Software & Systems Modeling (2020) 19:231–260
https://doi.org/10.1007/s10270-019-00738-9

REGULAR PAPER

Heterogeneous megamodel management using collection operators

Rick Salay[1] · Sahar Kokaly[2] · Alessio Di Sandro[1] · Nick L. S. Fung[1] · Marsha Chechik[1]

Received: 11 June 2018 / Revised: 11 March 2019 / Accepted: 28 May 2019 / Published online: 22 June 2019

Abstract
Model management techniques help tame the complexity caused by the many models used in large-scale software development; however, these techniques have focused on operators to manipulate individual models rather than entire collections of them. In this work, we begin to address this gap by adapting the widely used *map*, *reduce* and *filter* collection operators for collections of models represented by megamodels. Key parts of this adaptation include the special handling of relationships between models and the use of polymorphism to support heterogeneous model collections. We evaluate the complexity of our operators analytically and demonstrate their applicability on six diverse megamodel management scenarios. We describe our tool support for the approach and evaluate its scalability experimentally as well as its applicability on a practical application from the automotive domain.

Keywords Megamodel · Model management · Heterogeneous

1 Introduction

Large-scale software development often uses heterogeneous collections of related models; however, such collections create accidental complexity that must be managed. The field of Model Management [1] has emerged to address this challenge. Model management focuses on a high-level view in which entire models and their relationships (i.e., mappings between models) can be manipulated using specialized operators to achieve useful outcomes. For example, a model *match* operator [1] finds correspondences between the elements of two models and packages these as a mapping between the models. A *merge* operator [1] can then be used to combine the content of the two models using the correspondence information in the mapping. Model management approaches typically use *megamodels* [3] to represent sets of models and their relationships in this high-level view. For example, a megamodel could be a graphical model that uses nodes to represent models and edges to represent relationships.

Model management has been studied from many perspectives including algebraic properties of operators [4,34], categorical foundations [7], type theory [46], megamodeling languages [13,39] and practical implementations [23,27,34,36]. In these investigations, the focus is on the general manipulation of models rather than specifically on the manipulation of megamodels. Since megamodels are a special kind of model, we expect that general model management operators apply to them equally well. Yet, a megamodel is a hierarchical "model of models" and this special characteristic necessitates unique support in model management. Specifically, we identify the following requirements:

- (R1) Since megamodels represent *collections* (of models and relationships), their manipulation should be like that of other collection types (e.g., lists, graphs, etc.) commonly found in modern programming languages. In particular, three collection operators are widely used: *map*, *filter* and *reduce*.
- (R2) Since megamodels represent *heterogeneous* collections, their manipulation requires a sound approach for

Communicated by Dr Benoit Combemale.

