Property-based Locking in Collaborative Modeling

Csaba Debreceni[1,2], Gábor Bergmann[1,2], István Ráth[1] and Dániel Varró[1,2,3]
[1]Budapest University of Technologies and Economics, Department of Measurement and Information Systems, Hungary
[2]MTA-BME Lendület Research Group on Cyber-Physical Systems, Hungary
[3]McGill University of Montreal, Department of Electrical and Computer Engineering, Canada
Email: {debreceni,bergmann,rath,varro}@mit.bme.hu

***Abstract*—Large-scale model-driven engineering projects are carried out collaboratively. Enabling a high degree of concurrency is required to make the traditionally rigid development processes more agile. The increasing number of collaborators increases the probability of introducing conflicts which need to be resolved manually by the collaborators. In case of highly interdependent models, avoiding conflicts by the use of locks can save valuable time. However, traditional locking techniques such as *fragment-based* and *object-based* strategies may impose unnecessary restrictions on editing, which can decrease the efficiency of collaboration.**

In this paper, we propose a property-based locking approach that generalizes traditional locking techniques, and further allows more fine-grained locks in order to restrict modifications only when necessary. A lock is considered to be violated if a match appears or disappears for its associated graph pattern (formula), which captures the property of the model that the upcoming edit transaction can be freely executed. An initial evaluation has been carried out using a case study of the MONDO EU project.

I. Introduction

Context. Many large-scale software industries, from avionics to automotive or telecommunication domains, are often faced with the challenge of enabling the high degree of collaborative and concurrent development required to meet the aggressive delivery schedules while still maintaining a high quality of service necessitated by certification standards.

The adoption of model driven engineering (MDE) has been steadily increasing in the recent years [1] where the primary development artifacts are models. The use of models intensifies collaboration between geographically distributed engineers (system engineers, software engineers, hardware engineers, specialists, etc.) via model repositories to significantly enhance productivity and reduce time to market.

Effective collaborative development requires the ability to resolve or prevent interference between collaborators potentially making updates to the same part or fragment of the system. The interdependence and granularity within a model makes conflicts easy to introduce and hard to resolve in MDE, when compared to traditional software development.

Problem statement. *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* [2]–[6] 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 [2] or Git [5] extended by Gitolite [6]). 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* [7], [8] 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 paper, they do not scale well with the increasing number of collaborators.

Goals. In our preliminary work [9], we introduced the concept of *property-based (*PB*) 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 main objective of this paper is to propose a realization of *PB locking* that enables a high degree of collaborative

This paper is partially supported by the EU Commission with project MONDO (FP7-ICT-2013-10), no. 611125., the MTA-BME Lendület 2015 Research Group on Cyber-Physical Systems and NSERC RGPIN-04573-16. The second author was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

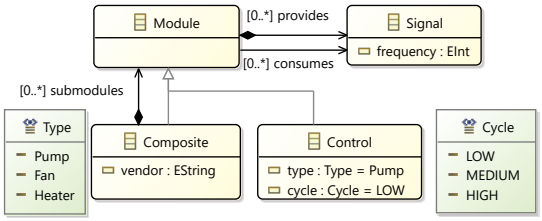


Fig. 1. Simplified Metamodel of Wind Turbine Controllers

development. In particular, we aim to address three goals:

G1 *Describe Locks as Properties.*
The solution shall provide a way to describe complex properties suitable for specifying locks.

G2 *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.

G3 *Support for Existing Locking Strategies.*
The solution must provide means to support existing locking strategies (such as FB and OB).

Contributions and Added Value. In this paper, we propose an underlying infrastructure realizing a property-based locking strategy. In our approach, **G1** is addressed using graph patterns. To cover **G2**, we propose an algorithm using a pattern matcher to notify upon violating a property owned by another collaborator and undo the executed operations. To address **G3**, we present how graph patterns can describe traditional locks and how our algorithm can enforce them. The proposed approach is illustrated on a case study of the MONDO EU project and evaluated by simulation with increasing number of collaborators concurrently developing models of various size.

