JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Policy-Driven Change Management in Model Based Systems Engineering

Anish Bhobe, Dominique Blouin, and Laurent Pautet
Télécom Paris, France
Institut Polytechnique de Paris, France

ABSTRACT Model-Based Systems Engineering (MBSE) uses models for designing, validating and synthesizing complex systems. Different models for large systems are synchronized via transformations, but current Model Management (MoM) approaches lack mechanisms to control or audit changes propagation. Managing such changes is very challenging, especially when the propagated modifications are unauthorized or rejected by downstream teams.

This paper introduces Senate, the first fine-grained change propagation control framework based on declarative change policies. Senate provides a declarative domain-specific language to specify policies to provide consistent change propagation in scenarios where changes may be rejected by models. These policies govern which users are permitted to create, modify or delete model fragments based on the affected elements, their types, or if they match given patterns. By enforcing these policies during consistency management, Senate prevents unauthorized or undesired changes from being propagated, thus reducing the delay between introduction of the inconsistency and its discovery during review. We implement and evaluate a prototype of Senate which shows the feasibility and scalability of the approach in an non-intrusive way, without requiring modifications to the models or the consistency management approaches required. We evaluate the design to show that declarative change policies are generalizable, non-intrusive to existing tools, and scalable with respect to the size of models and number of changes, and thus can be integrated into existing consistency management workflows in multi-model collaborative MBSE scenarios.

KEYWORDS Model Based Systems Engineering, Model Management, Change Policy

1. Introduction

Model Based Systems Engineering (MBSE) is increasingly adopted in engineering systems such as those of avionics, automotive, medical instrumentation to alleviate the issues caused by increasing systems complexity and support their development and maintenance. MBSE leverages formal models to aid requirement specification, system design, analysis, verification and validation (V&V), simulation and reliable code-generation throughout the development cycle (Cederbladh et al. 2024; Walden et al. 2015).

JOT reference format:
Anish Bhobe, Dominique Blouin, and Laurent Pautet. *Policy-Driven Change Management in Model Based Systems Engineering*. Journal of Object Technology. Vol. 25, No. 3, 2026.
http://dx.doi.org/10.5381/jot.2026.25.3.a20

Multi-Paradigm Modeling (MPM) (Van Tendeloo 2018; Amrani et al. 2021; Barišić et al. 2022) provides a systematic approach to modeling across different paradigms, requiring that all aspects of a system are modeled explicitly using the most appropriate formalism and abstraction level. The domain specific nature of models implies the need for different models and formalisms for different tasks. Each model captures different aspects of the same system and thus shares some information which must be kept consistent as models are manipulated by several teams, often concurrently. Manually synchronizing these heterogeneous models is both difficult and error-prone.

Model Management (MoM) aims at supporting global interactions across multiple models (Bézivin et al. 2005) and facilitate tasks such as consistency management, V&V, and analysis on a system-wide level, enabling what is often referred to as 'modeling in the large'. MoM thus plays a key role in

enabling and structuring effective MPM techniques.

Complex systems development involves multiple components that are developed by different teams or organizations. Each team is responsible for some of the components and must control changes made to the components in their responsibility, i.e. the team owns the said components. This is important not only in MBSE, but also for large software development projects where code ownership must be properly managed in order to reduce faults (Bird et al. 2011; Thongtanunam et al. 2016). Current industry practices use MoM tools that synchronize the different models during a dedicated synchronization step. System integrators define guidelines for the overall system that are dispatched to the domain engineers (Caron et al. 2020). But automatic constraint checks and validation takes place only during dedicated synchronization steps planned in the development cycle. As the changes are propagated automatically, tracking the source of the change becomes a critical issue. In case any models become inconsistent with the others, they will continue to be modified under the inconsistent state until the integration step, where developers will have to amend all the changes during the entire span since the last integration. Thus there is a need for an improved workflow that automates identification of inconsistencies as they arise, to speed up iteration.

Due to confidentiality requirements, the users that made a change that propagated to a model may not be able to see the impact to other models, and will need collaboration to correct these changes, thus leading to overhead. While Role-Based Access Control (RBAC) for MBSE is a well studied topic, this is done with respect to collaborative modeling on a set of homogeneous or directly modified models (Debreceni et al. 2019, 2017; Martínez et al. 2018) with focus on confidentiality and security as opposed to ease of collaboration. As such, there is a need to study the fine-grained role-based constraints in a multi-model context, where an access violation is not directly caused by a user, but by other tools as a direct consequence of the actions of the user.

This paper introduces Senate, a fine-grained change propagation control framework. Senate provides a domain specific language (DSL) to specify change policies that define which users may create, modify, or delete model elements across a set of synchronized models. The approach formalizes these constraints over model elements and synchronizations to prevent undesired or unauthorized changes without requiring global model access or centralized control. The feasibility of the approach is demonstrated through representative co-engineering scenarios.

The paper is divided as follows: Section 2 introduces prerequisite knowledge required to understand our problem and approach. Section 3 introduces a running example to illustrate the needs for change policies. Section 4 precisely specifies the problem that Senate tackles. Sections 5, 6 and 7 respectively present our approach, its evaluation, and its limitations. Section 8 describes the related work and where senate is placed with respect to it. Finally, Section 9 concludes the paper.

2. Background

For managing consistency of a set of models, state-of-the-art MoM approaches use automatic model transformations (MT) to propagate changes to synchronize the data within the models. We describe these concepts and their relevance to our problem below.

2.1. Model Transformation and Synchronization

An MT is an operation that converts one or several models into other model(s). For the purpose of synchronization, incremental model transformations are used, which update the target model based on the changes made to the source model. The information content of heterogeneous models differs from each other. So, instead of creating an entire target model from a source model, incremental MTs identify the changes applied in the source model, and convert it into a set of changes that are applied to the target model (Xiong et al. 2007). We refer to a sequence of changes applied to a model as a delta, and we represent a delta on a model M as $\Delta(M)$.

For any MT that synchronizes two models (A, B) that conform to different metamodels, the operations in $\Delta(A)$ do not necessarily correspond to the same kind of operations in $\Delta(B)$. A trivial example of this is the mapping between a family and a tax form, where creation of a child in the family only requires an update in the count of tax dependents. Here, a structural change (creating a child element) corresponds to an attribute change (update count of dependents). As such, models cannot be directly correlated to each other without taking into account the specification and implementation of the MT.

2.2. Model Management

Model management (MoM) facilitates operations on a set of models as a whole. These operations include, among others, automated inter-model consistency management, multi-model view generation, and global validation. MoM approaches can be generally categorized into Model Integration, Model Unification and Model Federation (International Organization for Standardization 2014).

Model Integration requires significant investment in developing a bespoke common language to cover all paradigms of the integrated models. This requires large development efforts for a language that depends on the project needs, and also for the tooling for this bespoke language.

Model Unification instead uses a pivot model that contains the shared information in a pivot language that is then synchronized with the individual domain specific models (DSM). While this allows use of more appropriate languages for the different engineering domains and levels of abstractions in MBSE, it still requires development of the pivot language and the tooling to synchronize the pivot model to all the individual DSMs.

Model Federation attempts to resolve these issues by using DSMs for development, and directly managing consistency between the models through individual links. Each DSM can thus reuse their existing tooling, and most of the investment is dedicated to the specification of such links. As a federated MoM approach itself does not dictate a common language of any kind, this allows flexibility in the implementation of the links. Due

to this, the inter-model links can also delegate synchronization to the MTs implemented with different tools. better suited for the job such as in case of hierarchical megamodels (Seibel et al. 2010).

However, due to the individual links, a model is not directly connected to all the models that it can affect. As such, in cases such as the one illustrated in Figure 1, a change can propagate to an indirectly linked model where consistency management still requires additional effort to ensure changes do not cause unintended side-effects downstream. While federation-based MoM addresses many practical issues with the previous methods, the state-of-the-art approaches cannot control and constrain the propagation of changes.