✉ Rick Salay
rsalay@cs.toronto.edu

Sahar Kokaly
kokalys@mcmaster.ca

Alessio Di Sandro
adisandro@cs.toronto.edu

Nick L. S. Fung
nlsfung@cs.toronto.edu

Marsha Chechik
chechik@cs.toronto.edu

[1] University of Toronto, Toronto, Canada

[2] McMaster University, Hamilton, Canada

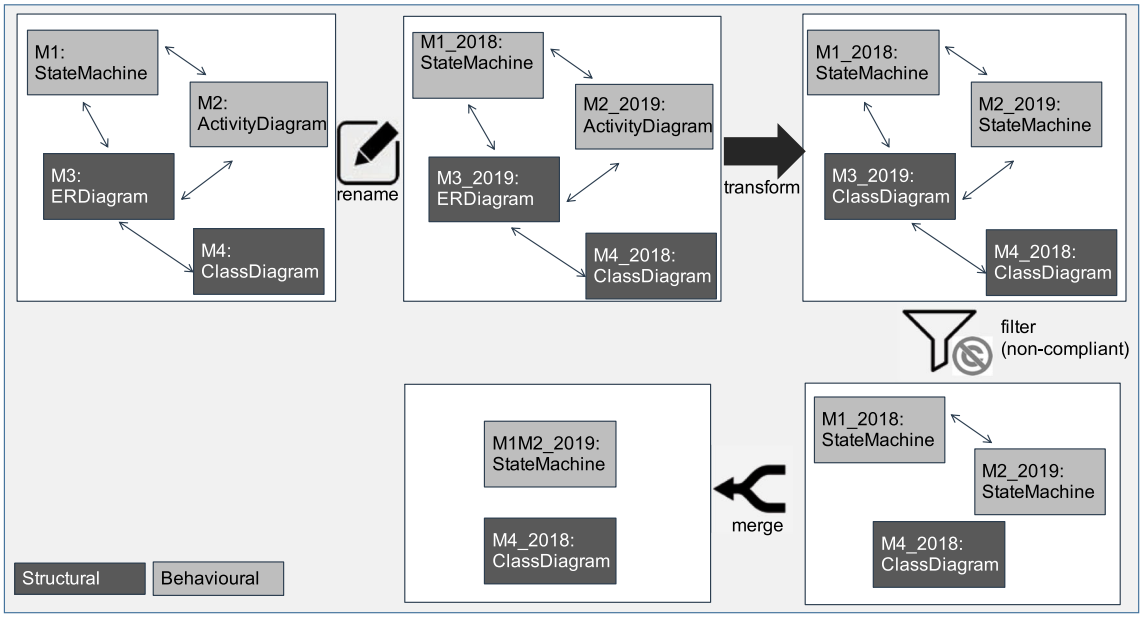

Fig. 1 Motivating example: IT standard change

decoupling megamodel management from the special handling required by the different model and relationship types found within them.

In this paper, we define a strategy for heterogeneous megamodel management that addresses these requirements. To address requirement (R1), we focus on three widely used collections operators: *map* for applying a function to every element of a collection, *reduce* for aggregating elements in a collection and *filter* for extracting a subset of the collection using a property as a selector. While megamodels bear similarity to collections in programming, they also have their unique challenges that limit our ability to apply these techniques without some adaptation. To address requirement (R2), we turn to the programming concept of polymorphism used to address function heterogeneity, and adapt it to transformations over models and relationships. We illustrate this strategy below.

Motivating scenario A company uses a megamodel to track its modeling artifacts (heterogeneous models and relationships between them), as seen in Fig. 1. The company wants to change the naming convention across all of its modeling artifacts by using a renaming operation, which has polymorphic variants for the different model types it is applied to. The renaming adds the year in which the model was created to its name. In addition, the company wants to eliminate the variety of different model types used for the same kind of information by using only UML state machines for state-like behavioral models (e.g., activity diagrams) and UML class diagrams for structural models (e.g., entity-relationship diagrams). Finally, they would like to filter out all the non-compliant models, where compliance is a polymorphic property, and merge the models of the same type to do some further analysis.

A natural way to execute these steps is to (1) use *map* to apply the renaming transformation to all models using the appropriate variant of the renaming operation for each model type, (2) use *map* again to apply the UML transformation to all renamed models using the appropriate transformation operation for each model type based on its nature (state-like behavioral vs. structural), (3) use *filter* to extract the non-compliant models *and the relationships between these*, (4) use *reduce* with a merge transformation to combine all the resulting non-compliant models pairwise, *correctly taking into account the relationships between them*.

Thus, we need collection operators to manipulate entire *graphs* of related models rather than just lists of models. Furthermore, we need to allow invoking *map* and *reduce* with transformations that can accept graphs of models and relationships as input and produce these as output. Finally, we need our collection-based operators to work on heterogeneous sets of models, correctly handling the application of operations or properties based on the model types they are applied to, and ensuring the relationships between models are correctly handled as well.

Contributions This paper makes the following contributions:

1. We formally define versions of *map*, *reduce* and *filter* collection operators adapted for heterogeneous megamodels which treat relationships between models as first class entities:

  - **map**—for applying a transformation to the elements of a megamodel;
  - **reduce**—for aggregating the elements of a megamodel using a transformation; and
  - **filter**—for extracting a subset of elements of a megamodel that satisfy a property.