Property-based locking can be a core service of a collaborative modeling framework that may provide model refactoring / editing tools equipped with their own lock definitions; as well as pre-defined domain-specific properties or a user-friendly interface to ease the definition of locks (which are out of scope for the current paper).

II. Case Study

A. Modeling Language

Our property-based locking technique will be illustrated using a simplified version of a modeling language for developers of *offshore wind turbine controllers*, which was one of the case studies of the MONDO EU FP7 project [10]. The metamodel, defined in EMF [11] and depicted by Fig. 1, describes how the system is modeled as `Modules` providing and consuming `Signals` produced at various frequencies. `Modules` are organized in a containment hierarchy of `Composite` modules shipped by external *vendors*. They ultimately contain `Control` unit modules responsible for a given type of physical device (such as *pumps*, *heaters* or *fans*) on a certain *cycle* level (*low*, *medium* or *high*).

Example. A sample instance model containing a hierarchy of 4 `Composite` modules and 4 `Control` units providing 4 `Signals` altogether is shown on Fig. 2. Boxes represent objects (with attribute values as entries within the box). Bold arrows represent the containment edges, while thin arrows represent cross-references. The model is divided into 4 model fragments represented by dashed borders surrounding objects.

B. 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 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.

Operations. Each operation is visualized in Fig. 3 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 Fig. 3a.

(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 Fig. 3b.

(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 Fig. 3c.

Example. 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 $v' = "E"$.

C. Usage Scenario

The wind turbine control model can be hosted on a collaboration server [3], [7], [12] 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. 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

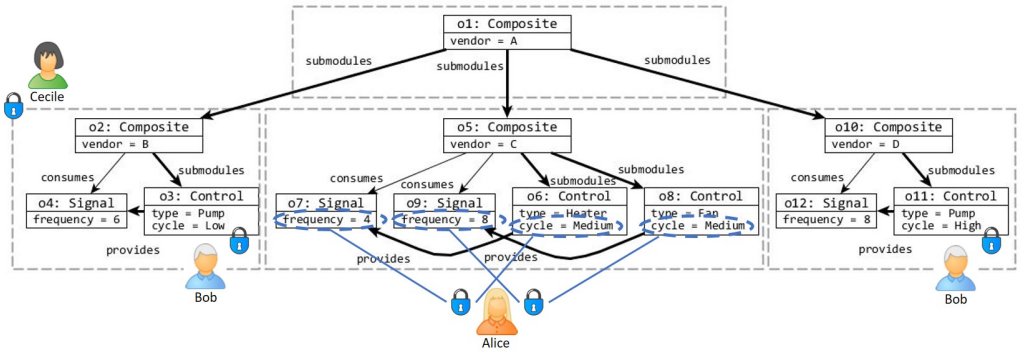
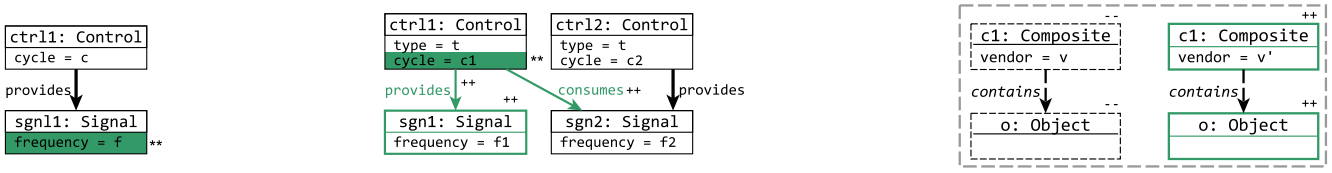


Fig. 2. Sample Wind Turbine Instance Model (stored in separate fragments denoted by dashed border)



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

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

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

Fig. 3. Sample Operations (++: creation, –: deletion, **: update)

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 Fig. 2.

