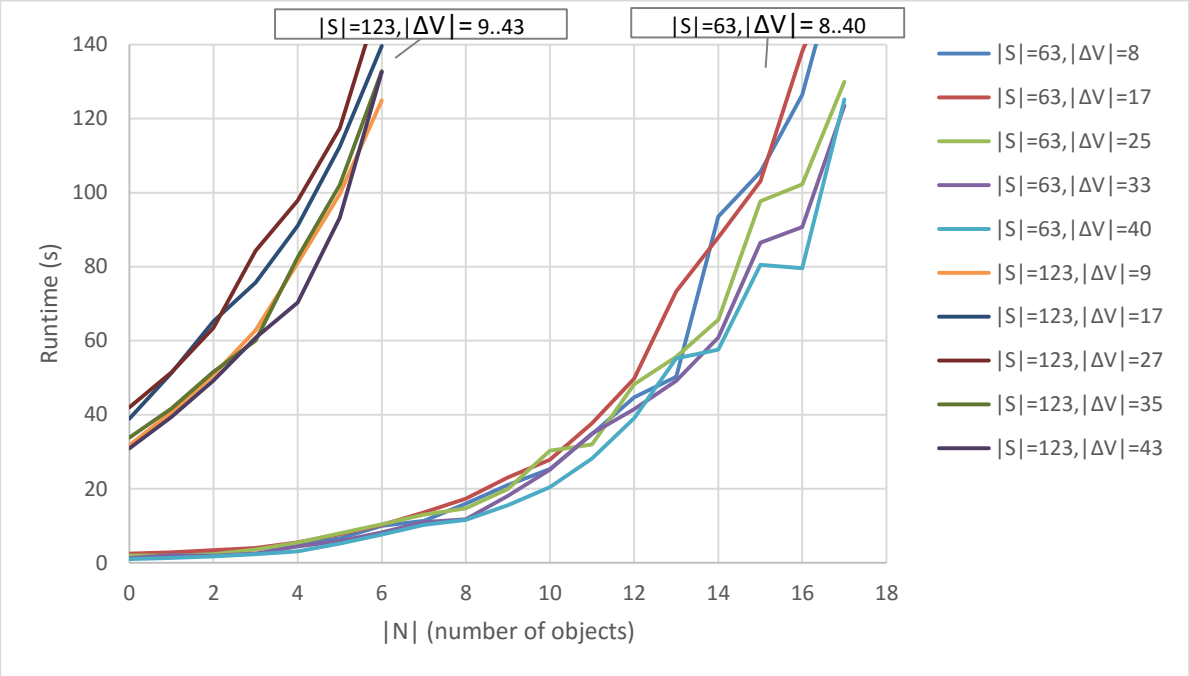


Figure 6.8: Runtime with different change size (vertical axis: size of source scope, series: size of model, size of change)

From the perspective of model quality, the full model generation approach redirected several relations where the target view model was unchanged (e.g. unchanged dataflows), which might be undesired in change propagation scenarios. On the other hand, the full change propagation may find solutions which requires changes in partitions categorized as unaffected by our change partitioning, which can be only retrieved by manual configuration of the affected part in case of incremental generation.

The difference between incremental and full model generation for backward change propagation purposes can be summarized as follows:

A2 *Incremental model generation provides better performance for source models of increasing size for a given scope. Full model generation may be able to retrieve some hidden solutions. Unlike full model generation, incremental generation is able to detect if a change propagation setup is unsatisfiable for a given scope.*

6.7.2.7 Limitations

Our initial experimental results revealed several limitations of our approach most of which were caused by using Alloy as an underlying logic solver. When investigating the runtime for model generation, we found that over 80% of the time is spent for an initial conversion to Conjunctive Normal Form (CNF) prior to starting the model finding step. In fact, CNF conversion took over 99% of the time for large models with $|S| = 160$ elements. The largest source model our technique was successfully executed had 243 objects, and the run-time for propagating a single target change was over 20 minutes (again, spent dominantly on CNF conversion).

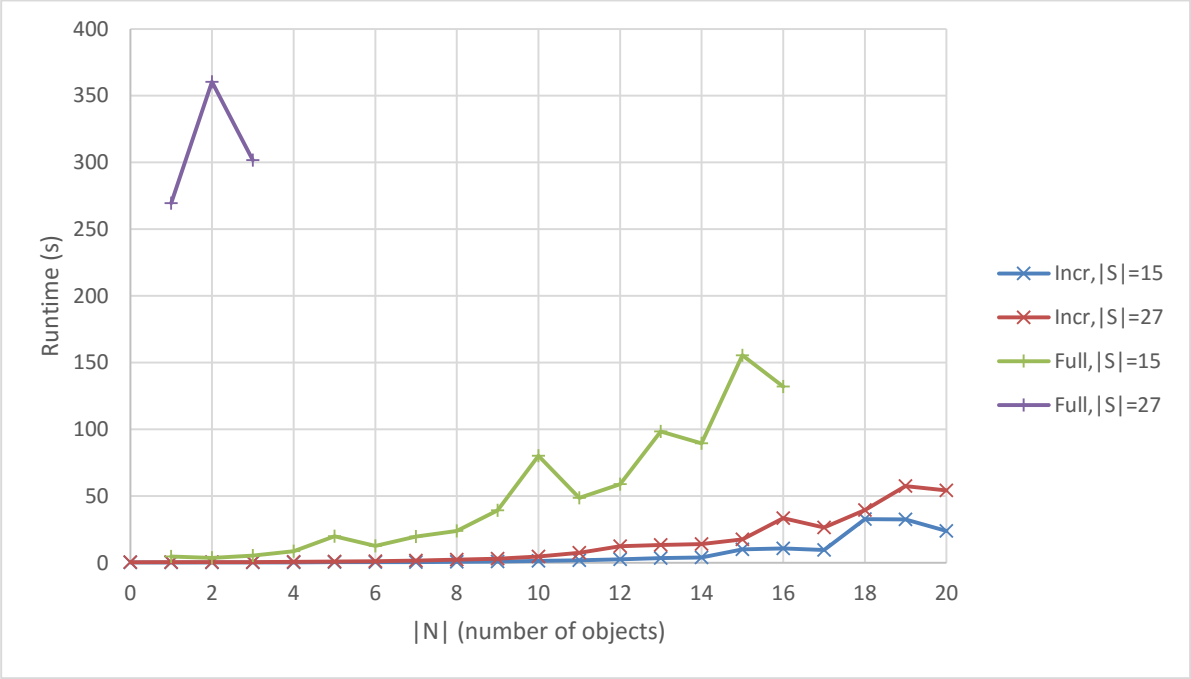


Figure 6.9: Incremental versus full model generation

While our incremental query-based forward transformation scales to source models orders of magnitude larger, we can state as a conclusion that such scalability cannot be achieved for incremental backward change propagation when using Alloy as a solver.

However, the fact that model generation time is dominantly spent on CNF conversion in Alloy (and not on the actual SAT/SMT-based model finding), this may trigger future research to replace Alloy with a dedicated incremental model finder for EMF-based models.

Furthermore, when investigating small backward change propagation problems with our approach, one may gain domain-specific insights to identify specific target changes which can be always mapped to a source change (e.g. by some rule-based transformation techniques).

A3 *Incremental backward change propagation using model generation by Alloy scales only to small models compared to incremental forward transformations. But an incremental model generation technique scales significantly better than full model generation, which is a hope for dedicated future model generators.*

6.7.2.8 Threats to Validity

Finally, let us investigate the most critical threats to validity of our conclusions.

- While our case study is relatively complex (as it originates from the CONCERTO project), our measurements were executed only on a single case study, thus our findings may not be applicable in a more general context. However, our model generator approach has strong roots in [SVV16] where efficiency of incremental model finding by using existing model solvers were assessed, and the experimental results point to similar limitations. Moreover, our negative results (e.g. poor performance of Alloy) are more likely generalizable for other model generation and transformation scenarios.

- We excluded the time of forward transformation and change partitioning from our measurements - as initial experiments showed that they were less than a second.
- As execution times of Alloy quickly started to increase, we only had 5 measurements for each case, thus we did not carry out a full-fledged statistical analysis of our results. Correspondingly, our findings are softer (more qualitative than quantitative).

6.7.3 VIATRA Viewers

"The goal of the VIATRA Viewers component is to help developing model-driven user interfaces by filling and updating model viewer results with the results of model queries. The implementation relies on (and is modeled after) the Event-driven Virtual Machine and JFace Viewers libraries. The VIATRA Viewers component can bind the results of queries to various viewer components such as JFace Viewers or GraphViewers."[VVi]

The implementation of VIATRA Viewers uses the proposed forward incremental synchronization approach. In this case, the view model is defined as a notion model consisting of *items*, *edges*, *containment edges* and *formatting settings*, while the source model can be any be EMF model.

As visualizing parts of EMF models are a recurring task, we have also defined a set of shorthands to make rule specification more concise. The (1) `@Item` rule creates an ITEM element from the Notation metamodel (depicted in Figure 6.10) with an attribute named LABEL. The (2) `@ContainsItem` and (3) `@Edge` rules create containment and non-containment references between selected model elements, respectively. Finally, the (4) `Format` annotation defines additional formatting rules to be attached as a FORMATSPECIFICATION instance to the current element.

Finally, the notational view model is used as the input of one of the available renderers that displays the model using e.g. JFace[JFace] viewers, Zest[Zest] or yFiles[YFiles] for Java graph visualization engines.

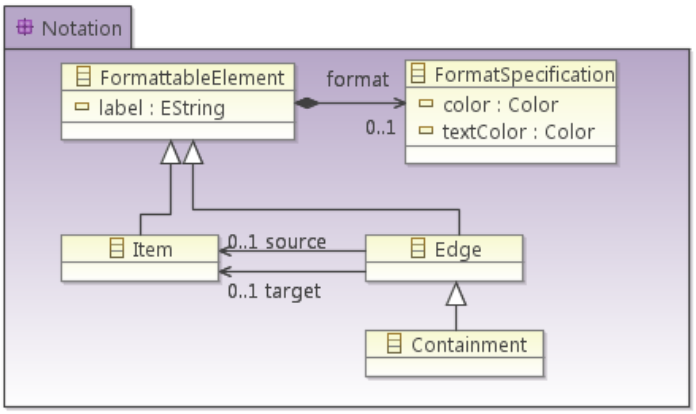


Figure 6.10: Notation metamodel for visualization

6.8 Contributions

In this chapter, we presented an approach for deriving and synchronizing non-persistent (virtual) view modes in an incremental (but unidirectional) way . Our approach supports the maintenance of

multiple views over the same source model, and also the chaining of view models along derivation rules. In order to automatically maintain and synchronize the view models, we extended the VIATRA Query framework with (1) the ability to define derivation rules using annotations over graph queries, (2) an explicit tracing mechanism and (3) an EVM-based runtime environment.