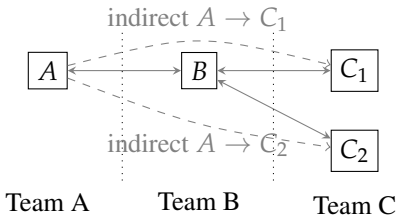


Figure 1 Example collaboration scenario

3. Running Example

In order to explain our approach, we consider a simple federated scenario in which multiple teams collaborate on the development of an aircraft, using industrial languages such as FACE and AADL to illustrate the types of changes that occur in realistic industry settings. FACE (The Open Group FACE Consortium 2025) is a modeling language describing software components: it specifies Portable Components (PCs), Platform-Specific Components (PSCs), their threads, and the data exchanged between them, focusing on software-level communication and timing constraints but without modeling hardware or execution platforms. AADL (SAE International 2022) is an SAE standard for modeling real-time embedded systems, capturing hardware (processors, memory, buses) and operating system primitives (processes, threads, subprograms), supporting analyses such as scheduling and resource usage. Although FACE and AADL address different concerns, they share overlapping information on software structure and timing. To bridge the two domains, SAE has standardized a FACE-to-AADL mapping (SAE International 2019), the relevant part of which is shown in Fig. 2, and is realized through incremental MTs that keep the models synchronized. The mapping is not one-to-one: FACE has no representation for AADL hardware components such as buses, processors and memories, and multiple FACE elements map to the same AADL element. For timing and schedulability analysis, the FACE model must be converted into an AADL model, during which elements with no FACE equivalent (e.g. hardware buses) are not generated and must be added by developers to complete the AADL model.

In this example, the development process involves multiple specialized teams: The FACEDev team develops FACE-compliant software models, denoted as A. This AADL model

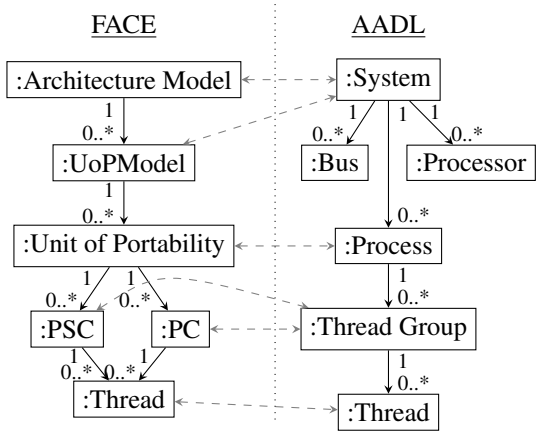


Figure 2 FACE to AADL mapping

is used by AADLDev team as a part of two deployment models, C_1 and C_2, on which the software executes. An integration team maintains the intermediate model B, which requires knowledge of both FACE and AADL, and constructs an AADL representation of A completed with buses and memory specification. Each team owns and maintains its respective models, with restricted modification rights outside its domain. Consequently, a change made in one model (e.g., in A) may need to be selectively propagated to other models while respecting ownership boundaries and access constraints.

We take a look at the following scenarios that can occur during such collaboration to highlight the need for controlling change propagation.

Type-Level Role-Based Control The AADLDev team requires that new Systems are only created or deleted by them, while FACEDev is only permitted to update system attributes. As FACE uses unique IDs for distinguishing its elements, a FACE Dev may rename an architecture model without any issues. In AADL the element's name is its identity. Therefore, when the renamed FACE element is synchronized to AADL, the transformation interprets the rename as a deletion of the old system and creation of a new one. Thus, the FACEDev violates the constraint without explicitly creating any elements in FACE. As such, we require role-based restrictions on certain actions on elements of a certain type.

Instance-Level Role-Based Control An update to threads of the FACE model A can lead to an update of the threads in C_1 and C_2 during early development phases, based on the mapping of Figure 2. Later in the development cycle, one thread in C_1 is already implemented in code and thus modification of that thread should be avoided. Thus a policy has to be added to disallow changes to the specific thread. Over the course of development, the different elements will be restricted and unrestricted and thus policies must be easy to modify. This scenario illustrates element-level restrictions that evolve over the development process.

Obfuscation and Review Control Due to IP protection, the model C_2 should not be accessible to FACEDev. As such, if a change to an AADL process is rejected at C_2, the developers at

A have no indication to the cause of the error, and cannot fix the source due to missing information. During development, it is recognized that changes to one specific process always require a review and further modification by the integrator team before being accepted in C_2. Propagation of changes to such components should be stopped at B so that further modification can be applied by team members having access rights to the obfuscated information. This scenario highlights using change control to manage changes at the site where the change can be corrected as opposed to where the change originated or where the fault was caused.

4. Problem Statement

We have established that in a multi-stakeholder federated MoM setting, unconditional change propagation is undesired and thus requires manual mitigation. This is further compounded by obfuscation and restrictions where the impact of changes are not visible to the creator of a change. As operations on a target model do not correspond to the operations on a source model, it is challenging to recognize ahead-of-time if an operation on the source model will cause an undesired operation on the target model.

From these considerations, we formulate our research questions (RQ):

RQ1 How can we specify policies to define what changes are acceptable in a multi-stakeholder and federated MoM setting?

RQ2 How do we execute the policies to prevent unacceptable changes from being propagated to a model?

RQ3 How does the approach to be developed to answer RQ1 and RQ2 can generalize to different transformations, and modeling languages of different technological spaces?

In order to better answer our RQs, we construct a set of requirements that a suitable solution must satisfy. In RQ1 and RQ2, we cover defining acceptability and propagation, i.e. a way to describe what triggers a policy and what is done when it is triggered. This gives rise to requirements 1, 2, and 3, which specify requirements for constraint description, propagation, and handling failures in propagation. For RQ3, generalization is split into two parts, Requirement 4 deals with an utility aspect of incorporating existing modeling formalisms by requiring non-intrusiveness, while Requirement 5 requires correctness in a non-intrusive situation.

Requirement 1 A policy shall be able to define changes in terms of types of model elements, model elements themselves, and the relation of elements with others.

Requirement 2 Given a change, a policy shall decide on propagation of the change - accept a change, reject a change, or to request human intervention.

Requirement 3 A policy must be able to handle if a change propagation is rejected downstream, i.e. it must be able to control the rejection.

Requirement 4 Implementing such policies must not remove benefits of federation, i.e. models should not need modification, and MTs should be treated as black-boxes.

Requirement 5 Propagation must never lead to a state where further changes will cause MTs to behave incorrectly.

We will thus use the complete or partial satisfaction of these requirements in order to answer the RQ.

5. The Senate Approach

In this Section we describe the main features of Senate. Section 5.1 introduces change policies and how they are interpreted during change propagation. In Section 5.2, we introduce the Domain-Specific Language (DSL) that is used to specify change policies, and describe the inputs and outputs of a policy. Finally, we describe the interpreter for the DSL in Section 5.3.

5.1. Change Policies

With Senate, systems engineers and domain engineers, referred to as policy makers, define policies that control how changes are allowed to propagate through a model. A policy applies to a specific model, independent of the source of a change, or a target of a change. Each model, together with all the associated policies and metadata forms a policy node. A policy node can choose to either accept a change and propagate it, accept but stop propagation pending review, reject the change, or accept the change with a fallback if the change is rejected downstream. Each policy node logs the change and the decision, providing traces for the user to use during integration.

Thus, the policies when evaluated over their models may return the following decisions:

Permit Allow the delta to propagate through the node.

Halt Apply the delta to the current node but do not propagate further.

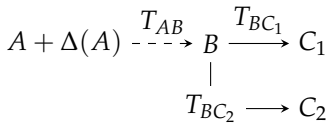
Revert Do not apply the delta to the current node and consider it as a failure.

Try `d1` **Else** `d2` Return the decision `d1` and if the decision fails, return the decision `d2`.

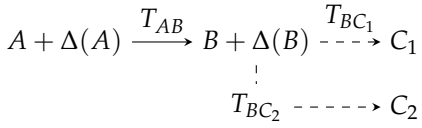
Permit, Halt and Revert decisions provide control over the propagation of the change itself, while Try provides control over the reverts.

Acceptable Change: Permit The set of nodes where a given change will propagate, and the MTs that carry out this propagation form a tree that we call a Transformation Chain (TC). The propagation of a change through a TC in absence of Senate occurs as shown in the Fig. 3. Here, a change is applied to A (Fig. 3a), then propagated to B (Fig. 3b). The propagation to C_1 and C_2 is unordered thus both Fig. 3c or Fig. 3d are possible intermediate states. Finally, the change propagates to all nodes (Fig. 3e).