D. 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. 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. 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**.

III. Property-based Locks

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 **G1** and capture exactly which changes violate the property, we adapt the concepts of a *change query* [13]. 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 Fig. 4. Sec. III-A introduces our choice for specifying such model queries, Sec. III-B follows with the definition of an entire lock. Sec. III-C and Sec. III-D show how the proposed formalism is a generalization of the FB and OB approaches.

A. Properties Captured by Graph Patterns

Model queries are formulae over models, declarative descriptions of a read-only computation. In this paper, 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

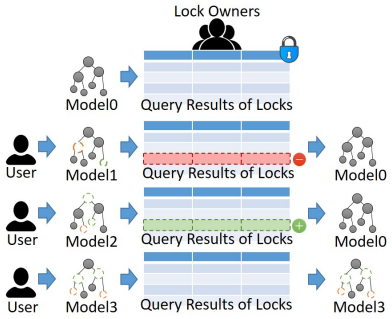


Fig. 4. Behavior of Property-Based Locks (in case of invariant property)

VIATRA Query framework [14], 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. Listing 1 displays a graph pattern in the declarative VIATRA Query [15] syntax, expressing a property of the wind turbine model. The pattern `signals` (pattern name after the `pattern` keyword) selects triples of $\langle$`sig`, `frq`, `cycle`$\rangle$ (pattern parameters between parentheses) where `sig` is an instance of the class `Signal` with its frequency attribute set to `frq` (Line 2), and there is a `Control` instance `ctrl` that provides `sig` (Line 3) and has its cycle attribute set to `cycle` (Line 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 1. Graph Pattern to Lock Signals

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).

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

After a modification is applied to the model, the query needs to be reevaluated. New matches may *appear* in the match set or existing matches may *disappear* from the match set.

Example. The $\mathbb{MS}_{\text{signals}}$ of pattern `signals` (Listing 1) contains the tuples $\langle o4, 6, low\rangle$, $\langle o7, 4, medium\rangle$, $\langle o9, 8, medium\rangle$ and $\langle o12, 8, high\rangle$. If the *cycle* attribute is bound to $medium$, the match set is reduced to $\langle o7, 4, medium\rangle$ and $\langle o9, 8, medium\rangle$.

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

B. 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.

$$lock = \langle owner, changeQuery\rangle$$

We use simple change queries [13] 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:

$$directions \subseteq \{d, a\}$$
$$changeQuery = \langle pattern, bindings, directions\rangle$$

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

Example. $Alice$ needs to define and put the following lock on the model using the pattern Listing 1 to prevent certain modifications described in Sec. II-C:

$$lock\langle Alice, \langle \texttt{signals}, \{cycle = medium\}, \{d, a\}\rangle\rangle \quad (1)$$

The $\mathbb{MS}_{\text{signals}}$ of pattern `signals` consists of $\langle o7, 4, medium\rangle$ and $\langle o9, 8, medium\rangle$.

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

C. 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. 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 2. Graph Patterns to Select Controls and Their Feature Values

$$lock\langle Bob, \langle \texttt{control}, ctrl = o3, \{d, a\}\rangle\rangle \quad (1)$$
$$lock\langle Bob, \langle \texttt{control}, ctrl = o11, \{d, a\}\rangle\rangle \quad (2)$$
$$lock\langle Bob, \langle \texttt{type}, ctrl = o3, \{d, a\}\rangle\rangle \quad (3)$$
$$lock\langle Bob, \langle \texttt{type}, ctrl = o11, \{d, a\}\rangle\rangle \quad (4)$$
$$lock\langle Bob, \langle \texttt{cycle}, ctrl = o3, \{d, a\}\rangle\rangle \quad (5)$$
$$lock\langle Bob, \langle \texttt{cycle}, ctrl = o11, \{d, a\}\rangle\rangle \quad (6)$$
$$lock\langle Bob, \langle \texttt{provides}, ctrl = o3, \{d, a\}\rangle\rangle \quad (7)$$
$$lock\langle Bob, \langle \texttt{provides}, ctrl = o11, \{d, a\}\rangle\rangle \quad (8)$$
$$lock\langle Bob, \langle \texttt{consumes}, ctrl = o3, \{d, a\}\rangle\rangle \quad (9)$$
$$lock\langle Bob, \langle \texttt{consumes}, ctrl = o11, \{d, a\}\rangle\rangle \quad (10)$$