In addition, we presented an incremental backward change propagation approach from view models to source models which (1) provides a change partitioning technique to separate possibly affected and unaffected partitions of the source model, (2) transforms the source model partitions, the queries and WF constraints of the source language to a logic problem and (3) generates well-formed and consistent source model candidates by the Alloy Analyzer. This way, valid source candidates can be deduced in case of multiple view models and when backward transformation rules are not explicitly specified.

Contribution 4 I proposed a novel technique of bidirectional synchronization of view models where the forward incremental synchronization is achieved by unidirectional derivation rules while the backward propagation of changes is generated using logic solvers. [*c*15], [*c*11], [*c*17], [*c*18], [*e*19]

- **4.1 Incremental Forward Synchronization.** I formalized a fully forward incremental, unidirectional synchronization technique of view models allowing chaining of views where the object of view model depend on the match set of the precondition of derivation rules. [*c*15], [*c*18], [*e*19]
- **4.2 Change Impact Analysis.** I analyzed the impact of changes in underlying source models in case of backward propagation. The impacted part is added to the logic solver as additional constraints to calculate minimally modified source model candidates. [*c*15], [*c*11]
- **4.3 Realization of Forward Synchronization.** I realized the incremental and forward view synchronization technique where elementary derivation rules are captured by graph patterns and the reactive synchronization process uses the Viatra EVM. [*c*15], [*c*18]
- **4.4 Evaluation.** I evaluated the scalability of the proposed approaches on case studies from the avionics and the health-care domain. [*c*15], [*c*18]

The query-based forward synchronization from an arbitrary model to its notional model for visual presentation is the contribution of Zoltán Ujhelyi[Ujh16], whereas my contribution generalizes his approach to view models. Introducing backward change propagation are shared contributions with Oszkár Semeráth[Sem19]. The transformation of the preconditions described by graph patterns and the impacted parts to first order logic is the contribution of Oszkár Semeráth whereas my contribution is the impact analysis for selecting changing parts.

Chapter 7

The MONDO Collaboration Framework

7.1 Introduction

The prototype implementation of the proposed approaches introduced in the previous chapters are developed within the MONDO Collaboration Framework. The MONDO Collaboration Framework[1] aims to (**C1**) extend traditional version control systems with advanced secure collaborative modeling features such as (**C2**) fine-grained model-level access control, (**C3**) property-based locking, (**C4**) automated model merge; in (**C5**) both offline and online collaboration scenarios[2].

C1. Integration with VCS: Our framework is currently integrated on the server side with Subversion (SVN) [SVN] (requested by the industrial partners of MONDO European FP7 Project [Kol+16]) as an underlying version control system, but the approach can be adapted to any other VCS supporting the appropriate hooks. In the offline scenario, users will interact with a version history in a regular SVN repository using an offline modeling tool and any SVN client. The results of online collaboration sessions will also be persisted in SVN.

C2. Fine-grained model-level access control: Traditional VCSs capture access control on the granularity of repositories or files, rigidly coupling model structure to permissions. Our framework can grant or deny access separately for each model element, cross-reference or attribute slot by offering fine-grained model access control policies (proposed in [*c*14; *c*12][PGC16]) on top of traditional file-level access control. Since large industrial models may have millions of model elements, assigning explicit permissions individually to each element would be laborious; thus we provide a *rule-based access control policy language* [*c*14; *c*12], where a single rule may grant or deny permissions for any number of assets. For concisely specifying and efficiently enforcing property-based access control rules, the affected assets are selected by a *declarative query*, which may take into account the type and attributes of model elements, as well as their wider context within the model. Our framework uses a bidirectional model transformation (a *lens* [*c*14]) to enforce access rules. Models filtered according to read permissions may violate high-level, language-specific *well-formedness constraints*, but *internal consistency* is maintained so that they can always be persisted, loaded and traversed.

C3. Property-based locking: Effective collaboration requires that engineers working simultaneously do not interfere with each other. This is traditionally achieved by partitioning the model into fragment files and providing file-level locks; while object-level locks are also available in some systems [CDO]. The former is prone to overlocking, the latter to underlocking [*c*5]. Our

[1] Source code of the framework can be found at the following link: https://github.com/FTSRG/mondo-collab-framework

[2] Screencast demonstration of the framework can be found at the following link: https://youtu.be/Ix3CgmsYIU0

framework implements a novel property-based locking technique introduced in [*c*16; *c*5] where locks protect operations by capturing their preconditions as a declarative query. The lock forbids other users from changing the model in a way that affects the result set of the query, thereby violating a precondition of the lock owner.

C4. Automated model merge: Asynchronous offline transactions lead to conflicts. Computing differences, conflicts and merged resolutions is significantly more complex over graph-based knowledge representations than over textual source code. Inter-model dependencies make conflicts surprisingly easy to introduce and hard to resolve. Our framework supports search-based automated model merge using a technique proposed in [*c*13] that computes several candidate resolutions to a conflict thus reducing the required user interaction to simply choosing the preferred solution.

C5. Offline & online collaboration: Since the actual architecture of our framework has major differences between the offline and (web-based) online collaboration scenarios, the upcoming sections of the paper aim to highlight and explain these differences.

Added Value Compared to Related Tools. Compared to traditional file-based collaborative version control systems (e.g. SVN, Git), we provide an extra layer of fine-grained server-side access control and locking that flexibly decouple model hierarchy from permissions. Compared to model repositories (e.g. CDO[CDO], EMFStore[EMFStore], MetaEdit+[Tol07]), our collaboration framework is transparent, i.e. it does not require any modifications to existing front-end (single-user) modeling tools.

Target Audience. Within the MONDO European FP7 Project [Kol+16], three kinds of users have been interacting with the MONDO Collaboration Framework. *Domain Engineers* read and write models taking advantage of advance support for access control, locking and merge. *Security Experts* act as superusers who are responsible for setting up secure repositories and defining access control policies (and lock types) that are enforced on Domain Engineers. Finally, *Language Developers* may extend our framework with knowledge specific to a given modeling language, which is required for the server to enforce fine-grained write access control, and optional for the client to improve the merge process.

7.2 Offline Collaboration Tool

In the offline scenario, models are stored in a Version Control System (VCS) as files (e.g. *XMI, binary*) and users work on local working copies of the models in long transactions called commits. The goal of our approach is to enforce fine-grained model access control rules and locks on top of existing security layers available in the VCS such as SVN or Git.

In the offline case, the MONDO Collaboration Framework uses two types of repositories called *gold* and *front* as they are introduced in Section 4.5.1.1 and depicted in Figure 4.5. The *gold* repository contains complete information about the models, but it is not accessible to collaborators. Each user has a *front* repository, containing a full version history of filtered and obfuscated copies of the gold models. A user first submits a new model version to his/her front repository; then changes introduced in this revision will be interleaved into the gold model; finally, the new gold revision will be propagated to the front repositories of other users. As a result, each collaborator continues to work with a dedicated VCS repository as before, being unaware that this front repository may contain filtered and obfuscated information only.

Key Features

Secure Access Control. Security policies[c12] consist of fine-grained write and read access control rules. *Write access control* rules limit which model elements of a model are allowed to be changed, while *read access control* rules determine which model elements of a model are visible to a user. These rules can be defined on the language and the model level.

Access control rules are enforced upon each commit at a front repository[c14]. When a new revision is attempted to be added to a front repository, its success is conditional to validating the write access control rules in the gold repository. If the commit is accepted, the read access control rules will determine which changes will be propagated to the other front repositories. This scheme enforces the access control rules even if users access their personal front repositories using standard VCS clients and off-the-shelf modeling tools, greatly reducing adoption costs in the offline scenario.

In practice, access control is enforced by *hooks* of the repositories. Upon each commit, the *pre-commit hook* of the front repository applies the *security lens* (a bidirectional model transformation parameterized by the user identity, the access control policy and the locks) to obtain the updated gold model, and rejects the commit immediately if any write rule is violated. If the commit is accepted, the gold repository is updated accordingly, where a *post-commit hook* will apply the lens to propagate changes to the rest of the front repositories.

The lens needs to identify corresponding objects between the models in front and gold repositories. By default, simple global identifiers are used to establish object identities, but *Language Developers* are able to extend this behavior to find an identifier-free solution.

Access control policies for different collaborators are defined by security experts who are superusers from a security perspective. We made the design decision that the model-level *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. Since the rule names and parameters do not normally contain sensitive information, these policies may potentially be made readable for every user, without major security concern. However, if it is required, the readablility of these *authorization files* can be denied separately on each front repository.

As future work to further improve usability with tighter feedback, offline clients may approximatively evaluate write access rules on their (incomplete) offline copies themselves prior to committing.

Automated Model Merge. When concurrent modifications are committed to the VCS, different versions of the models need to be merged. During the merge phase, changes introduced by different collaborators may contradict each other which leads to conflicts that have to be resolved. The MONDO Collaboration Framework integrates the automated model merge approach proposed in [c13] into the Eclipse teamwork UI. The tool computes and displays all possible candidate resolutions of a conflict, and the user can select the most suitable one. Choosing a solution is assisted by several metrics such as the number of elements deleted, number of changes included in the solution, etc.

By default, the model merge feature contains elementary conflict resolution strategies e.g. modify vs. delete objects that can be extended with complex domain-specific resolution rules by *Language Developers*.

Locking. Locks reduce the need for merging by preventing other users from modifying the models in a way that would cause conflicts. Similarly to security policies, fine-grained locks[c5] can be defined as rules on the levels of metamodel and instance model in accordance with [c16]. The evaluation of locks also happens at commit time at the gold repository, and its actual im-

plementation is similar to checking access control rules. As a potential future improvement, certain locks could be checked offline, just like write access rules.