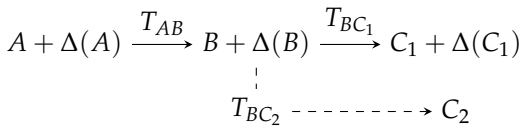
With Senate, propagation through the TC is described in Algorithm 1. Starting from the root of the TC, the propagation



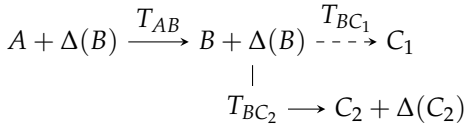
(a) Change at A



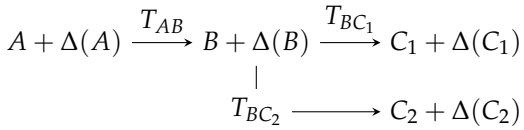
(b) Propagate from A to B



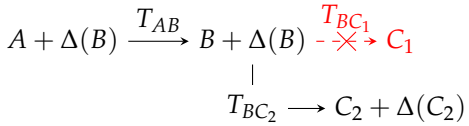
(c) Propagate from B to C_1



(d) Propagate from B to C_2



(e) Propagation complete



(f) Failure at C_1; identical to (3d)

Figure 3 Complete Change Propagation

takes place in a depth-first manner. Changes are applied to the model at the node, and the policy is evaluated on the modified model.

The policy returns a decision based on which the further propagation may change. The propagation in the case where all nodes accept the change corresponds to the propagation as defined by the MTs without Senate. The lack of a policy or the lack of a decision returned from a policy are both considered as tacit permits to maintain the default behavior of existing MTs.

Unacceptable Change: Revert In cases where a change is unacceptable, the policy has multiple options. The first, is to deny the change by returning a *revert* decision. For example, in the event where the change created a new system as discussed in Sec. 3, C_1 receives a change to 'create a system'. The policy at C_1 returns a 'revert' decision, and the change to the model node is undone. This leaves the TC in a state such that the MT between the rejecting node and its predecessor was not executed as shown in Fig. 3f. Note that the states of TC in Fig. 3d and Fig. 3f are identical.

Algorithm 1 Execute TC

Require: *delta* The delta incoming to the given node

```
1:  procedure PROPAGATETO(node, trace)
2:      APPLYDELTA(node)
3:      trace ← push(trace, node)
4:      decision ← EVALUATEPOLICY(node)
5:      return HANDLEDECISION(node, decision, trace)
6:  procedure HANDLEDECISION(node, decision, trace)
7:      switch decision do
8:          case PERMIT
9:              for all child ∈ children(node) do
10:                 finalResult ← PROPAGATETO(child)
11:                 trace ← push(trace, child)
12:                 if finalResult = FAILURE then
13:                     REVERTTRACEUNTIL(trace, node)
14:                     REVERTTOP(trace)
15:                     return FAILURE
16:             return SUCCESS
17:         case REVERT
18:             REVERTTOP(trace)
19:             return FAILURE
20:         case TRYPERMIT
21:             for all child ∈ children(node) do
22:                 finalResult ← PROPAGATETO(child, trace)
23:                 if finalResult = FAILURE ∧ hasElse(node) then
24:                     REVERTTRACEUNTIL(trace, node)
25:                     newDec ← EVALUATEELSE(node)
26:                     return HANDLEDECISION(node, newDec)
27:         case HALT
28:             return SUCCESS
```

The default behavior in case of a revert is to revert to the origin of the change. Thus each change is undone until only the initial change remains i.e. revert the TC to the state in Fig. 3a. Control of the behavior during the revert process is described in the Error Handling: Try-Else paragraph of Section 5.1.

Await Modifications: Halt Heterogeneous models do not have complete overlaps and thus, a change in a model may propagate to other models but leave the elements incomplete. For example, as types such as buses do not exist in FACE, addition of components that require buses to connect them to others will need human intervention to either assign an existing bus or create a new one. In the event an incomplete connection is created at B, the propagation must stop and the bus must be appropriately defined. In such cases, the policy at B should return a *Halt* decision. This will stop the propagation in the state shown in Fig. 3b, and will require human intervention.

Error Handling: Try-Else An all or nothing propagation is useful when an error is solvable at the source. However, it does not suffice in the case where the original user may not have the required context, expertise or access rights to the failure. At times, errors are also expected and need to be handled temporarily to avoid interruptions in the development process.

The Try-Else construct is introduced specifically to deal with propagation failures. For *try* `d1` *else* `d2`, the TC is expected to either end as if `d1` was successful, or as if `d2` was used in the first place. In case there is no `d2` provided, the given successor node is ignored and the propagation continues. For example, in the scenario of Sec. 3, the FACEDev team at A does not have access to the node at C_2, and lacks the required context to correct their change. Meanwhile, the change is acceptable at

C_1. In such case, instead of requiring each change to be fixed individually, the policy at B can return *try permit*. Thus, the offending change will be propagated to A, B, and C_1, but not propagated to C_2, irrespective of the relative ordering between C_1 and C_2.

Accumulated failures can be resolved later during integration. While this does not improve the frequency of integration for the offending change, traces allow users to identify the offending change and its source, and non-offending changes continue propagating.

If B requires review on downstream failures, the decision *try permit else halt* is used. When the permit fails, the change is halted at B, propagating to neither C_1 nor C_2 (Fig. 3b).

In order to propagate deltas correctly, an incremental MT requires that the source and target models are consistent before a change. If the target of an MT is reverted without reverting the source, the MT will simply recognize the differences in the source as the delta. However, if the source model is reverted without the target being reverted, the behavior of an MT is undefined and can cause faults. In order to ensure propagation ends in a state which allows further well-behaved changes, we must always ensure that a change is never applied to a node if it has not been applied to all of its previous nodes.

5.2. Policy Specification

The change policy DSL allows policy makers to specify fine-grained conditions on the operation, changed element, the role of the user making the change, and patterns. To allow instance level constraints without modification to the underlying model, we record the access rights on an instance in markers, which are stored in a marker model. In order to support identifying the structural relations between elements, we add a pattern matching language that can identify such patterns and make their changes available to a policy.

5.2.1. Policy DSL

The EBNF grammar of the change policy DSL is provided in Appendix A. A policy file declares the roles that are used within the file, the types of model elements that are referenced, and multiple policies that apply to the associated model. If multiple policies return decisions, the policies at the top are given precedence so the order of the policy declaration matches the order of evaluation. Each policy has three inputs: The set of deltas, the set of markers, and the set of patterns changed that are received from the query language.

A simple example of a policy is shown in Listing 1, which registers the role *AADLDev*, imports the type *SystemInstance*, and finds all changes to SystemInstances in the model at line 7. Finally, it reverts any change that contains a Create or Delete action by anyone who is not an AADLDev.

```
1  role AADLDev
2
3  import org.osate.aadl2.instance.SystemInstance as
       SysInst
4
5  policy DenySystemChange {
6    let {
7      instances = $input.where[.type = SysInst]
8    } in
9    postcondition {
10     on (Create, Delete) by all except AADLDev
11     over any of instances
12   }
13   then revert
14 }
```

Listing 1 A policy to deny all changes to System Instances except by AADLDev.

5.2.2. Marker Model

The per-object per-role access information is encoded as a set of *markers*. Each marker references a set of elements from the model, a set of operations (Create, Update, Delete), and a set of roles from the policy, as shown in Figure 4 The policy maker can use the $marker keyword in a policy to access this set of markers, which can be used to filter out elements. Markers are named, and thus can also be individually filtered from the model.