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

(3) The $\mathbb{MS}_{\text{type}}$ 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 $\mathbb{MS}_{\text{type}}$ and $\langle o3, Fan\rangle$ appears.

D. 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. $Cecile$ requests to lock the fragment whose root is $o2$. Listing 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 3. Fragment-based Lock Patterns

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

$$lock\langle Cecile, \langle \texttt{fragment}, root = o2, \{d, a\}\rangle\rangle \quad (1)$$

$$lock\langle Cecile, \langle \texttt{cycleInFragment}, root = o2, \{d, a\}\rangle\rangle \quad (2)$$

...

The $\mathbb{MS}_{\text{fragment}}$ of pattern `fragment` has the following matches $\langle o2, Composite\rangle$, $\langle o3, Control\rangle$, $\langle o4, Signal\rangle$. If the signal $o4$ is deleted, the match $\langle o4, Signal\rangle$ disappears. When a new signal o' is created under $o3$, a new match $\langle o', Signal\rangle$ appears.

The $\mathbb{MS}_{\text{cycleInFragment}}$ 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 $\langle o3, low\rangle$ disappears and a new match $\langle o3.high\rangle$ appears.

IV. 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 Sec. V-H for discussion of *lock-lock compatibility* (whether a new lock request should be granted or rejected given the set of active locks).

A. 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 $model$.

To evaluate the $\mathbb{MS}$ of a pattern, a *pattern matcher* is required. We define the pattern matcher as a function PM that takes a graph pattern, parameter bindings of the pattern and a model. It calculates the $\mathbb{MS}_{\text{pattern}}$ of $pattern$ restricted by parameter $Bindings$ on the given model.

Now we present an algorithm depicted in Alg. 1 to enforce PB locks as it is address by goal **G2**. 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 $\mathbb{MS}_{\text{lock}}$ 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 $\mathbb{MS}'_{\text{lock}}$ of each lock is reevaluated and compared to $\mathbb{MS}_{\text{lock}}$ calculated in Phase 1. The $transaction$ is rolled back and the function terminates if $\mathbb{MS}_{\text{lock}}$ is modified in a direction that is disallowed by the lock.

Algorithm 1 Enforcement of Property-based Locks

function ENFORCELOCKS(model, user, Locks, Operations)
  ▷ **Phase 1:** Evaluate match sets of relevant locks
  $Map\langle lock, \mathbb{MS}\rangle\ relevantMap \leftarrow \emptyset$
  for all $lock$ in Locks **do**
    if $lock.owner \neq$ user **then**
      $cq \leftarrow lock.changeQuery$
      $\mathbb{MS}_{\text{lock}} \leftarrow$ PM$(cq.pattern, cq.bindings,$ model$)$
      $relevantMap$.put$(lock, \mathbb{MS}_{\text{lock}})$
  ▷ **Phase 2:** $Operations$ are wrapped into a transaction and the $transaction$ is attempted to be executed.
  $transaction \leftarrow$ TRANSACTION(Operations)
  if EXECUTE($transaction$, model) fails **then**
    ▷ The execution can fail if access control rules
    ▷ are violated or conflicts are introduced
    return
  ▷ **Phase 3:** Reevaluate the match sets, check violations and roll back the $transaction$ if it is required
  for all $\langle lock, \mathbb{MS}_{\text{lock}}\rangle$ in $relevantMap$ **do**
    ▷ Reevaluate the match sets of relevant locks
    $cq \leftarrow lock.changeQuery$
    $\mathbb{MS}'_{\text{lock}} \leftarrow$ PM$(cq.pattern, cq.bindings,$ model$)$
    if $\mathbb{MS}'_{\text{lock}} \setminus \mathbb{MS}_{\text{lock}} \neq \emptyset$ and $a \in ctx.directions$ or $\mathbb{MS}_{\text{lock}} \setminus \mathbb{MS}'_{\text{lock}} \neq \emptyset$ and $d \in ctx.directions$ **then**
      ROLLBACK($transaction$, model)
      return
  return