Off-the-shelf Client. The *front* repository acts as a standard VCS server, so users can use their favorite off-the-shelf VCS clients and off-the-shelf (single-user) modeling tools without any mandatory collaboration-specific customization. This opt-in compatibility greatly enhances usability in case of the offline scenario.

Management UI. An optional client interface is integrated into the Eclipse IDE to allow users to discover the address of their front repository, while superusers can also execute server-side tasks, e.g. reset front repositories.

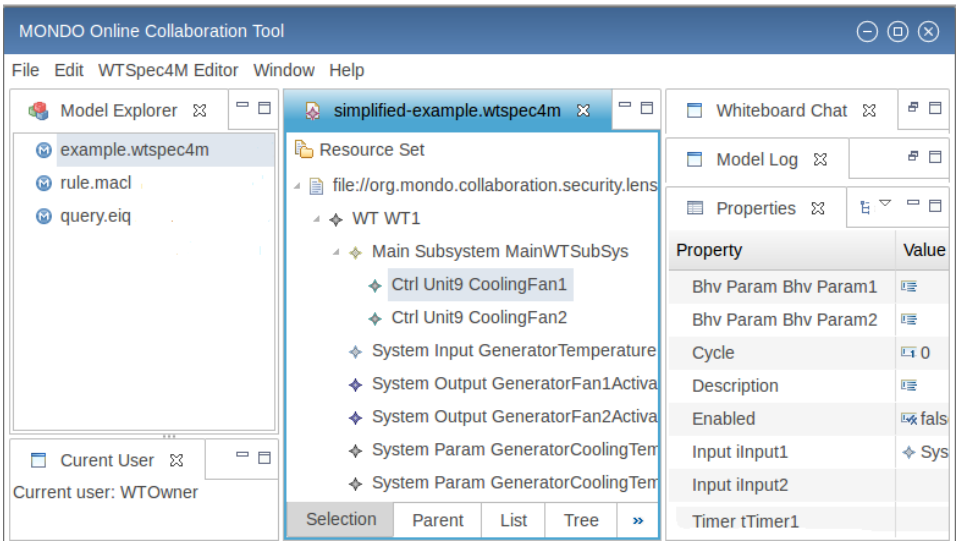


Figure 7.1: User Interface for Online Collaboration Tool

7.3 Online Collaboration Tool

In the online scenario, several users can join a *collaborative modeling session* (a *whiteboard*, see Section 4.5.2), and simultaneously display and edit the same model with short transactions where changes are propagated immediately to other users. Models are kept in server memory and users access the model directly on the server, in contrast to the offline scenario, where users manipulate local copies of the models.

Similarly to modern collaborative editing tools (such as Google Sheets [GDocs]), 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 an existing whiteboard. When all users have left, the whiteboard can be disposed. The model may be persisted periodically, or on demand ("save button") to a VCS. The *session manager* component enables collaborators to start / join or leave whiteboard sessions, persist models to disk and VCS. When a new whiteboard session is initiated, file-level locks are placed on the resources of the related model to prevent conflicts in the VCS upon persisting.

Collaborators perform model edit operations via a client application that directly connects to the server and the whiteboard in particular. In the current solution, this client is a modeling tool implemented as a web application accessible by any modern web browser (see Figure 7.1) where the

modeling tool can be extended by any known Eclipse plug-in. In the future, we plan to provide an accessible web API (and/or reuse CDO [CDO] API) to support desktop IDEs with our online features.

Key features

Secure Access Control and Locking Our framework enforces read access control rules for different collaborators also for the online scenario, thus different users do not actually see the same model contents. Instead, each of them actually connects to a dedicated view of the model that has been filtered and obfuscated, but kept in synch with (visible) modifications performed by other users.

The security lens, run as a live transformation, is responsible for incrementally maintaining consistency between front and gold models after each modification by enforcing access control rules and preventing lock violations. It is worth highlighting that access control and locking policies are editable by users with sufficient privileges and they will be reapplied upon modification within the same session.

Automated Model Merge In online collaboration, automated model merge is irrelevant: only short transactions (single edit operations) are allowed which are propagated immediately to all other views (i.e. model edit is synchronous), thus model conflicts may occur only in a very short time window.

Undo/Redo Undo/Redo is supported for all the changes carried out by a user. Each client stores only their own change history, thus no one can revert changes carried out by other users.

User Interface The prototype user interface of our tool, depicted in Figure 7.1, uses the editors automatically generated from EMF metamodels[3] which also provides similar modeling environment as the desktop Eclipse IDE. By default, it provides (1) *Tree editors* for accessing the models and (2) *Properties* view to modify the attributes of model element. We also implemented several additional views such as (3) *Current User* which shows the currently logged in user, and (4) *Model Explorer* that lists the accessible models, policy definitions (*.macl), lock definitions (*.mpbl) and related query files in the underlying VCS.

7.4 Secure Software Configuration Management

MONDO Collaboration Framework can be directly used in industrial software configuration management (SCM) scenarios.

Change requests In SCM, changes may require the approval of multiple collaborators in model/code reviews. Without sufficient permissions to commit to gold, a proposer may create a change request. The request can be inspected for approval by other collaborators, and committed once it is signed off by one or more reviewers with jointly sufficient write privileges.

Read access is enforced on change requests via GET before being shown to reviewers. However, as access control is attribute-based, this filtering can be manipulated, e.g. an attacker may propose changes in artifact ownership. Privilege escalation is averted as follows: first, a modified PUTBACK applies those proposed changes that are allowed with the joint write permissions of the proposer and the reviewer; then the merged result model is filtered by GET for the reviewer. This way, the front model on the change request will only contain modifications that the reviewer can enact, or the proposer would be allowed to commit anyway.

[3] EMF and RAP integration: http://tinyurl.com/emf-rap-integration

Collaboration workflows In the industrial context, complex collaboration workflows along multiple branches (production, feature, etc.) are needed to co-evolve multiple versions of a system design for continuous engineering. Since access control rules use a graph query language for identifying assets, the scope of restrictions can be easily extended to support multiple branches. Domain-specific types defined by the metamodel are complemented with SCM-specific concepts like workspace, resource, branch, revision or tag, but no further adaptation is needed to the access control language. Thus one can specify access control policies where a collaborator is restricted to observe model versions on specific branches or created after certain revisions.

Industrial applications Together with the industrial partners of MONDO, we carried out a detailed scalability evaluation of the collaboration server with respect to increasing number of front repositories, increasing size of models or increasing change size (for the offline case) with promising results. Further practical relevance stems from the fact that certain industrial products (like NoMagic Teamwork Cloud) have already adapted a mixed online and offline collaboration architecture, thus the concepts of secure views are directly applicable.

While we promote graph queries to drive rule-based access control management for collaborative modeling, it is worth emphasizing that graph queries can also support other server-side tasks such as change impact analysis or automated conformance checks. Moreover, since lenses for checking access control rules can be wrapped into micro-services (like Docker), we foresee the seamless adaptation and integration of secure views into existing product line management tools or future collaborative cloud-based model repositories.

Limitations Our collaboration server is currently integrated with Subversion (SVN) as requested by the industrial partners of MONDO European FP7 Project, where version history is managed in a standard SVN repository and users collaborate by using existing modeling tools and any SVN client. Integration with Git as underlying VCS technology is ongoing work.

7.5 Practical Benefits

Key benefits of our collaborative modeling framework for MBSE include the following:

Collaboration of heterogeneous stakeholders. Our framework supports collaborative modeling between engineering teams of different companies (e.g. an integrator and its subcontractors) while protecting their intellectual property with the secure storage of the gold model.

Extra layer of access control. Model-level fine-grained access control using secure views injects an extra layer of protection on top of existing protection offered by the underlying repository.

Validation of access control policies. Access control rules support the consistent assignment and maintenance of permissions for large models and enable the systematic validation of access control policies (e.g. to ensure export control regulations or investigate a security breach).

Compliance with SCM practices. Access control policies can be defined for modern SCM practices to collaborate along multiple branches, formal change request, etc.

Smooth integration with existing tools. Our framework extends existing server-side repositories while keeping client-side modeling tools intact, thus engineers may continue using existing collaborative tools.

As such, powerful existing collaboration practices used in software engineering can be complemented when collaborating over models.

Chapter 8

Conclusion

Current dissertation is focused on developing (i) a modeling language to capture high-level access control policies (ii) a general secure collaboration scheme that guarantees that high-level access control policies are respected during collaboration and it can be integrated into existing version control systems (e.g. SVN) to support offline scenario; (iii) automated merging and fine-grained locking to enhance the efficiency of conflict resolution and prevention upon concurrent modification of the models; (iv) derivation and incremental maintenance of view models to provide specific focus of the designers by abstracting from unnecessary details of the underlying system model. The presented approaches have been developed within several international research projects: MONDO, TRANS-IMA, CERTIMOT and CONCERTO. The prototype of the contributions are implemented within the MONDO Collaboration Framework.

8.1 Fine-grained Access Control Management

Approach. A parameterizable fine-grained policy language was proposed to define concise access control rules and be able to fine-tune the resolution of conflicting rules. Rules can permit, obfuscate or deny permissions of read or write for model asset (object, link or slot) selected by a graph query (see Section 3.8). Default permissions are assigned to the rest of model assets .

Deterministic application of the access control rules was defined to obtain the same effective permissions after every execution where all internal consistency rules are taken into account. During the process of conflict resolution, our approach maintains a set of permission set. The initial permission set is obtained from rules and defaults. Then direct and indirect consequences of the access control rules are propagated in permission set (e.g. in slot should be visible, its owner object should be visible as well). Our approach results in a conflict-free effective permission set where exactly one read and one write permissions is assigned to each model asset without exception.

Uniqueness. The approach allows system engineers to capture high-level policy rules instead of explicitly assigning permissions for each of them where the result of effective permission provides consistent model. Effective permission set can be incrementally reevaluated if a modification occurred in the model.

Application. The proposed concepts are applied by IncQuery Server[Heg+18] product which can provide additional incremental query evaluation services including change impact analysis, ad-hoc queries as well as *fine-grained access control management* to several model repositories (e.g. NoMagic's Teamwork Cloud).

Future Work. As future work, we plan to (i) extend the security language with preselected sets of policy options (such as the resolution strategies of XACML) and accompanying "design patterns" on how policies should be constructed (ii) investigate incrementality of conflict resolution algorithms with respect to policy changes.