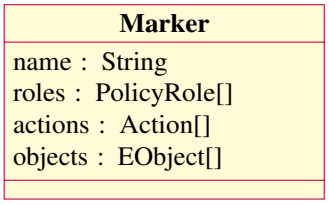


Figure 4 Marker class diagram

```
1  policy DenyModificationExceptTaggedProcesses {
2    let {
3      // Find changes that affect instances of type
         Process
4      processes = $input.where[.type = Process]
5      // Find all markers that relate to the FACEDev
         role.
6      faceDevRelated = $markers.where[.role = FACEDev]
7      // Choose only the markers that correspond to
         Update action
8      canUpdate = faceDevRelated.where[.action =
         Update]
9      // Choose only the markers that correspond to
         Delete Action
10     canDelete = faceDevRelated.where[.action =
         Delete]
11     // Markers for instances FACEDev can update or
         delete
12     canModify = union(canUpdate, canDelete)
13     // Remove the processes that FACEDev is allowed
         to Modify
14     cannotModify = subtract(processes, canModify)
15   } in
16   postcondition { // precondition is reserved for
       future use
17     // Trigger if Update or Delete by FACEDev is
         found in the set
18     on (Update, Delete) by FACEDev
19     over any of cannotModify
20   }
21   // Return Revert if triggered.
22   then revert
23 }
```

Listing 2 Permit FACEDev to only update the marked elements.

Listing 2 shows the DenyModificationExceptTaggedProcesses policy, which combines markers and filters to revert any Update or Delete by FACEDev on Process instances that are not explicitly marked as modifiable. The subtract operation yields only the unmarked instances, on which the policy is then triggered.

5.2.3. Pattern Matching In order to support recognition of structural changes in Senate, we use the VIATRA Query Language (VQL)to define the graph patterns to be matched. The VQL query engine supports an incremental pattern matcher and can identify if a new occurrence of a pattern was found (Create), any element in an existing occurrence was changed (Update), or a previously matched occurrance disappeared (Delete). For example, Listing 3 shows how to describe a pattern query for recognizing AADL subsystems i.e. a SystemInstance that is a subcomponent of another system.

```
pattern subsystem(sys: SystemInstance, sub:
    SystemInstance) {
    SystemInstance.subcomponents(sys, sub);
}
```

Listing 3 Finding subsystems.

Once a pattern is registered as a VQL query, the policy language in Senate can find changes of this pattern by name using `$patterns.where[.name = "subsystem"]`, which returns the set of all the changes in the matched patterns.

5.3. Policy Interpreter

After having presented how policies are specified, we explain the interpretation of policies to control change propagation.

When a change is propagated to a model, the node processes the change as a sequence of edits to the elements and changes in patterns that are declared as input to the policy. This creates two sequences of changes: element changes and pattern changes. In the policy specification, the set of individual element changes is provided through the `$input` variable, and the set of pattern changes as `$pattern` variable, which are a part of the *primaryInput*s shown in the Senate Grammar of Listing A.

Each change is recorded in a *change object* with the following accessible fields:

type Type(s) of the elements that are changed.

action The operation of this change such as Create, Update, or Delete.

role The role of the user that created the original change.

name The name of the pattern that is matched, which is invalid in case of single elements.

Once the inputs are available, the policy file associated with the node is interpreted. The file can contain any number of policies, but each policy is executed in sequence.

In order to select the specific changes of interest, the language provides filtering constructs to progressively narrow the changes of interest. Such is the case of the 'where' clause, which allows the policy maker to filter the changes based on the fields within the above change object. If the field contains more than one value, the filter is considered satisfied if any one of the types matches the filter.

The markers associated with the model are accessed through the `$marker` variable. This provides a set of marker objects shown in Fig. 4, with only action and role as accessible fields, and the 'where' clause then applies to markers in the same way as to the changes.

Since the filters do not contain negation, branching, or logic operations, we need alternate methods to create more complex sets of changes. This is accomplished through set operations such as union, intersection and subtraction. All the set operations apply to sets of changes or sets of markers. However, in order to support filtering markers out of a set, subtract operation is overloaded to act as a per-object filter when subtracting marked objects from the set of changes. Thus, subtracting a set of markers from a set of changes returns a set of changes where only the changes affecting elements that are not in the marker model are returned. The inline comments in Listing 2 explain each of the important lines in a policy, which can be used as a reference.

In order to simplify writing more complex filtering, the language provides a binding block denoted by `let`. The bindings are also interpreted from top to bottom, but are immutable and have no state change associated. The condition statement provides a final filtering which decides whether a policy is triggered. If the condition does not trigger, the interpreter moves to the next policy in the file. Otherwise, the decision and fallbacks in case of 'try' are returned by the interpreter.

Policies have an implicit execution priority based on the order of declaration, with top priority assigned to the policy at the top of a file. This allows multiple policies to be composed such that a more constrained policy is provided higher priority. For example, a policy explicitly permitting changes to a specific element when placed before a policy that reverts changes to all elements of a specific type, has the same effect as adding an exclusion filter to the type-level policy. Senate always returns the decision of the applicable policy which has the highest priority.

6. Evaluation

We evaluate Senate's language construct and semantics against the requirements stated in Section 4, and show that our prototype implements these on a specific technological space. We then outline the conditions in which this can be generalized, as well as the future work required for other domains. This is followed by a proof that the propagation semantics of Senate are well-behaved. We finally present tests on our prototype and evaluate the scalability of our approach.

The Senate prototype[1] is implemented using the Eclipse Modeling Framework (EMF). The Policy DSL is implemented in Xtext (Eysholdt & Behrens 2010) with a Java based interpreter for policy execution, while reverts are implemented using EMF Transactions (*Eclipse EMF Transaction* n.d.). The model change observers are implemented using EMF-IncQuery (Ujhelyi et al. 2015) to query for single element and pattern changes.

6.1. Requirement Satisfaction

We evaluate how Senate satisfies the requirements specified in Section 4.

Requirement 1 is satisfied by using markers with the model element and pattern changes to filter specific set of elements of interest to the policy. The markers allow flexible description of

[1] Available at https://github.com/bhobe/senate

criteria to find specific elements in the model. Since markers are only triplets without an enforced semantic, they can be used to describe inclusion or exclusion criteria. Thus the policy makers have access to all the relevant information about the changes and the elements involved.

Requirement 2 is satisfied by the decision primitives, with some tradeoffs. *permit* and *revert* are normal behaviors for accept-and-propagate and reject-as-failure respectively. *halt* applies the change but stops further propagation, but there is no such counterpart where changes are not applied to the given node, i.e. where propagation is stopped before the given node. A permit result is defined as something that will only succeed if all the successors succeed. Here, success is defined as applying the delta to a model. As such, stopping propagation before the node violates this definition and thus, we choose to limit the primitive to apply but do not propagate.

Requirement 3 is satisfied by the *try-`d1`-else-`d2`* construct, which allows a node to control its failure based on results of the next nodes. The decisions of a node, and the failure handling are aligned to a node making decisions about acceptability of the change as opposed to a general control flow. This also has the effect of making the policies independent of changes to a federation. We thus trade the simplicity of expressing the constraints with the flexibility to provide control flow.

Satisfaction of these three requirements also allows us to answer RQ1, since the Senate language provides the necessary capability to find relevant changes to provide decisions that can correctly control predecessor-to-successor propagation.

Requirement 4 is satisfied by design, as Senate uses observers which identify the changes to the model. Neither the DSL nor the algorithm require modifications of the modeling languages of the federated models and the MTs in use. The policy interpreter only reads inputs from the observers and outputs decisions to demand synchronization or reverts from the executor such as a MoM framework. As such, any MoM framework can produce or forward a set of changes that Senate will be notified of, and propagate the change according to the requested synchronization specified in MoM.

Since Senate applies the policies to deltas, it requires that unchanged elements do not produce deltas. The prototype INC-query based observer identifies writes irrespective of the content, and thus identifies deltas that do not produce changes. As such, the prototype requires the MT to be *hippocratic* i.e. if an element of a source model is currently synchronized with its corresponding element(s) in the target model, then executing the MT should not modify the target model.

Generalization The prototype demonstrates policy application on tree-like propagation patterns, on EMF based models, using hippocratic incremental transformations. However, the approach behind the implementation generalizes to a wider domain.

An attributed typed graph (ATG) (Ehrig et al. 2004), is a formalism that represents a variety of models, including all EMF-based models. Senate's approach is based on finding changes applied to patterns within a graph, by means of a model change observer that can identify such changes. While the implementation itself demonstrates EMF based models, the policy evaluation will correctly operate if the observer is replaced with any other ATG based models.

Thus, with respect to RQ3, the approach of Senate generalizes over ATG-conforming models such as EMF, Unified Modeling Language, Business Process Modeling Language, Petrinets, Automata, etc. and all hippocratic transformations.