Note that the practical execution of *Phase 1* and *Phase 2* can be simplified using incremental evaluation techniques [16]. Additionally, we assume the algorithm is executed by

a collaboration framework (e.g. [12]) on the server side that ensures any other submission attempt including lock addition will be rejected while the algorithm is under execution.

B. 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. □

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 $\subseteq (Users \setminus \{u\})$.*

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

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''.*

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

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 $\subseteq (Users \setminus \{u\})$.*

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

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 [17] in details, but out of scope for the current paper). *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. □

V. 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 of this paper.

The evaluation is based on stochastic simulation of collaborators modeled as a Discrete Event System Specification (DEVS) [18]. 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:

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

A. Behavior of Collaborators

The behavior of each collaborator is modeled as a state machine depicted in Fig. 5. 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

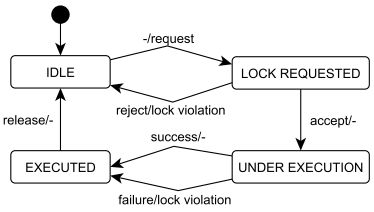


Fig. 5. Behavior of a Collaborator

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.

B. Operations of Collaborators

In order to obtain repeatable (cyclic) behavior for stochastic simulation, the operations introduced in Sec. II-B 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.

C. Model Synthesis

For the simulation, we used the metamodel shown in Fig. 1 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 Fig. 2 consists of 4 *Composite*s, *Signal*s and *Control* units. Each copy of the structure contains at most one other 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.

D. 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 I 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.

TABLE I
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	$\lambda^{\mathbf{M}}_{\text{WAIT}} = \frac{1}{4}$	$4h$	$\lambda^{\mathbf{M}}_{\text{EXEC}} = \frac{1}{1}$	$1h$

E. Coupled DEVS Model

The coupled DEVS model [18] 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 Fig. 5. The time advances (ta) of `Idle` and `Under Execution` states are parametrized for each type of collaborators (shown in Table I) 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.

F. 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 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*.

G. 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 Fig. 6 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.

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.

H. Discussion of Limitations

User Experience. We believe that the proposed approach of property-based locking (shown in Sec. III-C and Sec. III-D to be a generalization of OB and FB) 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 parametrize a lock to request. The detailed evaluation of user experience is out of scope for the current paper.

Lock-lock Compatibility. As indicated before in Sec. IV, 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 Fig. 5 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 according to Sec. III-C, we do have a notion of lock-lock compatibility, which a sufficiently careful implementation could enforce.

I. Integration into MONDO Collaboration Framework

PB locking approach is successfully implemented and integrated into the MONDO Collaboration Framework [12]. The framework extends existing version control systems (VCS) such as SVN [2] to support fine-grained access control rules [17] and PB locking strategy.

Traditional file-level FB locking behavior of VCSs is broadened with PB locking strategy using *hook* methods triggered by repository events e.g. commit. When a collaborator commits her changes to a certain repository the related *hook* method is executed to evaluate the locks and reject the commit if a PB lock is violated.

VI. Related Work

A. 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 [19], AMOR [20], Eclipse Modeling Team Framework [21] or EMFStore [3], rely on traditional version control systems using file-based locking with contributors committing large deltas of work.