8.2 General Secure Collaboration Scheme

Approach. Bidirectional model transformations was defined to (i) derive filtered views (*front models*) for each collaborator from the original model (*gold model*) containing all the information and to (ii) propagate changes introduced into these views back to a server in both *online* and *offline* scenarios (see Section 4.7). Access control policies consist of rules that allow, obfuscate or deny read and/or write permissions of model parts identified by graph patterns.

A collaboration scheme between the clients of multiple collaborators and exactly one server was described to support fine-grained access control in offline scenario. The server stores the gold models and the clients can download their specific front models. Modifications, executed by a clients, are submitted to the server and they are accepted if write permissions are successfully checked. Right after the submission, the changes are propagated to the other front model while read permissions are enforced. Finally, clients can downloaded their updated front models.

The scheme was realized by extending *SVN*[SVN] using its hooks. The server and clients are realized as a *gold repository* and multiple *front repositories*, respecively. The *gold repository* contains *gold models*, but it is not accessible to collaborators. Each collaborator is assigned to a specific *front repository* containing a full version history of the front models. Change propagations are maintained between the repositories. As a result, each collaborator continues to work with a dedicated VCS as before, thus they are unaware that this front repository may contain filtered and obfuscated data.

Uniqueness. The provenly correct collaboration scheme is able to enforce fine-grained access control policies of modeling artifacts over existing version control system in case of offline scenarios. The scheme and its realization is demonstrated in MONDO Collaboration Framework as an integration with SVN[SVN].

Application. A usability evaluation of the prototype implementation of the proposed collaboration schema has been carried out by industrial partners *IK4-Ikerlan* and *UNINOVA* within the evaluation phase of MONDO project in the context of (1) a wind turbine case study (with small models but many users) and (2) a building information model (with models over 100,000 elements but fewer users). The number of concurrent users working on different views of the same model and the time for propagating changes and notifications among the concurrent users were evaluated both in offline and online cases, and the engineers could successfully collaborate in both cases. An experience report on wind turbines control applications development is also presented in [Góm+17] by IK4-Ikerlan.

Future Work. As future work, we would like to (i) address the limitations presented in Section 4.6.2, (ii) investigate the possibilities of building correspondence relations between the original model and filtered copy of it dedicated to a certain collaborator, and (iii) realize our collaboration scheme with other frameworks (e.g. Git, GenMyModel) and with support for continuous integration and review / change request management systems.

8.3 Conflict Reduction and Handling

Approach. Property-based locking was introduced to provide fine-grained conflict reduction while DSE-Merge was proposed to handle conflicts implied by concurrent editing (see Section 5.6).

The concept of property-based locking was described where collaborators request locks specified as a property of the model which need to be maintained as long as the lock is active. Hence, other collaborators are permitted to carry out any modifications that do not violate the defined property of the lock. The realization of property-based locking strategy is proposed as a common generalization of existing fragment-based and object-based locking approaches. Complex properties are described as graph patterns to express structural (and attribute) constraints for a model where the result set, i.e. the matches of graph pattern, can be calculated by pattern matchers or query engines. Only those modifications are allowed that do not change the result set of a list of queries.

DSE-Merge was proposed that exploits guided rule-based *design space exploration (DSE)* to automate the three-way model merge. Three-way model merge is applied to DSE problem where the *initial model* consists of the original model O and two difference models (ΔL and ΔR); the *goal* is that there are no executable changes left in ΔL and ΔR; *operations* are defined by change driven transformation rules to process generic composite (domain-specific) operators; and *constraints* may identify inconsistencies and conflicts to eliminate certain trajectories. The output is a *set of solutions* consisting of (i) the well-formed merged model M; (ii) the set of non-executed changes $\Delta L'$,$\Delta R'$; and (iii) the collection of the deleted objects stored in $Cemetery$.

Uniqueness. The property-based approach its general and can be used for both implicit locking of subtrees and set of elements or explicit locking of a certain element and its incoming and outgoing references. In addition it extends these lock types with the definition of properties to provide less restrictive locking for the collaborators.

The closest to our merge approach are [DRE14] and [Man+15], but we rely on state-based comparison, apply a guided local-search strategy (vs. [Man+15]), detect conflicts at runtime and allow complex generic merge operations (vs. [DRE14]). Internally, we uniquely use incremental and change-driven transformations to derive the merged models. Finally, we reported scalability of merge process for models which are at least one order of magnitude larger compared to [DRE14] and [Man+15].

Application. The efficiency of the DSE Merge technique has been systematically evaluated from the user point of view using a experimental software engineering approach. The empirical tests included the involvement of the intended end users (i.e. engineers), namely undergraduate students, which were expected to confirm the impact of design decisions. In particular, we asked users to merge the different versions of the same model using DSE Merge when compared to using Diff Merge. The experiment showed that to use DSE Merge participant required lower cognitive effort, and expressed their preference and satisfaction with it.

Future Work. As future work, we plan to improve our model merge technique by further search strategies to better exploit the dependencies between rules and constraints and compare it with other search-based merge techniques [Man+15]. In case of property-based locking, we plan to extend our evaluation with respect to *underlocking* and investigate the use of incremental pattern matchers to support on-line collaboration where the collaborators work with short transactions of modifications and the response time needs to be immediate.

8.4 Synchronization of View Models

Approach. An approach was introduced where view models are conceptually equivalent to regular models and they were defined using a fully declarative, rule based formalism. *Preconditions* of rules are defined by graph patterns, which identify parts of interest in the source model. *Derivation rules* then use the match set of a graph pattern to define elements of the view model. Informally, when a new match of a query appears then the corresponding derivation rule is fired to create elements of the view model. When an existing match of a query disappears, the inverse of the derivation rule is fired to delete the corresponding view model elements (see Section 6.8).

View models derived by a unidirectional transformation are read-only representations, and they cannot be changed directly. To tackle this problem, we proposed an approach to automatically calculate possible source model candidates for a set of changes in different view models. First, the possibly impacted partition of the source model is need to be identified by observing traceability links to restrict the impact of a view modification. Then the modified view models and the query-based view specification are transformed into logic formulae. Finally, multiple valid resolutions of the source model are generated using logic solvers corresponding to the changes of view models and the constraints of the source model from the users can manually select a proper solution.

Uniqueness. Definition of a view model is *unidirectional*, while the forward propagation of the *operation-based* changes are *live*, *incremental* and executed *automatically* that also maintains *explicit traces*. At backward propagation, using partitioning as an additional input of the logic solver improves scalability issues and limits the impact of changes to a well-defined part of the source model.

Application. The proposed forward incremental view synchronization and chaining of such view models are applied by Embraer[Deb+14a] to provide *functional architecture model* and its graphical representation as view models derived from low-level Simulink models. Backward change propagation is used by the CONCERTO project for the synchronization of certain views and the underlying systems in the TeleCare domain.

Future Work. As future work, we plan to (i) prioritize the synthesized solutions, (ii) improve the calculation of the source model by calling multiple solvers to minimize the size of the solution or (iii) to use SAT/SMT solvers as replacement of the Alloy Analyzer used currently in our approach.

Publications

Number of publications:	20
Number of peer-reviewed journal papers (written in English):	2
Number of articles in journals indexed by WoS or Scopus:	2
Number of publications (in English) with at least 50% contribution of the author:	3
Number of peer-reviewed publications:	20
Number of independent citations:	68

Publications Linked to the Theses

	Journal papers	International conference and workshop papers
Thesis 1	[*j*1],[*j*2]	[*c*6],[*c*7],[*c*3]
Thesis 2	[*j*1],[*j*2]	[*c*14],[*c*12],[*c*6],[*c*9],[*c*7],[*c*10]
Thesis 3	[*j*1]	[*c*5],[*c*13],[*c*16],[*c*17],[*c*6],[*c*9],[*c*7],[*c*4]
Thesis 4	—	[*c*15],[*c*11],[*e*19],[*c*18],[*c*7]

Journal Papers

[*j*1] Csaba Debreceni, Gábor Bergmann, István Ráth, and Dániel Varró. Secure views for collaborative modeling. *IEEE Software*, 2018. DOI: 10.1109/MS.2018.290101728.

[*j*2] Csaba Debreceni, Gábor Bergmann, István Ráth, and Dániel Varró. Enforcing fine-grained access control for secure collaborative modeling using bidirectional transformations. *Software and System Modeling, MODELS 2016 Special Section*, 2017. DOI: 10.1007/s10270-017-0631-8.

International Conference and Workshop Papers

[*c*3] Ábel Hegedüs, Gábor Bergmann, Csaba Debreceni, Ákos Horváth, Péter Lunk, Ákos Menyhért, István Papp, Dániel Varró, Tomas Vileiniskis, and István Ráth. IncQuery Server for Teamwork Cloud: scalable query evaluation over collaborative model repositories. In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pp. 27–31. 2018. DOI: 10.1145/3270112.3270125.

[c4] Ankica Barisic, Csaba Debreceni, Dániel Varró, Vasco Amaral, and Miguel Goulão. Evaluating the efficiency of using a search-based automated model merge technique. In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2018, Lisbon, Portugal, October 1-4, 2018*, pp. 193–197. 2018. DOI: 10.1109/VLHCC.2018.8506512.

[c5] Csaba Debreceni, Gábor Bergmann, István Ráth, and Dániel Varró. Property-based locking in collaborative modeling. In: *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*, pp. 199–209. 2017. URL: https://doi.org/10.1109/MODELS.2017.33.

[c6] Csaba Debreceni, Gábor Bergmann, Márton Búr, István Ráth, and Dániel Varró. The MONDO collaboration framework: secure collaborative modeling over existing version control systems. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pp. 984–988. 2017. DOI: 10.1145/3106237.3122829.

[c7] Csaba Debreceni. Advanced techniques and tools for secure collaborative modeling. In: *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017.* Pp. 549–554. 2017. URL: http://ceur-ws.org/Vol-2019/acm_src_1.pdf.

[c8] Gábor Bergmann, Csaba Debreceni, István Ráth, and Dániel Varró. Towards efficient evaluation of rule-based permissions for fine-grained access control in collaborative modeling. In: *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017.* Pp. 135–144. 2017. URL: http://ceur-ws.org/Vol-2019/commitmde_2.pdf.