6.2. Proof of Correctness

In order to ensure that the change propagation never ends in a state where further changes are invalid, we inductively show that any propagation, with any policy, always ensures that a child node does not receive a change or is not left changed if its parent is unchanged. For the purposes of notation, we use $M \rightarrow N$ to show a synchronized mapping from M to N and $M + \Delta_M \dashrightarrow N$ as a yet unsynchronized mapping.

A model transformation T is defined as a graph rewrite, where an unsynchronized mapping is synchronized by editing the target model, as shown in statement (1)

$$(M + \Delta_M \dashrightarrow N) \overset{T}{\hookrightarrow} (M + \Delta_M \rightarrow N + \Delta_N) \tag{1}$$

We assume that the transformation T is hippocratic which can be formalized as statement (2)

$$(M \rightarrow N) \overset{T}{\hookrightarrow} (M \rightarrow N) \tag{2}$$

Let S_Δ be the set of all models that have been modified during the given propagation. Thus, we can represent the aforementioned invalid state as a state where M is not modified, and N is modified, which is formalized in statement (3)

$$\exists N \in \mathit{successor}(M), M \notin S_\Delta \wedge N \in S_\Delta \tag{3}$$

As such, in order to allow Senate to generalize to any MT, propagation must *never* end in the state (3), assuming that the models are reverted correctly.

We make three assertions from Algorithm 1: (i) Evaluation occurs through a depth-first pre-order traversal, (ii) The final result of a node only depends on the decision of the policies of that node and the final results of its successors, and (iii) All revert operations of a node are completed before the node returns the final results.

We consider a TC, where a model M has k successors $N_1, N_2, \ldots, N_k$ as shown in Figure 5. We start with showing by induction that for the given TC, S_Δ does not satisfy statement 3.

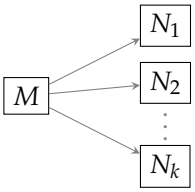


Figure 5 Successor relation between M and $N_{[1,k]}$

We denote the decision of a model M as D_M, and final result of the node of model M as R_M. In table 1 we show how revert maps to failure, while the other decisions map to success.

D_M	R_M	S_Δ
revert	failure	$\emptyset$
permit	success	{M}
halt	success	{M}
try permit	success	{M}

Table 1 Results of a node for $k = 0$

D_M	R_N	$R_{M/N}$	S_Δ
revert	×	failure	$\emptyset$
permit	failure	failure	$\emptyset$
halt	×	success	{M}
try permit	failure	success	{M}
try permit	success	success	{M, N}
permit	success	success	{M, N}

Table 2 Results of a node for $k = 1$

Table 2 illustrates the final result of M given result of N ($R_{M/N}$), based on each possible decision D_M, and the possible result from the successor N (R_N). We also use this to construct the set of modified models S_Δ.

In all cases, S_Δ does not satisfy statement 3. We also note lemma 4 and 5.

$$R_{M/N} = success \implies M \in S_\Delta \tag{4}$$

$$R_{M/N} = failure \implies S_\Delta = \emptyset \tag{5}$$

D_M	$R_{M/N_{[1,k-1]}}$	R_{N_k}	$R_{M/N_{[1,k]}}$	$S_\Delta^{(k)}$
revert	×	×	failure	$\emptyset$
permit	failure	×	failure	$\emptyset$
permit	success∗	failure	failure	$\emptyset$
halt	×	×	success	{M}
try permit	success	failure	success	$S_\Delta^{(k-1)}$
try permit	success	success	success	$S_\Delta^{(k-1)} \cup \{N_k\}$
permit	success	success	success	$S_\Delta^{(k-1)} \cup \{N_k\}$

Table 3 Propagation where k > 1

We continue induction for $k > 1$, where we look at how result (R_k) of k^{th} successor modifies the final result given the $k-1$ previous successors. We use $S_\Delta^{(k-1)}$ to denote the set of changed models for $k-1$. Results of successor nodes that are never evaluated, and thus irrelevant to the computation of the final result of M are marked with ×. We use ∗ alongside the results to mark that any additions to the set S_Δ in the previous are reverted, and thus removed from the set.

From lemma 4 and table 3,

$$\forall k \geq 1, R_{M/N_{[1,k]}} = success \implies M \in S_\Delta \tag{6}$$

And in table 3, we note that:

$$\forall k \geq 1, R_{M/N_{[1,k]}} = failure \implies M \notin S_\Delta \wedge \bigwedge_{i=1}^{i=k} N_i \notin S_\Delta \tag{7}$$

Thus, if the final result is success, then M is modified, while if the final result is a failure, no successor is modified. Thus, from 6 and 7, we can show that for all possible cases, statement 3 is unsatisfied.

From our initial assertions (ii) and (iii), final result of the node only depends on the decision and the next nodes, and all the revert operations complete before returning to the parent node. Thus, for all possible decisions in Senate, the change propagation never leaves the TC in an inconsistent state. Thus we can answer RQ2 as the approach is correct and will never result in propagation of unexpected changes

6.3. Test Suite Description

We construct three test suites that cover different aspects of technical correctness. First, a basic test suite which contains a small set of hand written tests mapping to the collaboration scenarios described in Section 3. Second, a large model test suite which exhaustively tests different policies and their combinations on a given chain of models. Third, a multiple change test suite which tests for sequential changes across different combinations of policies.

The basic test suite contains test cases to first verify the feasibility and correctness of the approach on simple, easy to understand models, and collaboration scenarios. We use a combinations of policies to test different decisions at a model and compare the final state with the expected propagation calculated as per the semantics in Section 5.1.

We use the Basic Avionics Lightweight Source Archetype (BALSA) proposed by the OpenGroup as an example of a FACE software model. A VIATRA incremental transformation is used to keep the FACE model (A) (Figure 1) synchronized with the AADL intermediate model (B). This model is then synchronized with the two other AADL models (C_1, C_2).

During testing, we only modify processes, as there are multiple instances of processes in the BALSA AADL model, each of which has both parent and child elements. This allows the flexibility of different scenarios while having to only use a single type in the policy. We thus construct the following policies: RevertProcesses, RevertUnmarkedProcesses, HaltProcesses, HaltUnmarkedProcesses, TryPermitProcesses, TryPermitUnmarkedProcesses.

The names map to the intent as follows:

Revert/Halt/TryPermit The decision the policy sends.

Unmarked The policy only triggers if the process is not marked 'Update' in the marker model.

We then make changes to two processes 'ATC' and 'ADSB' (names irrelevant). The associated marker model contains a

marker which provides 'Update' access for 'ATC'. Thus, modifying 'ATC' will not trigger any policies of unmarked type, while modifying 'ADSB' will trigger any of the policies.

Each test in the test suite contains the above model synchronization setup, with each test applying a different combination of policies at B, C_1 and C_2. Each test modifies either 'ATC', 'ADSB', 'ATC' then 'ADSB' or 'ADSB' then 'ATC'. Starting from a synchronized state, each test applies a change to the FACE model, lets Senate evaluate the policies and drive the executor, and then compares the resulting state against the expected state using EMFCompare utilities (Toulmé 2006); subsequent changes follow the same cycle.

The large model test suite contains synthetic examples for larger models with 256 processes each. This is used to test the scalability of policies over larger models and larger changes. This test suite contains a set of identical AADL instance models with a simple MTs that keep the models identical. As these MTs are simple 1:1 conversions of deltas, they minimize the impact of the variance in MT execution on measured performance. The MTs are in the form of a sequence $A \rightarrow B \rightarrow C \rightarrow \cdots \rightarrow H$, with automatically assigned policies that accept, reject, halt, try-permit changes to an arbitrarily chosen element common in all models as they are copies.

The multiple modification test suite stress-tests Senate with sequential changes on the same models, prioritizing modifications on unmodified elements before revisiting already-modified ones. Thus, scaling from 1 to 256 modifications measures per-element overhead, while beyond 256 the added changes only revisit existing elements. The suite is exhaustive but prunes redundant cases: e.g., once a change is halted at B, the policies at C_1 and C_2 are irrelevant, so a single test suffices for that branch.