2. We analyze the complexity of the operators.
3. We demonstrate the approach by using the operators to express several non-trivial megamodel management scenarios.
4. We report on an implementation of the operators and evaluate its scalability experimentally.
5. We apply the approach and tool support to a practical problem from the automotive safety domain.

The rest of this paper is organized as follows: After fixing the terminology in Sect. 2, we define the three collection operators for megamodels in Sect. 3. Section 4 illustrates these on six practical scenarios. Section 5 analyzes the complexity of the operators. Section 6 describes tool support with experiments reported in Sect. 7. We describe a practical application of automotive safety case impact assessment in Sect. 8. We compare our approach with related work in Sect. 9 and conclude in Sect. 10 with a summary of the paper and a discussion of future research directions.

This paper expands our previous results [38] on homogeneous megamodel collection operators in several ways. First, we have added the formal details for supporting polymorphism (see Sect. 2). They are needed for supporting heterogeneity within the collection operators. Second, we have adapted the operator descriptions to address heterogeneity (see Sect. 3). Third, we significantly modified and expanded the set of scenarios we handle to include operators with heterogeneity (see Sect. 4). Fourth, we have developed tool support for handling polymorphism, which we discuss in Sect. 6. Fifth, we report on the results of experiments to evaluate the scalability of the tool support in Sect. 7. Finally, we have added a qualitative evaluation of the tool support by using it to implement a practical problem from the automotive safety domain in Sect. 8.

2 Preliminaries

In this section, we formalize the concept of megamodel and give other necessary definitions.

2.1 Basic types

We begin by defining the concept of "mega-graphs"—*mgraphs*. Informally, a *megamodel* is an mgraph whose nodes refer to artifacts in a repository.

Definition 1 (*mgraph*) An *mgraph* is a structure that is an instance of the metamodel in Fig. 2. Given an mgraph G, we write G_C to denote the set of nodes in node class C. When C is omitted, the node class is `Node`. We use abbreviations `Mod` and `Rel` for node classes `Model` and `Relationship`, respectively. For $n \in G$, we write $n.R$ to denote the set of nodes on the other end of reference R from node n.

In this paper, we limit our focus to megamodels that can refer to artifacts corresponding to the concrete node classes in Fig. 2. A *relationship* is a mapping between two or more models. A *transformation application* is the record of having performed a given transformation on a set of input models and relationships to produce a set of output models and relationships. We make no further assumptions about the way models relationships or transformation applications are represented or what they contain. The "mega" versions of these artifacts: megamodels, megarels and megaApps are defined below.

We assume the existence of a repository.

Definition 2 (*Repository*) A *repository* $\mathscr{R}$ is a store for artifacts that is itself structured as an mgraph of artifacts (i.e., rather than an mgraph of symbols).

Models, relationships and transformation applications are typed by model types, relationship types and transformations, respectively.

We assume that there exists a *type compatibility preorder* $\preceq_{\mathrm{TC}}$ over model and relationship types where $T' \preceq_{\mathrm{TC}} T$ means that an instance of type T' can be used wherever an instance of type T is needed. Figure 3 shows an example type compatibility preorder with specialized model types of class diagram (`CD`) to only allow single inheritance (`SICD`) and providing Java features (`JCD`). `CDrel` is a type of relationship.

Mappings between mgraphs are called *mgraph homomorphisms*.

Definition 3 (*mgraph homomorphism*) Given mgraphs G, G' and a type compatibility preorder $\preceq_{\mathrm{TC}}$, an *mgraph homomorphism* $f : G \to G'$ is a function $f_{\mathtt{Node}} : G_{\mathtt{Node}} \to G'_{\mathtt{Node}}$