[c9] Abel Gómez, Xabier Mendialdua, Gábor Bergmann, Jordi Cabot, Csaba Debreceni, Antonio Garmendia, Dimitrios S. Kolovos, Juan de Lara, and Salvador Trujillo. On the opportunities of scalable modeling technologies: an experience report on wind turbines control applications development. In: *Modelling Foundations and Applications - 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings*, pp. 300–315. 2017. DOI: 10.1007/978-3-319-61482-3_18.

[c10] Csaba Debreceni. Approaches to identify object correspondences between source models and their view models. In: *Proceedings of the 24th PHD Mini-Symposium (MINISY@DMIS), Budapest, Hungary, January 30-31, 2017*, pp. 14–17. 2017.

[c11] Oszkár Semeráth, Csaba Debreceni, Ákos Horváth, and Dániel Varró. Change propagation of view models by logic synthesis using SAT solvers. In: *Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016.* Pp. 40–44. 2016. URL: http://ceur-ws.org/Vol-1571/paper_6.pdf.

[c12] Csaba Debreceni, Gábor Bergmann, István Ráth, and Dániel Varró. Deriving effective permissions for modeling artifacts from fine-grained access control rules. In: *Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016) co-located with*

ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), St. Malo, France, October 4, 2016. Pp. 17–26. 2016. URL: http://ceur-ws.org/Vol-1717/paper6.pdf.

[*c*13] Csaba Debreceni, István Ráth, Dániel Varró, Xabier De Carlos, Xabier Mendialdua, and Salvador Trujillo. Automated model merge by design space exploration. In: *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pp. 104–121. 2016. DOI: 10.1007/978-3-662-49665-7_7.

[*c*14] Gábor Bergmann, Csaba Debreceni, István Ráth, and Dániel Varró. Query-based access control for secure collaborative modeling using bidirectional transformations. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, pp. 351–361. 2016. ACM Distinguished Paper Award. URL: http://dl.acm.org/citation.cfm?id=2976793.

[*c*15] Oszkár Semeráth, Csaba Debreceni, Ákos Horváth, and Dániel Varró. Incremental backward change propagation of view models by logic solvers. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, pp. 306–316. 2016. URL: http://dl.acm.org/citation.cfm?id=2976788.

[*c*16] Marsha Chechik, Fabiano Dalpiaz, Csaba Debreceni, Jennifer Horkoff, István Ráth, Rick Salay, and Dániel Varró. Property-based methods for collaborative model development. In: *Joint Proceedings of the 3rd International Workshop on the Globalization Of Modeling Languages and the 9th International Workshop on Multi-Paradigm Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, GEMOC+MPM@MoDELS 2015, Ottawa, Canada, September 28, 2015.* Pp. 1–7. 2015. URL: http://ceur-ws.org/Vol-1511/paper-01.pdf.

[*c*17] Hani Abdeen, Dániel Varró, Houari A. Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. Multi-objective optimization in rule-based design space exploration. In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pp. 289–300. 2014. DOI: 10.1145/2642937.2643005.

[*c*18] Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró. Query-driven incremental synchronization of view models. In: *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, VAO@STAF 2014, York, United Kingdom, July 22, 2014*, pp. 31–38. 2014. DOI: 10.1145/2631675.2631677.

Local Conference and Workshop Papers

[*e*19] Csaba Debreceni. Automated abstraction in model-driven engineering. In: *Mesterpróba 2014 Tudományos konferencia végzös MSc és elsöéves PhD hallgatóknak Távközlés és infokommunikáció témakörében, Budapest, Hungary, May 29, 2014*, pp. 67–70. 2014.

Additional Publications (Not Linked to Theses)

International Conference and Workshop Papers

[*c*20] Gábor Szárnyas, Oszkár Semeráth, Benedek Izsó, Csaba Debreceni, Ábel Hegedüs, Zoltán Ujhelyi, and Gábor Bergmann. Movie Database Case: An EMF-IncQuery Solution. In: *Proceedings of the 7th Transformation Tool Contest part of the Software Technologies: Applications and Foundations (STAF 2014) federation of conferences, York, United Kingdom, July 25, 2014.* Pp. 103–115. 2014. URL: http://ceur-ws.org/Vol-1305/paper14.pdf.

Bibliography

[4store] Garlik. 4store. http://4store.org/trac/wiki/GraphAccessControl.

[Abe+07] Fabian Abel et al. Enabling adanced and context-dependent access control in RDF stores. In: *The Semantic Web, 6th Int. Semantic Web Conf., 2nd Asian Semantic Web Conf.* Pp. 1–14. 2007.

[Alt+08] Kerstin Altmanninger, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Martina Seidl, Wieland Schwinger, and Manuel Wimmer. Amor–towards adaptable model versioning. In: *1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS*, vol. 8, pp. 4–50. 2008.

[Anj+14] Anthony Anjorin, Sebastian Rose, Frederik Deckwerth, and Andy Schürr. Efficient model synchronization with view triple graph grammars. In: *Modelling Foundations and Applications*, pp. 1–17. Springer, 2014.

[ASW09] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems* 5(3), 2009, pp. 271–304.

[Bag+14] Alessandra Bagnato, Etienne Brosse, Andrey Sadovykh, Pedro Maló, Salvador Trujillo, Xabier Mendialdua, and Xabier De Carlos. Flexible and scalable modelling in the mondo project: industrial case studies. In: *XM@ MoDELS*, pp. 42–51. 2014.

[Bar] Mikael Barbero. EMF Compare 2.0: scaling to millions. In: *EclipseCON '13, Boston*,

[Bas+14] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. MDEForge: an extensible web-based modeling platform. In: *CloudMDE@MoDELS*, 2014.

[Ben+14] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4EMF, A scalable persistence layer for EMF models. In: *Modelling Foundations and Applications - 10th European Conf., ECMFA 2014*, pp. 230–241. 2014.

[Ber+11] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for emf models. In: *International Conference on Theory and Practice of Model Transformations*, pp. 167–182. 2011.

[Ber+12] Gábor Bergmann, István Ráth, Gergely Varró, and Dániel Varró. Change-driven model transformations. change (in) the rule to rule the change. *Software and Systems Modeling* 11, 2012, pp. 431–461. DOI: 10.1007/s10270-011-0197-9.

[Ber+15a] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. VIATRA 3: a reactive model transformation platform. In: *International Conference on Theory and Practice of Model Transformations*, pp. 101–110. 2015.

[Ber+15b] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A reactive model transformation platform. In: *Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings*, pp. 101–110. 2015.

[BFK98] Matt Blaze, Joan Feigenbaum, and Angelos D Keromytis. Keynote: trust management for public-key infrastructures. In: *International Workshop on Security Protocols*, pp. 59–63. 1998.

[BPA07] Ruth Breu, Gerhard Popp, and Muhammad Alam. Model based development of access policies. *International Journal on Software Tools for Technology Transfer* 9(5), 2007, pp. 457–470.

[Bro+09] Petra Brosch, Martina Seidl, Konrad Wieland, and Manuel Wimmer. We can work it out: collaborative conflict resolution in model versioning. In: *Proceedings of the Eleventh European Conference on Computer Supported Cooperative Work, ECSCW 2009, 7-11 September 2009, Vienna, Austria*, pp. 207–214. 2009. URL: http://www.ecscw.org/2009/15-BroschEtAl.pdf.

[Bro+11] Petra Brosch, Uwe Egly, Sebastian Gabmeyer, Gerti Kappel, Martina Seidl, Hans Tompits, Magdalena Widl, and Manuel Wimmer. Towards semantics-aware merge support in optimistic model versioning. In: *Models in Software Engineering - Workshops and Symposia at MODELS 2011, Wellington, New Zealand, October 16-21, 2011, Reports and Revised Selected Papers*, pp. 246–256. 2011.

[Bro+12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An introduction to model versioning. In: *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, pp. 336–398. 2012.

[Bru+15] Hugo Brunelière, Jokin Garcıa Perez, Manuel Wimmer, and Jordi Cabot. EMF views: A view mechanism for integrating heterogeneous models. In: *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings*, pp. 317–325. 2015. DOI: 10.1007/978-3-319-25264-3_23.

[Cam17] Cambridge Dictionary. Obfuscate, 2017. http://dictionary.cambridge.org/dictionary/english/obfuscate.

[CDO] The Eclipse Foundation. CDO. http://eclipse.org/cdo.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal* 45(3), 2006, pp. 621–645.

[Cha+13] Jian Chang, Peter Gebhard, Andreas Haeberlen, Zachary G. Ives, Insup Lee, Oleg Sokolsky, and Krishna K. Venkatasubramanian. Trustforge: flexible access control for collaborative crowd-sourced environment. In: *Eleventh Annual International Conference on Privacy, Security and Trust, PST 2013, 10-12 July, 2013, Tarragona, Catalonia, Spain, July 10-12, 2013*, pp. 291–300. 2013. DOI: 10.1109/PST.2013.6596065.

[Cha14] Sitaram Chamarty. *Gitolite Essentials*. Packt Publishing Ltd, 2014.

[CHP06] Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, eds. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. ACM, 2006.

[Cic+10] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL: a bidirectional and change propagating transformation language. In: *Software Language Engineering*, pp. 183–202. Springer, 2010.

[CJC11] Cauê Clasen, Frédéric Jouault, and Jordi Cabot. VirtualEMF: a model virtualization tool. In: *Advances in Conceptual Modeling. Recent Developments and New Directions*, pp. 332–335. Springer, 2011.

[CK13] Glenn Callow and Roy Kalawsky. A satisficing bi-directional model transformation engine using mixed integer linear programming. *Journal of Object Technology* 12(1), 2013, 1: 1–43. DOI: 10.5381/jot.2013.12.1.a1.

[Con08] Nancy Conner. *Google Apps: The Missing Manual: The Missing Manual*. " O'Reilly Media, Inc.", 2008.

[Conc] CONCERTO ARTEMIS project. http://concerto-project.org/.

[CS14] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.

[CZ88] Arturo I. Concepcion and Bernard P. Zeigler. DEVS formalism: A framework for hierarchical model development. *IEEE Trans. Software Eng.* 14(2), 1988, pp. 228–241. DOI: 10.1109/32.4640.

[Deb+02] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation* 6(2), 2002, pp. 182–197.

[Deb+14a] Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró. Query-driven incremental synchronization of view models. In: *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, p. 31. 2014.

[Dis08] Zinovy Diskin. Algebraic models for bidirectional model synchronization. In: *MoDELS*, pp. 21–36. 2008.