6.4. Comparison with current practice

In current practice, for the running example shown in Section 3, merges are handled during a dedicated synchronization step. Multiple changes are applied on a model, for example, model A in the running example, over a period of time before the synchronization. The changed A is then synchronized to B, C_1, and C_2, then reviewed manually against guidelines by the owners of each model. These guidelines are plain text and thus depend on the reviewer correctly identifying issues. When the reviewers find any guideline violations, the reviewers need to identify the change that caused the violation, and trace it to the source. The reviewers must then find the team or developer who can resolve the change, who may be in a different organization. Since multiple models may be modified over the period, B must be synchronized to all connected models, which makes tracing further complicated.

Senate's policies are explicit and incremental, providing automatic guideline enforcement at the time of applying changes with logging that allow the integrators to find the source of changes in the event Senate does not find an error. As a models is edited, the changes are tracked and propagated through the federation. Propagation of a violating change is caught before proceeding with the next changes. Multiple changes if not propagated are still held individually and applied in sequence, providing the same benefit.

6.5. Results

All test cases meet the expected behavior of Senate as manually computed based on the semantics presented in Section 5.1.

In-tool instrumentation allows profiling of the prototype and provides insight into the observed performance. The time logging is divided into regions such as PolicyInterpret, Revert, and Transformation. We also log the time taken to load policies and to run the test etc. However, as policies should stay loaded in a real world scenario, we exclude these timings. A plot of Senate's contribution to the total time taken to execute the synchronization is shown as a stack plot in Fig. 6[2].The hatched section shows the baseline MT time without Senate. The stacked elements above show Senate's overhead.

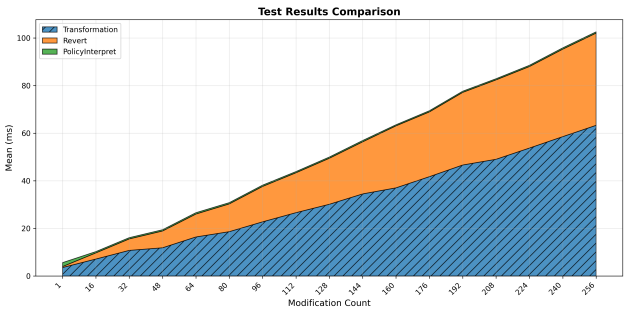
We observe that the execution of the policy itself takes a small portion of the time, as a majority of the time is spent on the revert and transformations. The revert execution time correlates to the number of unique elements that are changed, as opposed to the number of changes, as shown by the plateau in time taken after 256 changes in Figure 6b. As such, we note that the impact of senate on performance scales linearly with the number of unique elements changed, but has relatively insignificant impact on the number of changes to the same element. For most real scenarios, the number of changes exceeds the number of unique elements changed, and as such, while the growth will be still linear, the expected overhead will be less than the overhead noted in the graph. This overhead will be further reduced as the complexity of the transformation itself increases.

7. Limitations

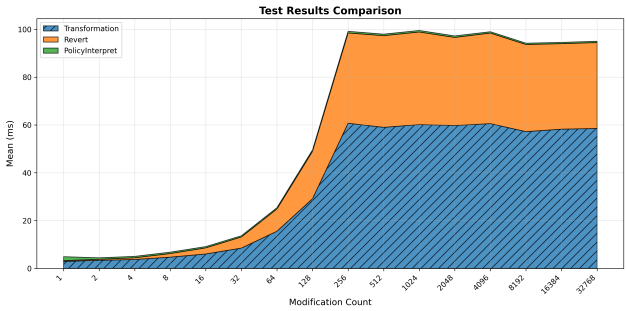
Senate performance scales linearly with respect to the number of changes, and is not affected by the size of the model. While Senate can process large deltas, each delta is rejected or accepted in its whole. This is optimal behavior in scenarios with small deltas being processed, as the failures are easily and quickly pointed out. By default, Senate is supposed to be observing changes in all models, thus producing a delta for each change. However, if the users make large changes to the models without Senate in the loop, this can produce a large delta. If any change in the delta violates the policy, all the changes in the delta will be rejected. Such large deltas, though discouraged in current practice, are inevitable when unmanaged external models are updated in a federation. While Senate does not improve on this particular use-case, we focus on the benefit of Senate to the other scenarios where the models are being managed in the federation.

The approach currently assumes that changes are propagated as trees, which is a common scenario in federated MoM. However, in a scenario where a model receives deltas from multiple models, changes propagation forms a Directed Acyclic Graph (DAG). However, in a DAG, as different changes propagate or are halted, dissimilar changes can be propagated to a model from its sources, which needs additional handling. Similarly,

[2] Complete data available at https://github.com/bhobe/senate



(a) 16 to 256 changes, 256 unique elements



(b) 2^0 to 2^{16} changes, 256 unique elements

Figure 6 Time (ms) taken to propagate changes

revert for a DAG requires a separate failure propagation calculation to compute the nodes that need to be reverted as opposed to reversing in the order of application. Thus, we leave this as an extension for future work. Similarly, generalizing to general graphs with cycles requires fixed point computation which adds further complexity in policy description as each policy may be interpreted more than one time as a change propagates in a cycle.

The current approach and implementation requires a hippocratic transformation. A non-hippocratic transformation can replace an element with an identical copy, which is recognized by the EMF-IncQuery engine as a change. This can cause false-positive applications of policies. Using an observer that does not recognize such replacements as changes should produce the same observed changes as a hippocratic transformation, but will impact performance negatively due to additional comparisons. However, this requires additional testing.

We currently only support and test constraints on structural and attribute changes in model. This does not include constraints on the permitted values of a specific attribute, which a policy may want to check before or after change application. However, this is a possible extension to the policy DSL that does not require any changes at the semantic level for the evaluation or propagation.

8. Related Work

We look at the state-of-the-art works that are related to change management in context of MBSE. There are no current works that provide change management in MoM. However, we look at the state-of-the-art for access control in MBSE and MoM, as access control involves a form of change management. While Senate does not provide read access control, it features write access control in a single model as well as in a MoM context. In Section 8.1, we look at works related to access control. We also look at the various methods related to specification of access-control policies in Section 8.2 to note the other possible approaches in specifications, including ideas that can be integrated into Senate for future improvements. Finally, we look at approaches of ad-hoc change management in MoM in Section 8.4.

8.1. Access Control in MBSE

RBAC has been extensively studied in the context of knowledge stores and databases. In the context of MBSE, the available metamodels and semantics of the process allow for more refined Access Control methods.

Early work in access control in MBSE focused on generic model repositories. The EMFStore (Koegel & Helming 2010) platform provided access control at the level of entire models, similar to file-level access control in other contexts.

Debreceni et al. (Debreceni et al. 2019) propose an approach which provides fine-grained access control on a collaborative platform. This approach is built in the context of a single model, where the complete version is stored on a remote server. The local copy for a user only contains the parts of the model that they have access control to. The changes made by the user are sent to the remote. These can be accepted if correct or rejected if they violate access control rules. While the approach can specify and govern access, it will either accept a change or reject it entirely. This requires a central authority to accept the changes.

Martinez et al. (Martínez et al. 2018) propose an approach to generate views of models based on Object Constraint Language (OCL) (Object Management Group 2014) rules that only show read accessible elements and only allow writes to the write-accessible elements. However, the views of models do not limit write access to other models that are directly synchronized to the current model.

Currently, fine-grained access control is usually not considered in the state-of-the-art MoM tools such as DesignSpace (Demuth et al. 2015), OpenFlexo (Bach et al. 2024), Vitruvius (Klare et al. 2021) and approaches such as Comprehensive Systems (Stünkel et al. 2021), Hierarchical MegaModels (Seibel et al. 2010).

Exelmans et al. (Exelmans et al. 2024) propose an initial approach for secure model versioning which provides a primitive element level read/write access control by requiring the user to have access to each element to retrieve its content. They put forth the idea of synchronization via shared immutable data where the data itself is access controlled, but cannot manage constraints over derived or computed values.

8.2. Formal Access Specification

The need for cyber-security, coupled with the benefits provided by MBSE, has given rise to the field of Model Driven Security (MDS) (Lúcio et al. 2014). MDS studies the use of models to specify and verify security of systems being designed. Thus, there is a large body of work on specification of access control.

The initial contributions by Epstein and Sandhu (Epstein & Sandhu 1999), Jurgens (Jürjens 2001), and Lodderstedt et al. (Lodderstedt et al. 2002) propose using UML notations to specify security requirements for software applications. These approaches are used for design specification, and the specification language is insufficient to describe changes on modification of a model.