Model repositories such as CDO [7], MetaEdit+ [22] and Morsa [8], 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 [23], CoolModes [24], SpacEclipse [25], WebGME [26] and ATOMPM [27], 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

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

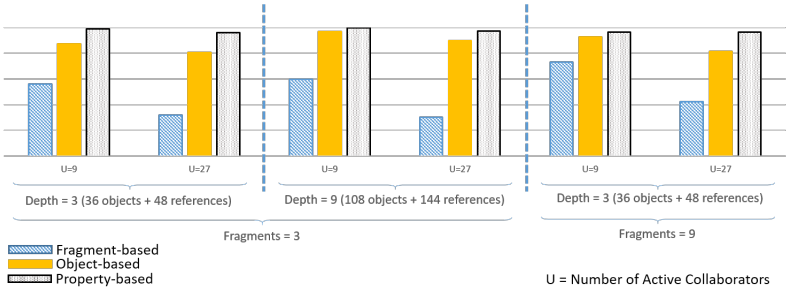


Fig. 6. Success Rates of the Simulation - Property-based (filled), Object-based (crossed) and Fragment-based (dotted)

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.

B. 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 [28]. 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 [29] 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 [9], 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.

C. 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 [30], [31] and graph-based [32]–[34] 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. [2], [3]) due to the atomic transactional executions of changes.

VII. Conclusion and Future Work

In this paper, we introduced an underlying infrastructure to realize a property-based locking strategy as a common generalization of existing fragment-based and object-based locking approaches. Our property-based locking approach enables to define fine-grained locks as properties allowing only modifications that preserve the property.

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. Our locking strategy prevents modifications that change the result set make a match of the pattern appear or disappear.

We have carried out an initial evaluation with respect to overlocking simulating collaborators working on the same model. The results show that our approach locks fewer number of elements compared to traditional locking approaches to further enhance collaboration.

Our locking approach has been integrated into the collaborative tools developed within the MONDO project for off-line collaboration. As future work, 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.

VIII. Acknowledgment

We would like to acknowledge Marsha Chechik, Fabiano Dalpiaz, Jennifer Horkoff and Rick Salay, for the fruitful discussions when preparing [9] which contained some core ideas for this paper.

References

[1] J. Whittle, J. E. Hutchinson, and M. Rouncefield, "The State of Practice in Model-Driven Engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014. [Online]. Available: http://dx.doi.org/10.1109/MS.2013.65

[2] Apache, "Subversion," https://subversion.apache.org/.

[3] M. Koegel and J. Helming, "Emfstore: a model repository for EMF models," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 307–308. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810364

[4] The ModelCVS project, "A Semantic Infrastructure for Model-based Tool Integration," 2006, http://modelcvs.org.

[5] S. Chacon and B. Straub, *Pro git*. Apress, 2014.

[6] S. Chamarty, *Gitolite Essentials*. Packt Publishing Ltd, 2014.

[7] The Eclipse Foundation, "CDO," http://www.eclipse.org/cdo.

[8] J. Espinazo-Pagán, J. S. Cuadrado, and J. G. 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*, 2011, pp. 77–92. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24485-8_7

[9] M. Chechik, F. Dalpiaz, C. Debreceni, J. Horkoff, I. Ráth, R. Salay, and D. Varró, "Property-based methods for collaborative model development," in *Joint Proc. of the 3rd Int. Workshop on the Glob. Of Modeling Lang. and the 9th Int, Workshop on Multi-Paradigm Modeling co-located with ACM/IEEE 18th Int. Conf. on Model Driven Engineering Languages and Systems, GEMOC+MPM@MoDELS 2015, Ottawa, Canada, September 28, 2015.*, 2015, pp. 1–7. [Online]. Available: http://ceur-ws.org/Vol-1511/paper-01.pdf

[10] A. Gmez, X. Mendialdua, G. Bergmann, J. Cabot, C. Debreceni, A. Garmendia, D. S. Kolovos, J. de Lara, and S. Trujillo, "On the opportunities of scalable modeling technologies: An experience report on wind turbines control applications development," in *13th European Conference on Modelling Foundations and Applications*, Eindhoven, The Netherlands, In press 2017.