[Dou96] Paul Dourish. Consistency guarantees: exploiting application semantics for consistency management in a collaboration toolkit. In: *CSCW '96, Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work, Boston, MA, USA, November 16-20, 1996*, pp. 268–277. 1996. DOI: 10.1145/240080.240300.

[DRE14] Hoa Khanh Dam, Alexander Reder, and Alexander Egyed. Inconsistency resolution in merging versions of architectural models. In: *2014 IEEE/IFIP Conference on Software Architecture, WICSA 2014, Sydney, Australia, April 7-11, 2014*, pp. 153–162. 2014. DOI: 10.1109/WICSA.2014.31.

[ECM11] Javier Espinazo-Pagán, Jesús Sánchez Cuadrado, and Jesús Garcıa Molina. Morsa: A scalable approach for persisting and accessing large models. In: *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, pp. 77–92. 2011. DOI: 10.1007/978-3-642-24485-8_7.

[EMF] The Eclipse Project. *Eclipse Modeling Framework*. http://www.eclipse.org/emf.

[EMF-Comp] The Eclipse Foundation. EMF Compare. http://eclipse.org/emf/compare/.

[EMF-Diff] The Eclipse Foundation. EMF Diff/Merge. http://eclipse.org/diffmerge/.

[EMFStore] The Eclipse Foundation. EMFStore. http://eclipse.org/emfstore.

[Far+10] Matthias Farwick, Berthold Agreiter, Jules White, Simon Forster, Norbert Lanzanasto, and Ruth Breu. A web-based collaborative metamodeling environment with secure remote model access. In: *Web Engineering, 10th International Conference, ICWE 2010, Vienna, Austria, July 5-9, 2010. Proceedings*, LNCS, vol. 6189, pp. 278–291. Springer, 2010.

[Fea89] Martin S Feather. Detecting interference when merging specification evolutions. In: *ACM SIGSOFT Software Engineering Notes*, vol. 14, pp. 169–176. 1989.

[FM04] Irini Fundulaki and Maarten Marx. Specifying access control policies for XML documents with XPath. In: *9th ACM Symposium on Access Control Models and Technologies*, pp. 61–69. 2004.

[Fos+07] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems* 29(3), 2007, p. 17.

[FPZ09] J. Nathan Foster, Benjamin C. Pierce, and Steve Zdancewic. Updatable security views. In: *Proceedings of the 2009 22Nd IEEE Computer Security Foundations Symposium*, CSF '09, pp. 60–74. IEEE Computer Society, 2009. DOI: 10.1109/CSF.2009.25.

[FSC12] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: towards modeling and reasoning with uncertainty. In: *Proceedings of the 34th International Conference on Software Engineering*, pp. 573–583. IEEE Press, 2012.

[Gal+11] Jesús Gallardo, Ana I. Molina, Crescencio Bravo, Miguel A. Redondo, and César A. Collazos. An ontological conceptualization approach for awareness in domain-independent collaborative modeling systems: application to a model-driven development method. *Expert Syst. Appl.* 38(2), 2011, pp. 1099–1118. DOI: 10.1016/j.eswa.2010.05.005.

[GBR12] Jesús Gallardo, Crescencio Bravo, and Miguel A. Redondo. A model-driven development method for collaborative modeling tools. *J. Network and Computer Applications* 35(3), 2012, pp. 1086–1105. DOI: 10.1016/j.jnca.2011.12.009.

[GDocs] Google. Google Sheets. docs.google.com.

[Ge03] S. Godik and T. Moses (eds). EXtensible access control markup language (XACML) version 1.0. In: 2003.

[Gen] Axellience. GenMyModel. http://genmymodel.com.

[Gho+15] Hamid Gholizadeh, Zinovy Diskin, Sahar Kokaly, and Tom Maibaum. Analysis of source-to-target model transformations in quest. In: *Proceedings of the 4th Workshop on the Analysis of Model Transformations co-located with (MODELS 2015, Ottawa, Canada*, pp. 46–55. 2015. URL: http://ceur-ws.org/Vol-1500/paper6.pdf.

[Gib+14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In: Erika Ábrahám and Klaus Havelund (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 8413, pp. 187–201. 2014.

[GK15] Loıc Gammaitoni and Pierre Kelsen. F-alloy: an alloy based model transformation language. In: *Theory and Practice of Model Transformations*, pp. 166–180. Springer, 2015.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally (extended abstract). In: *Proc. of the Int. Conf. on Management of Data, ACM*, pp. 157–166. 1993.

[Gre+13] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Found. Trends databases* 5(2), 2013, pp. 105–195. DOI: 10.1561/1900000017.

[GW09] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling (SoSyM)* 8(1), 2009.

[Heg14] Ábel Hegedüs. Back-annotation of Execution Sequences by Advanced Search and Traceability Techniques. PhD thesis. Budapest University of Technology and Economics, 2014.

[Her+15] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software & Systems Modeling* 14(1), 2015, pp. 241–269.

[Het10] Thomas Hettel. Model round-trip engineering. PhD thesis. Queensland University of Technology, 2010.

[HHV15a] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. A model-driven framework for guided design space exploration. *Autom. Softw. Eng.* 22(3), 2015, pp. 399–436. DOI: 10.1007/s10515-014-0163-1.

[HHV15b] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. A model-driven framework for guided design space exploration. *Autom. Softw. Eng.* 22(3), 2015, pp. 399–436.

[Hid+15] Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. Feature-based classification of bidirectional transformation approaches. *Software & Systems Modeling*, 2015, pp. 1–22. DOI: 10.1007/s10270-014-0450-0.

[HLR06] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In: *Model Driven Engineering Languages and Systems*, pp. 321–335. Springer, 2006.

[IFC] *ISO 16739:2013: Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries*. Int. Organization for Standardization. 2013.

[Jac] Daniel Jackson. Alloy Analyzer. http://alloy.mit.edu/.

[Jae90] Rex Jaeschke. Encrypting c source for distribution. *Journal of C Language Translation* 2(1), 1990, pp. 71–80.

[JF06] Amit Jain and Csilla Farkas. Secure resource description framework: an access control model. In: *11th ACM Symposium on Access Control Models and Technologies*, pp. 121–129. 2006.

[JFace] The Eclipse Foundation. JFace. https://wiki.eclipse.org/JFace.

[Jür08] Jan Jürjens. Model-based run-time checking of security permissions using guarded objects. In: Martin Leucker (ed.), *Proc. of the 8th International Workshop on Runtime Verification*, LNCS, vol. 5289, pp. 36–50. Springer, 2008.

[Kol+16] Dimitrios S. Kolovos, Antonio Garcıa-Domınguez, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan de Lara, István Ráth, Dániel Varró, Gerson Sunyé, and Massimo Tisi. MONDO: scalable modelling and model management on the cloud. In: *Joint Proceedings of the Doctoral Symposium and Projects Showcase at STAF 2016*, pp. 55–64. 2016.

[Kra+06] Gerhard Kramler, Gerti Kappel, Thomas Reiter, Elisabeth Kapsammer, Werner Retschitzegger, and Wieland Schwinger. Towards a semantic infrastructure supporting model-based tool integration. In: *Proceedings of the 2006 international workshop on Global integrated model management*, pp. 43–46. 2006.

[Lau+12] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Efficient model synchronization with precedence triple graph grammars. In: *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, 2012*, LNCS, pp. 401–415. 2012.

[Luc+14] Levi Lucio, Qin Zhang, Phu Hong Nguyen, Moussa Amrani, Jacques Klein, Hans Vangheluwe, and Yves Le Traon. Advances in model-driven security. *Advances in Computers* 93, 2014, pp. 103–152. URL: http://dx.doi.org/10.1016/B978-0-12-800162-2.00003-8.

[LW13] Philip Langer and Manuel Wimmer. A benchmark for conflict detection components of model versioning systems. In: vol. 33, 2013.

[Man+15] Usman Mansoor, Marouane Kessentini, Philip Langer, Manuel Wimmer, Slim Bechikh, and Kalyanmoy Deb. MOMM: multi-objective model merging. *Journal of Systems and Software* 103, 2015, pp. 423–439. DOI: 10.1016/j.jss.2014.11.043.

[Mar+14] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, and Ákos Lédeczi. Next generation (meta)modeling: web- and cloud-based collaborative tool infrastructure. In: *Proceedings of the 8th Workshop on Multi-Paradigm Modeling co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, MPM@MODELS 2014, Valencia, Spain, September 30, 2014.* Pp. 41–60. 2014. URL: http://ceur-ws.org/Vol-1237/paper5.pdf.

[MC13] Nuno Macedo and Alcino Cunha. Implementing QVT-R bidirectional model transformations using Alloy. In: *Fundamental Approaches to Software Engineering*, pp. 297–311. Springer, 2013.

[Men02] Tom Mens. A state-of-the-art survey on software merging. *IEEE Trans. Software Eng.* 28(5), 2002, pp. 449–462.

[MHT04] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In: *Mathematics of Program Construction*, pp. 289–313. 2004.

[Mog] Mogentes EU project. URL: http://www.mogentes.eu/.

[MS-SQL] MS-SQL. URL: https://www.microsoft.com/en-us/sql-server/sql-server-2016.

[MSV18] Kristóf Marussy, Oszkár Semeráth, and Dániel Varró. Incremental view model synchronization using partial models. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pp. 323–333. 2018.

[MTF] Eclipse. Modeling Team Framework proposal. http://www.eclipse.org/proposals/mtf/. 2011.

[Neo4J] Neo4J. URL: https://www.neo4j.com.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The fujaba environment. In: *Proceedings of the 22nd international conference on Software engineering*, pp. 742–745. 2000.

[Obeo] Obeo. Obeo Designer. https://www.obeodesigner.com.

[OCL] OMG Object Constraint Language. http://www.omg.org/spec/OCL/. 2014.

[Oracle] Oracle. URL: https://www.oracle.com/.

[Orient] OrientDB. URL: https://www.orientdb.com.

[PGC16] Salvador Martınez Perez, Jokin Garcıa, and Jordi Cabot. Runtime support for rule-based access-control evaluation through model-transformation. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pp. 57–69. 2016. URL: http://dl.acm.org/citation.cfm?id=2997375.

[Pin03] Niels Pinkwart. A Plug-In Architecture for Graph Based Collaborative Modeling Systems. In: *Supplementary Proceedings of the 11th Conference on Artificial Intelligence in Education, Sydney (Australia)*, pp. 89–94. SIT, 2003.