Elrakaiby et al. (Elrakaiby et al. 2014) propose Security@Runtime DSL to specify access control policies that are enforced in a Java based application. Martinez et al. (Martínez et al. 2016) show a model driven approach to enforce access control at runtime by translating policies into executable transformations that can convert access requests to decisions. In both approaches, the focus is on preventing incorrect access to information, thus they control access to elements and their attributes, but do not have or need support for pattern recognition.

Constraints related to acceptable attributes in elements as well as the references between elements are often described using OCL (Object Management Group 2014). While the existence of a pattern can be recognized and disallowed, there are no standardized mechanisms to react to events such as creation, update, or deletion that are provided by VQL.

8.3. Impact Analysis

Impact analysis has been studied for over two decades to trace how changes in a system apply to the whole. Netaji et. al. (Nejati et al. 2016) puts forth an approach to map changes in requirements to changes in the design. This method uses natural language processing to grasp the initial change required, and then user input to refine the set. Jagla et. al. (Jagla et al. 2021) uses SysML to model the links between a change and the elements it impacts, in order to estimate the implementation effort. Impact analysis also exists in general systems engineering outside of modeling and software (Clarkson et al. 2004) which model the propagation ahead of time to collect the possible impacts. In all the above works, the goal is to estimate the effort required or to avoid changes propagating to model which has high cost of modification, i.e. provide a specific form of constraint on the propagation, without a focus on actual change applied as they only study the possible impact.

Impact analysis has been studied in MoM in heterogeneous contexts by using model slicing in megamodels (Salay et al. 2016). Megamodels use links that denote the elements that are connected and synchronized. Salay et. al. use these links to transitively mark all elements that are connected to the initial element to find a slice of the megamodel that will be impacted by a given change. This can be used to assess change impact and thus analyze the possible impact of a change. The Senate approach requires element level information to identify violations of element level constraint, and thus cannot directly use such an ahead-of-time approach. However, impact analysis can be used to trace the possible type level policy violations ahead of time, to warn users of possibility of a violation before execution, which would improve the usability of Senate.

8.4. Change Management

The state-of-the-art MoM approaches provide features such as model querying, model synchronization, consistency management, verification and validation as well as describing model constraints (Amrani et al. 2024). However, the model synchronization can only be customized by changing the model synchronizations used between the specific models. Thus, the synchronization can be statically defined to write or to not write to any elements of a particular type in a particular model. This is not role aware, and is not as fine-grained as element level restrictions provided by Senate.

OpenFlexo (Bach et al. 2024) is a federated MoM approach that uses the Federation Modeling Language (FML) to describe the relation between the different model elements as well as their synchronization. FML is a turing-complete imperative language that can be used to customize behavior of each individual element level relation at runtime. While this allows control over change propagation and synchronization, it is not inherently role aware and does not provide multi-stakeholder constraints, and thus would effectively require an implementation of change management in FML.

9. Conclusion

In this paper we address a gap in MoM research: The lack of control over change application and propagation. We present Senate, a novel framework for controlling change propagation in federated MoM. Senate provides a DSL for describing fine-grained, role-aware change policies; and an evaluator for correct transformation application and reverts for consistent behavior. This allows stakeholders and component owners to control the automated changes applied to their models and reduce the manual review effort. We validate that the policy semantics match the implementation, and show that the approach itself is generalizable to all ATG-conforming models and hippocratic transformations by implementing external plugins.

We also evaluate the performance impact of using Senate within a transformation change, to show that the overhead of Senate in non-failure cases is small, and the overhead in failure cases grows linearly with the size of the changes made, thus providing strong scalability characteristics of smaller incremental changes.

This work provides the basis for change propagation management in federated MoM frameworks. It leaves open many avenues for future development. The next steps include evaluation of performance and scalability of the approach with larger industrial models. This includes performance with respect to the complexity of the delta, and the TC. We will also explore the use of change management methods beyond role-based, to generalizations such as attribute-based access controls as well as capture notions such as source of truths. Further, we can explore generalization of the change propagation to directed-acyclic graphs or directed graphs. Finally, a detailed user study to assess the expressivity and usability of the policy language.

References

Amrani, M., Blouin, D., Heinrich, R., Rensink, A., Vangheluwe, H., & Wortmann, A. (2021). Multi-paradigm modelling for cyber–physical systems: A descriptive framework. *Software and Systems Modeling*, *20*(3), 611–639. doi: 10.1007/s10270-021-00876-z

Amrani, M., Rakshit, M., Goulão, M., Amaral, V., Guérin, S., Martínez, S., … Hallak, Y. (2024). A Survey of Federative Approaches for Model Management in MBSE. In ACM (Ed.), *1st workshop on model management (mom) at models 2024*. Linz (AUSTRIA), Austria. doi: 10.1145/3652620.3688221

Bach, J.-C., Beugnard, A., Champeau, J., Dagnat, F., Guérin, S., & Martínez, S. (2024). 10 years of Model Federation with Openflexo: Challenges and Lessons Learned. In *Proc. models* (pp. 25–36). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3640310.3674084

Barišić, A., Ruchkin, I., Savić, D., Mohamed, M. A., Al-Ali, R., Li, L. W., … Cicchetti, A. (2022). Multi-paradigm modeling for cyber–physical systems: A systematic mapping review. *Journal of Systems and Software*, *183*, 111081. doi: 10.1016/j.jss.2021.111081

Bézivin, J., Jouault, F., Rosenthal, P., & Valduriez, P. (2005). Modeling in the Large and Modeling in the Small. In U. Aßmann, M. Aksit, & A. Rensink (Eds.), *Model Driven Architecture* (pp. 33–46). Berlin, Heidelberg: Springer. doi: 10.1007/11538097_3

Bird, C., Nagappan, N., Murphy, B., Gall, H., & Devanbu, P. (2011). Don't touch my code! examining the effects of ownership on software quality. In *Proc. esec/fse* (pp. 4–14). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2025113.2025119

Caron, F., Blouin, D., Crisafulli, P., & Maxim, C. (2020). Seamless Integration between Real-time Analyses and Systems Engineering with the PST Approach. In *Proc. syscon* (pp. 1–8). doi: 10.1109/SysCon47679.2020.9275907

Cederbladh, J., Cicchetti, A., & Suryadevara, J. (2024). Early Validation and Verification of System Behaviour in Model-based Systems Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.*, *33*(3), 81:1–81:67. doi: 10.1145/3631976

Clarkson, P. J., Simons, C., & Eckert, C. (2004). Predicting change propagation in complex design. *Journal of Mechanical Design*, *126*(5), 788-797. doi: 10.1115/1.1765117

Debreceni, C., Bergmann, G., Ráth, I., & Dániel, V. (2017). Property-Based Locking in Collaborative Modeling. In *Proc. models* (pp. 199–209). doi: 10.1109/MODELS.2017.33

Debreceni, C., Bergmann, G., Ráth, I., & Varró, D. (2019). Enforcing fine-grained access control for secure collaborative modelling using bidirectional transformations. *Software & Systems Modeling*, *18*(3), 1737–1769. doi: 10.1007/s10270-017-0631-8

Demuth, A., Riedl-Ehrenleitner, M., Nöhrer, A., Hehenberger, P., Zeman, K., & Egyed, A. (2015). DesignSpace: An infrastructure for multi-user/multi-tool engineering. In *Proc. acm sac* (pp. 1486–1491). Salamanca Spain: ACM. doi: 10.1145/2695664.2695697

Eclipse EMF Transaction. (n.d.). https://www.eclipse.org/emf-transaction/. ([Accessed 18-02-2026])

Ehrig, H., Prange, U., & Taentzer, G. (2004). Fundamental Theory for Typed Attributed Graph Transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, & G. Rozenberg (Eds.), *Graph Transformations* (pp. 161–177). Berlin, Heidelberg: Springer. doi: 10.1007/978-3-540-30203-2_13

Elrakaiby, Y., Amrani, M., & Le Traon, Y. (2014). Security@Runtime: A Flexible MDE Approach to Enforce Fine-grained Security Policies. In J. Jürjens, F. Piessens, & N. Bielova (Eds.), *Engineering Secure Software and Systems* (pp. 19–34). Cham: Springer International Publishing. doi: 10.1007/978-3-319-04897-0_2

Epstein, P., & Sandhu, R. (1999). Towards a UML based approach to role engineering. In *Proc. acm workshop on role-based access control* (pp. 135–143). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/319171.319184

Exelmans, J., Pietron, J., Raschke, A., & Vangheluwe, H. (2024). A Virtual Global Monorepo of Immutable Linked Data. In *Proc. models* (pp. 1000–1004). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3652620.3688222

Eysholdt, M., & Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In *Proc. oopsla companion* (pp. 307–309).