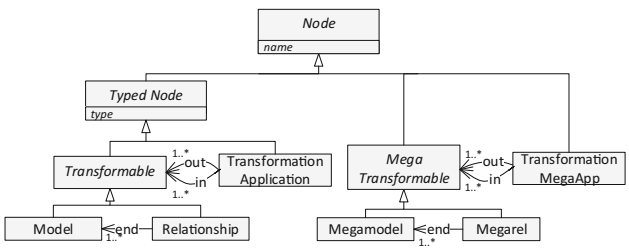

Fig. 2 Metamodel of an mgraph. We use the Ecore conventions for metamodeling [42] where the boxes represent *classes* of elements and the named directed associations between classes are called *references*

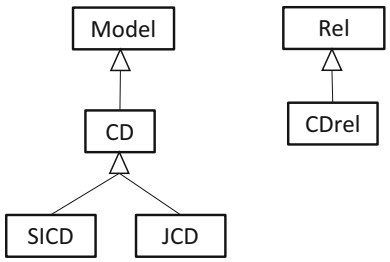

Fig. 3 A fragment of a type hierarchy with hollow arrows showing the type compatibility preorder $\preceq_{\text{TC}}$

that satisfies the following conditions for preserving all node classes C, references R and types:

1. $\forall n \in G \cdot n \in G_C \Rightarrow f_{\text{Node}}(n) \in G'_C$
2. $\forall n, n' \in G \cdot n' \in n.R \Rightarrow f_{\text{Node}}(n') \in f_{\text{Node}}(n).R$
3. $\forall n \in G_{\text{TypedNode}} \cdot f_{\text{Node}}(n).\texttt{type} \preceq_{\text{TC}} n.\texttt{type}$

A *typed mgraph homomorphism* is one where $\preceq_{\text{TC}}$ is equality. An *mgraph isomorphism* is a one where f_{Node} is a bijection and the types are equal.

Condition (1) ensures that f preserves node classes and condition (2) ensures that f preserves the endpoints of references. These are standard conditions for a homomorphism to be a structure-preserving mapping. Condition (3) additionally ensures that for typed nodes, f preserves type compatibility of nodes. Note that node names need not be preserved by f. The composition $h = f \circ g$ of mgraph homomorphisms is formed by composing functions $h_{\text{Node}} = f_{\text{Node}} \circ g_{\text{Node}}$. The three conditions hold for h; thus, it is an mgraph homomorphism.

2.2 Mega artifacts

Intuitively, all mega artifacts represent collections of artifacts.

Definition 4 (*Megamodels*) Let a model repository $\mathscr{R}$ of artifacts be given. A *megamodel* is a pair $\langle G, d \rangle$, where G is an mgraph and $d : G \to \mathscr{R}$ is a typed mgraph homomorphism, called the *dereferencing mapping*, that maps the nodes of G to the artifacts they represent in $\mathscr{R}$.

When it is clear from the context, we use a megamodel interchangeably with its mgraph.

Definition 5 (*Megarel*) A *megarel* is a tuple $\langle G, d, end \rangle$, where G is an mgraph restricted to containing only `Relationship` and `Megarel` nodes, $d : G \to \mathscr{R}$ is the dereferencing map and $end = \{\langle X, r \rangle | X$ *is a megamodel in* $\mathscr{R}$ *and* r *is the set of* `end` *references from nodes in* G *to nodes in* $X\}$.

Thus, a megarel is a "relationship-like" collection that has megamodels on its ends.

Definition 6 (*megaApp*) A *megaApp* is a tuple $\langle G, d, in, out \rangle$, where G is an mgraph restricted to containing only `Transformation Application` and `Transformation MegaApp` nodes, $d : G \to \mathscr{R}$ is the dereferencing map, $in = \{\langle X, r \rangle | X$ *is a megamodel or megarel in* $\mathscr{R}$ *and* r *is the set of* `in` *references from nodes in* G *to nodes in* $X\}$, and out is defined similarly to in.

Thus, both megarel and megaApp artifacts are connected to other artifacts in $\mathscr{R}$. Figure 4 gives an example of a repository showing different artifacts including the three megamodels `X`, `X1`, `X2`, megarel `XR`, as well as multiple models and relationships. To avoid visual clutter in this example, only six dereferencing mappings are shown for illustration purposes (dotted arrows). The remaining dereferencing mappings are implied by the naming; however, in general, names across the mapping may be different.