[PS16] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. In: 2016.

[QVT] OMG. MOF 2.0 Query/View/Transformation specification (QVT), version 1.1. http://www.omg.org/spec/QVT/1.2/.

[Rát+08] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live model transformations driven by incremental pattern matching. In: *Proc. First International Conference on the Theory and Practice of Model Transformations (ICMT 2008)*, LNCS, vol. 5063, pp. 107–121. Springer Berlin-Heidelberg, 2008. DOI: 10.1007/978-3-540-69927-9_8.

[RC13] Julia Rubin and Marsha Chechik. N-way model merging. In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pp. 301–311. 2013. DOI: 10.1145/2491411.2491446.

[Rek93] Jun Rekimoto. CSCW platform system teidan and its concurrency control algorithm. *Advances in Software Science and Technology* 5, 1993, pp. 239–255.

[RHV12] István Ráth, Ábel Hegedüs, and Dániel Varró. Derived features for EMF by integrating advanced model queries. In: *Modelling Foundations and Applications*, LNCS 7349, pp. 102–117. Springer, 2012.

[Roc+15] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Collaborative repositories in model-driven engineering [software technology]. *IEEE Software* 32(3), 2015, pp. 28–34. DOI: 10.1109/MS.2015.61.

[Ros10] Andrew William Roscoe. *Understanding concurrent systems*. Springer Science & Business Media, 2010.

[Ros98] Bill Roscoe. The theory and practice of concurrency, 1998.

[Sal+15] Rick Salay, Marsha Chechik, Michalis Famelis, and Jan Gorzny. A methodology for verifying refinements of partial models. *Journal of Object Technology* 14(3), 2015, 3:1–31.

[SC15] Rick Salay and Marsha Chechik. A generalized formal framework for partial modeling. In: *Fundamental Approaches to Software Engineering*, LNCS, vol. 9033, pp. 133–148. Springer, 2015.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In: *Graph-Theoretic Concepts in Computer Science*, pp. 151–163. 1994.

[Sem+15] Oszkár Semeráth, Ágnes Barta, Ákos Horváth, Zoltán Szatmári, and Dániel Varró. Formal validation of domain-specific languages with derived features and well-formedness constraints. English. *Software and Systems Modeling*, 2015, pp. 1–36.

[Sem19] Oszkár Semeráth. Formal Validation and Model Generation for Domain-Specific Languages by Logic Solvers. PhD thesis. Budapest University of Technology and Economics, 2019.

[SFC12] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In: *Fundamental Approaches to Software Engineering*, LNCS, vol. 7212, pp. 224–239. Springer, 2012.

[Stardog] Stardog. URL: https://www.stardog.com.

[Ste+96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: managing the evolution of reusable assets. In: *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, October 6-10, 1996.* Pp. 268–285. 1996.

[Ste08] Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software & Systems Modeling* 9(1), 2008, pp. 7–20. DOI: 10.1007/s10270-008-0109-9.

[SUW13] Felix Schwägerl, Sabrina Uhrig, and Bernhard Westfechtel. Model-based tool support for consistent three-way merging of EMF models. In: *Proc. of the workshop on ACadeMics Tooling with Eclipse*, p. 2. 2013.

[SVN] Apache. Subversion. https://subversion.apache.org/.

[SVV16] Oszkár Semeráth, András Vörös, and Dániel Varró. Iterative and incremental model generation by logic solvers. *Fundamental Approaches to Software Engineering, 19th International Conference, FASE 2016*, 2016.

[Syr+13] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In: *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 29 - October 4, 2013.* Pp. 21–25. 2013. URL: http://ceur-ws.org/Vol-1115/demo4.pdf.

[Szá+15] Gábor Szárnyas, Oszkár Semeráth, István Ráth, and Dániel Varró. The TTC 2015 Train Benchmark Case for Incremental Model Validation. *Transformation Tool Contest*, 2015, pp. 129–141.

[Tol07] Juha-Pekka Tolvanen. MetaEdit+: domain-specific modeling and product generation environment. In: *Software Product Lines, 11th Int. Conf. SPLC 2007, Kyoto, Japan*, pp. 145–146. 2007.

[Tol16] Juha-Pekka Tolvanen. Metaedit+ for collaborative language engineering and language use (tool demo). In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pp. 41–45. 2016. URL: http://dl.acm.org/citation.cfm?id=2997379.

[TR05] Gabriele Taentzer and Arend Rensink. Ensuring structural constraints in graph-based models with type inheritance. In: *International Conference on Fundamental Approaches to Software Engineering*, pp. 64–79. 2005.

[Ujh+15] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* 98, 2015, pp. 80–99.

[Ujh16] Zoltán Ujhelyi. Program Analysis Techniques for Model Queries and Transformations. PhD thesis. Budapest University of Technology and Economics, 2016.

[Var+16] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and Systems Modeling* 15(3), 2016, pp. 609–629. DOI: 10.1007/s10270-016-0530-4.

[VVi] VIATRA. VIATRA Viewers Documentation. https : / / www . eclipse . org / viatra / documentation/addons.html.

[Wes14] Bernhard Westfechtel. Merging of EMF models - formal foundations. *Software and System Modeling* 13(2), 2014, pp. 757–788.

[WHR14] Jon Whittle, John Edward Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software* 31(3), 2014, pp. 79–85. DOI: 10.1109/MS.2013.65.

[Wie+13] Konrad Wieland, Philip Langer, Martina Seidl, Manuel Wimmer, and Gerti Kappel. Turning conflicts into collaboration. *Computer Supported Cooperative Work* 22(2-3), 2013, pp. 181–240.

[Xio+07] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In: *Proceedings of the 22nd IEEE/ACM Int. Conf. on Automated Software Engineering*, pp. 164–173. 2007.

[YFiles] yWorks. yFiles. https://www.yworks.com/products/yfiles.

[Zest] The Eclipse Foundation. Zest. https://www.eclipse.org/gef/zest/.

Appendix A

Collaboration Scheme Formalized as Communicating Sequential Processes

Algorithm 6 Collaboration Scheme Formalized as Communicating Sequential Processes

Range = 1..N
channel commit, update, checkout, needToUpdate, accessDenied, otherCommitUnderExecution, policyViolated, upToDate, failure, success, lock, unlock, putback, get: *Range*

Checkout(x) = checkout.x $\rightarrow$(success.x $\rightarrow$ SKIP $\square$ accessDenied.x $\rightarrow$ failure.x $\rightarrow$ SKIP)
Update(x) = update.x $\rightarrow$ (success.x $\rightarrow$ SKIP $\square$ accessDenied.x $\rightarrow$ failure.x $\rightarrow$ SKIP $\square$ upToDate.x $\rightarrow$ failure.x $\rightarrow$ SKIP)
Pre-commit$_{\text{succ}}(x)$ = commit.x $\rightarrow$ lock.x $\rightarrow$ SKIP
Pre-commit$_{\text{fail}}(x)$ = commit.x $\rightarrow$ (accessDenied.x $\rightarrow$ failure.x $\rightarrow$ SKIP $\square$ needToUpdate.x $\rightarrow$ failure.x $\rightarrow$ SKIP)
Pre-commit$_{\text{reject}}(x)$ = commit.x $\rightarrow$ otherCommitUnderExecution.x $\rightarrow$ SKIP
Commit$_{\text{succ}}(x)$ = putback.x $\rightarrow$ success.x SKIP
Commit$_{\text{fail}}(x)$ = policyViolated.x $\rightarrow$ failure.x $\rightarrow$ SKIP
Post-commit$_{\text{sync}}(x, z)$ = if x $\neq$ z get.x $\rightarrow$ SKIP else SKIP
Post-commit$_{\text{succ}}(x)$ = unlock.x $\rightarrow$ SKIP
Post-commit$_{\text{fail}}(x)$ = unlock.x $\rightarrow$ SKIP

Server$_{\text{idle}}()$ = $\square$ y:Range @ **Checkout**(y); **Server**$_{\text{idle}}()$
$\square$ **Update**(y); **Server**$_{\text{idle}}()$
$\square$ **Pre-commit**$_{\text{succ}}(y)$; **Server**$_{\text{locked}}(y)$
$\square$ **Pre-commit**$_{\text{fail}}(y)$; **Server**$_{\text{idle}}()$

Server$_{\text{locked}}(x)$ = $\square$ y:Range @ **Checkout**(y); **Server**$_{\text{locked}}(x)$
$\square$ **Update**(y); **Server**$_{\text{locked}}(x)$
$\square$ **Pre-commit**$_{\text{reject}}(y)$; **Server**$_{\text{locked}}(x)$
$\square$ **Commit**$_{\text{succ}}(x)$; **Server**$^{\text{sync}}_{\text{unlocked}}(x, 1)$
$\square$ **Commit**$_{\text{fail}}(x)$; **Server**$_{\text{locked}}(x)$

Server$^{\text{sync}}_{\text{unlocked}}(x, z)$ = $\square$ y:Range @ **Checkout**(y); **Server**$^{\text{sync}}_{\text{unlocked}}(x)$
$\square$ **Update**(y); **Server**$^{\text{sync}}_{\text{unlocked}}(x)$
$\square$ **Pre-commit**$_{\text{reject}}(y)$; **Server**$^{\text{sync}}_{\text{unlocked}}(x)$
$\square$ **Post-commit**$_{\text{sync}}(x, z)$; if z $\neq$ N then **Server**$^{\text{sync}}_{\text{unlocked}}(x, z + 1)$ else **Server**$^{\text{succ}}_{\text{unlocked}}(x)$

Server$^{\text{succ}}_{\text{unlocked}}(x)$ = $\square$ y:Range @ **Checkout**(y); **Server**$^{\text{succ}}_{\text{unlocked}}(x)$
$\square$ **Update**(y); **Server**$^{\text{succ}}_{\text{unlocked}}(x)$
$\square$ **Pre-commit**$_{\text{reject}}(y)$; **Server**$^{\text{succ}}_{\text{unlocked}}(x)$
$\square$ **Post-commit**$_{\text{succ}}(x)$; **Server**$_{\text{idle}}()$