International Organization for Standardization. (2014). *Iso:14258:2014: Industrial Automation Systems: Concepts and Rules for Enterprise Models* (No. ISO/TC 184/SC 5 14258:1998). Geneva, Switzerland.

Jagla, P., Jacobs, G., Siebrecht, J., Wischmann, S., & Sprehe, J. (2021). Using SysML to Support Impact Analysis on Structural Dynamics Simulation Models. *Procedia CIRP*, *100*, 91–96. doi: 10.1016/j.procir.2021.05.015

Jürjens, J. (2001). Towards Development of Secure Systems Using UMLsec. In H. Hussmann (Ed.), *Fundamental Approaches to Software Engineering* (pp. 187–200). Springer. doi: 10.1007/3-540-45314-8_14

Klare, H., Kramer, M. E., Langhammer, M., Werle, D., Burger, E., & Reussner, R. (2021). Enabling consistency in view-based system development — The Vitruvius approach. *Journal of Systems and Software*, *171*, 110815. doi: 10.1016/j.jss.2020.110815

Koegel, M., & Helming, J. (2010). EMFStore: A model repository for EMF models. In *Proc. icse* (Vol. 2, pp. 307–308). doi: 10.1145/1810295.1810364

Lodderstedt, T., Basin, D., & Doser, J. (2002). SecureUML: A UML-Based Modeling Language for Model-Driven Security. In G. Goos, J. Hartmanis, J. Van Leeuwen, J.-M. Jézéquel, H. Hussmann, & S. Cook (Eds.), *≪UML≫ 2002 — The Unified Modeling Language* (Vol. 2460, pp. 426–441). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/3-540-45800-X_33

Lúcio, L., Zhang, Q., Nguyen, P. H., Amrani, M., Klein, J., Vangheluwe, H., & Traon, Y. L. (2014). Advances in Model-Driven Security. In A. Memon (Ed.), *Advances in computers* (Vol. 93, p. 103-152). Elsevier. doi: https://doi.org/10.1016/

B978-0-12-800162-2.00003-8

Martínez, S., Fouche, A., Gérard, S., & Cabot, J. (2018). Automatic Generation of Security Compliant (Virtual) Model Views. In J. C. Trujillo et al. (Eds.), *Conceptual Modeling* (Vol. 11157, pp. 109–117). Cham: Springer International Publishing. doi: 10.1007/978-3-030-00847-5_10

Martínez, S., García, J., & Cabot, J. (2016). Runtime support for rule-based access-control evaluation through model-transformation. In *Proc. acm sigplan sle* (pp. 57–69). Amsterdam Netherlands: ACM. doi: 10.1145/2997364.2997375

Nejati, S., Sabetzadeh, M., Arora, C., Briand, L. C., & Mandoux, F. (2016). Automated change impact analysis between SysML models of requirements and design. In *Proc. fse* (pp. 242–253). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2950290.2950293

Object Management Group. (2014). *Object Constraint Language (OCL) Specification* (Technical Standard No. formal/14-02-03). Object Management Group.

SAE International. (2019). *Architecture Analysis & Design Language (AADL) Annex f: Aadl Annex for the FACE™ Technical Standard, edition 3.0* (Technical Standard No. AS5506/4). SAE International. doi: 10.4271/AS5506/4

SAE International. (2022). *Architecture Analysis & Design Language (AADL)* (Technical Standard No. AS5506D). SAE International. doi: 10.4271/AS5506D

Salay, R., Kokaly, S., Chechik, M., & Maibaum, T. (2016). Heterogeneous megamodel slicing for model evolution. In T. Mayerhofer, A. Pierantonio, B. Schätz, & D. Tamzalit (Eds.), *Proc. workshop on models and evolution co-located with models 2016* (Vol. 1706, pp. 50–59). CEUR-WS.org.

Seibel, A., Neumann, S., & Giese, H. (2010). Dynamic hierarchical mega models: Comprehensive traceability and its efficient maintenance. *Software & Systems Modeling*, *9*(4), 493–528. doi: 10.1007/s10270-009-0146-z

Stünkel, P., König, H., Lamo, Y., & Rutle, A. (2021). Comprehensive Systems: A formal foundation for Multi-Model Consistency Management. *Formal Aspects of Computing*, *33*(6), 1067–1114. doi: 10.1007/s00165-021-00555-2

The Open Group FACE Consortium. (2025). *Future airborne capability environment (FACE) technical standard* (Technical Standard). The Open Group.

Thongtanunam, P., McIntosh, S., Hassan, A. E., & Iida, H. (2016). Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proc. icse* (pp. 1039–1050). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2884781.2884852

Toulmé, A. (2006). Presentation of emf compare utility.

Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., … Varró, D. (2015). EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, *98*, 80–99. doi: 10.1016/j.scico.2014.01.004

Van Tendeloo, Y. (2018). *A Foundation for Multi-Paradigm Modelling* (Unpublished doctoral dissertation). Universiteit Antwerpen.

Walden, D. D., Roedler, G. J., & Forsberg, K. (2015). INCOSE Systems Engineering Handbook Version 4: Updating the Reference for Practitioners. *INCOSE International Symposium*, *25*(1), 678–686. doi: 10.1002/j.2334-5837.2015.00089.x

Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., & Mei, H. (2007). Towards automatic model synchronization from model transformations. In *Proc. ase* (pp. 164–173). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1321631.1321657

About the authors

Anish Bhobe is a doctoral student at Télécom Paris, and Institut Polytechnique de Paris. His research is focused on model management (MoM) topics, notably model federation, model synchronization and change management. You can contact the author at anish.bhobe@telecom-paris.fr or visit https://abhobe.com.

Dominique Blouin is an associate professor at Télécom Paris. He obtained this Ph.D. in computer science from the University of South-Brittany in 2013. He was a post-doctoral researcher in the system analysis and modeling group of the Hasso Plattner institute. His research interests are model management, multi-paradigm modeling, domain-specific languages, requirements engineering and Cyber-Physical Systems. You can contact the author at dominique.blouin@telecom-paris.fr.

Laurent Pautet is a Professor at Télécom Paris. His research activities focus on design and validation of critical embedded systems (non-functional properties verification, real-time scheduling, real-time kernels). He contributes technically and scientifically to several international free software projects such as GNAT/GCC. He is an editor and a contributor to several books in the area of distributed real-time embedded systems.You can contact the author at laurent.pautet@telecom-paris.fr.

A. Policy DSL Grammar

```
policyModel   = element*;
element       = roleDecl | importDecl | policyDecl;
roleDecl      = "role" role+;
importDecl    = "import" typeName "as" ID;
policyDecl    = "policy" ID "{"
                  letBlock?
                  trigger decision "}";
letBlock      = "let" "{" assignment+ "}" "in";
assignment    = variable "=" expr;
expr          = baseExpr ("." filter)*;
baseExpr      = variable | input | "(" expr ")"
              | setOperation "(" expr "," expr ")";
filter        = "where" "[" "." key "=" value "]";
trigger       = "postcondition" "{" condition "}";
decision      = "then" (termDecision | tryDecision);
termDecision = "revert" | "permit" | "halt";
tryDecision = "try" termDecision ("else" decision)?;
condition     = "on"
                "(" operation ("," operation)* ")"
                ("by" ("all" | roleList)
                 ("except" roleList)?)?
                "over" ("any"|"all") "of" variable;
roleList      = role | "(" role ("," role)+ ")";
operation     = "Create" | "Update" | "Delete";
setOpertion   = "union" | "intersect" | "subtract";
input         = "$input" | "$patterns" | "$markers";
```

Listing 4 EBNF grammar for the Policy DSL