[11] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[12] C. Debreceni, G. Bergmann, M. Búr, I. Ráth, and D. Varró, "The mondo collaboration framework: Secure collaborative modeling over existing version control systems," *European Software Engineering Conference and the ACM SIG-SOFT Symposium on the Foundations of Software Engineering, Padeborn, Germany, September 0408*, pp. 1–5, 2017.

[13] G. Bergmann, I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations. change (in) the rule to rule the change." *Software and Systems Modeling*, vol. 11, pp. 431–461, 2012 2012. [Online]. Available: http://dx.doi.org/10.1007/s10270-011-0197-9

[14] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. 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*, 2015, pp. 101–110. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21155-8_8

[15] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, "A graph query language for EMF models," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2011, pp. 167–182.

[16] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró, "EMF-IncQuery: An integrated development environment for live model queries," *Sci. Comput. Program.*, vol. 98, pp. 80–99, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2014.01.004

[17] G. Bergmann, C. Debreceni, I. Ráth, and D. 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*, 2016, pp. 351–361. [Online]. Available: http://dl.acm.org/citation.cfm?id=2976793

[18] A. I. Concepcion and B. P. Zeigler, "DEVS formalism: A framework for hierarchical model development," *IEEE Trans. Software Eng.*, vol. 14, no. 2, pp. 228–241, 1988. [Online]. Available: http://dx.doi.org/10.1109/32.4640

[19] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger, "Towards a Semantic Infrastructure Supporting Model-Based Tool Integration," in *Proc. of GaMMa@ICSE'06*, 2006.

[20] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer, "AMOR - towards adaptable model versioning," in *1st Int. Workshop on Model Co-Evolution and Consistency Management*, 2008.

[21] Eclipse, "Modeling team framework proposal," 2011, http://www.eclipse.org/proposals/mtf/.

[22] J. 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*, 2016, pp. 41–45. [Online]. Available: http://dl.acm.org/citation.cfm?id=2997379

[23] M. Dirix, A. Muller, and V. Aranega, "Genmymodel: an online uml case tool," in *ECOOP*, 2013.

[24] N. 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)*. Sydney, Australia: SIT, 2003, pp. 89–94.

[25] J. Gallardo, A. I. Molina, C. Bravo, M. A. Redondo, and C. A. Collazos, "An ontological conceptualization approach for awareness in domain-independent collaborative modeling systems: Application to a model-driven development method," *Expert Syst. Appl.*, vol. 38, no. 2, pp. 1099–1118, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2010.05.005

[26] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, and Á. 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.*, 2014, pp. 41–60. [Online]. Available: http://ceur-ws.org/Vol-1237/paper5.pdf

[27] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. V. Mierlo, and H. 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.*, 2013, pp. 21–25. [Online]. Available: http://ceur-ws.org/Vol-1115/demo4.pdf

[28] J. Rekimoto, "CSCW platform system teidan and its concurrency control algorithm," *Advances in Software Science and Technology*, vol. 5, pp. 239–255, 1993.

[29] P. 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*, 1996, pp. 268–277. [Online]. Available: http://doi.acm.org/10.1145/240080.240300

[30] "Oracle." [Online]. Available: https://www.oracle.com/

[31] "MS-SQL." [Online]. Available: https://tinyurl.com/sql-server16

[32] "Neo4J." [Online]. Available: https://www.neo4j.com

[33] "OrientDB." [Online]. Available: https://www.orientdb.com

[34] "Stardog." [Online]. Available: https://www.stardog.com