Server$^{\text{fail}}_{\text{unlocked}}(x)$ = $\square$ y:Range @ **Checkout**(y); **Server**$^{\text{fail}}_{\text{unlocked}}(x)$
$\square$ **Update**(y); **Server**$^{\text{fail}}_{\text{unlocked}}(x)$
$\square$ **Pre-commit**$_{\text{reject}}(y)$; **Server**$^{\text{fail}}_{\text{unlocked}}(x)$
$\square$ **Post-commit**$_{\text{fail}}(x)$; **Server**$_{\text{idle}}(x)$

Client$_{\text{checkout}}(x)$ = checkout.$x \rightarrow$ (success.$x \rightarrow$ **Client**$_{\text{idle}}(x)\square$accessDenied.$x \rightarrow$ failure.$x \rightarrow$ **Client**$_{\text{checkout}}(x)$)
Client$_{\text{update}}(x)$ = update.$x \rightarrow$ (success.$x \rightarrow$ **Client**$_{\text{idle}}(x)$
$\square$**AccessDenied**(x); **Client**$_{\text{update}}(x)$)
$\square$upToDate.$x \rightarrow$ success.$x \rightarrow$ **Client**$_{\text{idle}}(x)$)

Client$_{\text{commit}}(x)$ = commit.$x \rightarrow$ (success.$x \rightarrow$ **Client**$_{\text{idle}}(x)$
$\square$accessDenied.$x \rightarrow$ failure.$x \rightarrow$ **Client**$_{\text{idle}}(x)$)
$\square$needToUpdate.$x \rightarrow$ failure.$x \rightarrow$ **Client**$_{\text{update}}(x)$)
$\square$policyViolated.$x \rightarrow$ failure.$x \rightarrow$ **Client**$_{\text{update}}(x)$)
$\square$otherCommitUnderExecution.$x \rightarrow$ failure.$x \rightarrow$ **Client**$_{\text{update}}(x)$)

Client$_{\text{idle}}(x)$ = **Client**$_{\text{commit}}(x)\square$**Client**$_{\text{update}}(x)$

SyncEvents = {|commit,update,checkout,accessDenied,policyViolated,needToUpdate,failure,success|}

Server = **Server**$_{\text{idle}}()$

Clients = ||| y : Range @ **Client**$_{\text{checkout}}(y)$
Collaboration = **Server** $||_{\textit{SyncEvents}}$ **Clients**

Appendix B

Get Transformation Rules

rule	Subtractive Get Reference	Priority	1
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{ReferenceAsset}\}$			
$\{r(s_F, ref, t_F) \mid r(s_F, ref, t_F) \in RA_F,$ $\exists o(s_F, t_s) \in OA_F, o(t_F, t_T) \in OA_F : \textit{trace}(o(s_G, t_s)) = o(s_F, t_s),$ $\textit{trace}(o(t_G, t_t)) = o(t_F, t_t), \nexists r(s_G, ref, t_G) : r(s_G, ref, t_G) \in RA_G$ $\text{permission}_{\text{Eff}}(r(s_G, ref, t_G), read) \neq deny\}$			
Action			
$RA_F := RA_F \setminus \{r(s_F, ref, t_F)\}$			

rule	Subtractive Get Attribute	Priority	2
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{AttributeAsset}\}$			
$\{a(o_F, attr, v') \mid a(o_F, attr, v') \in AA_F, \exists o(o_F, t) : o(o_F, t) \in OA_F$ $\exists o(o_G, t) : o(o_G, t) \in OA_G, trace(o(o_G, t)) = o(o_F, t), \nexists a(o_G, attr, v) :$ $v' = \begin{cases} v, \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) = allow \\ obf(v), \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) = obfuscate \end{cases}, a(o_G, attr, v) \in AA_G\}$			
Action			
$AA_F := AA_F \setminus \{a(o_F, attr, v')\}$			

rule	Subtractive Get Object	Priority	3
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{ObjectAsset}\}$			
$\{o(o_F, t) \mid o(o_F, t) \in OA_F, \nexists o(o_G, t') : \textit{trace}(o(o_G, t')) = o(o_F, t), t = t'$ $\text{permission}_{\text{Eff}}(o(o_G, t'), read) \neq deny\}$			
Action			
$OA_F := OA_F \setminus \{o(o_F, t)\}, \textit{trace} \setminus o(o_G, t') \mid \textit{trace}(o(o_G, t')) = o(o_F, t)$			

rule	Additive Get Object	Priority	4
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{ObjectAsset}\}^2$			
$\{o(o_G, t), o(o_F, t') \mid o(o_G, t) \in OA_G, \text{permission}_{\text{Eff}}(o(o_G, t), \text{read}) \neq \text{deny},$ $\nexists o(o_F, t') \in OA_F : \textit{trace}(o(o_G, t)) = o(o_F, t'), t = t'\}$			
Action			
$OA_F := OA_F \cup \{o(o_F, t')\}, trace(o(o_G, t)) := o(o_F, t')$			

rule	Additive Get Attribute	Priority	5
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{AttributeAsset}\}$			
$\{a(o_F, attr, v') \mid a(o_G, attr, v) \in AA_G, \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) \neq deny,$ $\exists o(o_F, t) : o(o_F, t) \in AA_F, trace(o(o_G, t)) = o(o_F, t), \nexists a(o_F, attr, v') :$ $v' = \begin{cases} v, \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) = allow \\ obf(v), \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) = obfuscate \end{cases}, a(o_F, attr, v') \in AA_F\}$			
Action			
$AA_F := AA_F \cup \{a(o_F, attr, v')\}$			

rule	Additive Get Reference	Priority	6
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{ReferenceAsset}\}$			
$\{r(s_F, ref, t_F) \mid r(s_G, ref, t_G) \in RA_G, permission_{\text{Eff}}(r(s_G, ref, t_G), read) \neq deny,$ $\exists o(s_G, t_s), o(t_G, t_t) : trace(o(s_G, t_s)) = o(s_F, t_s),$ $trace(o(t_G, t_t)) = o(t_F, t_t), \nexists r(s_F, ref, t_F) : r(s_F, ref, t_F) \in RA_F\}$			
Action			
$RA_F := RA_F \cup \{r(s_F, ref, t_F)\}$			

Appendix C

PutBack Transformation Rules

rule	Subtractive PutBack Reference	Priority	1
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{ReferenceAsset}\}$			
$\{r(s_G, ref, t_G) \mid r(s_G, ref, t_G) \in RA_G, permission_{\text{Eff}}(r(s_G, ref, t_G), read) \neq deny,$ $\exists o(s_G, t_s), o(t_G, t_t) : trace(o(s_G, t_s)) = o(s_F, t_s),$ $trace(o(t_G, t_t)) = o(t_F, t_t), \nexists r(s_F, ref, t_F) : r(s_F, ref, t_F) \in RA_F\}$			
Action			
If $\text{permission}_{\text{Eff}}(r(s_G, ref, t_G), write) \neq deny$ then $RA_G := RA_G \setminus r(s_G, ref, t_G)$ else↙↘			

rule	Subtractive PutBack Attribute	Priority	2
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{AttributeAsset}\}$			
$\{a(o_G, attr, v) \mid \nexists a(o_F, attr, v') :$ $\exists o(o_F, t) : o(o_F, t) \in OA_F, trace(o(o_G, t)) = o(o_F, t),$ $v = \begin{cases} v', \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) = allow \\ obf^{-1}(v'), \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) = obfuscate \end{cases}, a(o_G, attr, v) \in AA_G\}$			
Action			
If $\text{permission}_{\text{Eff}}(a(o_G, attr, v), write) \neq deny$ then $AA_G := AA_G \setminus a(o_G, attr, v)$ else↙↘			

rule	Subtractive PutBack Object	Priority	3
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{ObjectAsset}\}$			
$\{o(o_G, t) \mid o(o_G, t) \in OA_G, \text{permission}_{\text{Eff}}(o(o_G, t), \text{read}) \neq \text{deny},$ $\nexists o(o_F, t') : \textit{trace}(o(o_G, t)) = o(o_F, t'), t = t'\}$			
Action			
If $\text{permission}_{\text{Eff}}(o(o_G, type), write) \neq deny$ then $OA_G := OA_G \setminus o(o_G, t)$, $\textit{trace} \setminus o(o_G, t) \mid \textit{trace}(o(o_G, t)) = o(o_F, t)$ else↙↘			

rule	Additive PutBack Object	Priority	4
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{ObjectAsset}\}^2$			
$\{o(o_F, t), o(o_G, t) \mid o(o_F, t) \in OA_F, \nexists o(o_G, t) : \textit{trace}(o(o_G, t)) = o(o_F, t)\}$			
Action			
$OA_G := OA_G \cup \{o(o_G, t)\}$ If $\text{permission}_{\text{Eff}}(o(o_G, t), write) \neq deny$ then $trace(o(o_G, t)) := o(o_F, t)$ else↙↘			

rule	Additive PutBack Attribute	Priority	5
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{AttributeAsset}\}$			
$\{a(o_G, attr, v) \mid a(o_F, attr, v') \in AA_F, \exists o(o_F, t) : o(o_F, t) \in AA_F$ $\exists o(o_G, t) : o(o_G, t) \in OA_G, trace(o(o_G, t)) = o(o_F, t), \nexists a(o_G, attr, v) :$ $v = \begin{cases} v', \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) = allow \\ obf^{-1}(v'), \text{permission}_{\text{Eff}}(a(o_G, attr, v), read) = obfuscate \end{cases}, a(o_G, attr, v) \in AA_G\}$			
Action			
$AA_G := AA_G \cup \{\textit{AttributeAsset}(o_G, att, v)\}$ If ($\text{permission}_{\text{Eff}}(a(o_G, att, v), write) =$ deny) then↙↘			

rule	Additive PutBack Reference	Priority	6
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \text{permission}_{\text{Eff}}) \rightarrow \{\textit{ReferenceAsset}\}$			
$\{r(s_G, ref, t_G) \mid r(s_F, ref, t_F) \in RA_F,$ $\exists o(s_F, t_s), o(t_F, t_t) \in OA_F : trace(o(s_G, t_s)) = o(s_F, t_s),$ $trace(o(t_G, t_t)) = o(t_F, t_t), \nexists r(s_G, ref, t_G) : r(s_G, ref, t_G) \in RA_G\}$			
Action			
$RA_G := RA_G \cup \{r(s_G, ref, t_G)\}$ If $\text{permission}_{\text{Eff}}(r(s_G, ref, t_G), write) = deny$ then↙↘			