In the concrete syntax for megamodels and megarels that we use for illustrations in this paper, models are depicted as boxes; relationships are depicted as diamonds with binary relationships shown optionally as a line. A transformation application is depicted as an oval, with the input elements connected by dashed arrows pointing into the oval, and output elements connected by dashed arrows pointing out of the oval. All models, relationships and transformation applications have a label of form *name:type*, where the name is optional. Megamodels, megarels and megaApps are shown similarly to their non-"mega" counterparts but with thick borders. Furthermore, these elements are not typed.

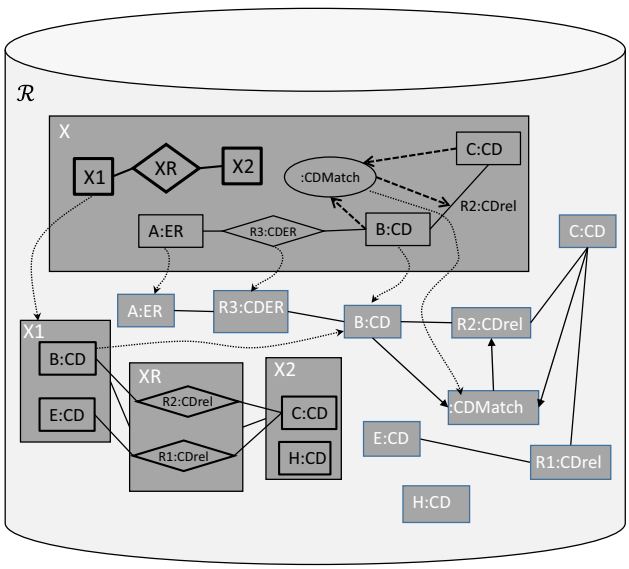

Fig. 4 An example of a repository including megamodels and a megarel showing the concrete syntax. Example dereferencing mappings from megamodel elements to corresponding artifacts are shown as dotted arrows

For example, in megamodel X at the top of Fig. 4, the box with label B : CD refers to the class diagram with name B; the diamond R3 : CDER refers to the corresponding CDER relationship artifact with name R3; the oval labeled :CDMatch refers to the corresponding transformation application artifact containing a record (e.g., trace mappings) of applying the transformation CDMatch to class diagrams B and C; and, the thick-bordered box labeled X1 refers to the megamodel X1 shown below it, which itself refers to models B and E.

2.3 Properties and transformations

Models and relationships can satisfy properties and participate in transformations. We define these below.

Definition 7 (*Property*) A *property* is a constraint on an artifact. Given an artifact A and a property P, we write $A \models P$ to denote that A satisfies P. Every property is defined for an artifact of a specific type. If A has type T_A, P has type T_P and the type compatibility preorder $\preceq_{TC}$ is given, the condition $(A \models P) \Rightarrow T_A \preceq_{TC} T_P$ must hold.

Thus, we assume that only artifacts compatible with the type of the property can satisfy the property.

A *transformation* is a function that maps models and relationships to other models and relationships.

Definition 8 (*Transformation*) A *transformation* is a pair $\langle \Sigma, F \rangle$ where

- $\Sigma = \langle I, O \rangle$ where $I \cup O$ is an mgraph, I is an mgraph called the *input signature* and O is an mgraph fragment called the *output signature*;
- F is an implementation of a function from megamodels with mgraph I to megamodels with mgraph $I \cup O$.

Thus, the transformation takes a megamodel structured by I as input and produces new models and relationships as output according to O. We make no assumptions about the language used for expressing properties or defining transformations. Note that the output signature is only an mgraph fragment since the output of a transformation can contain relationships that connect models given as the input. For example, Fig. 5 shows a signature for a transformation CDMerge that accepts two class diagrams and a relationship between them and produces the merged class diagram with relationships back to the original two class diagrams. The input signature consists of the models a, b and relationship r and the output signature has model ab with relationships ra and rb which connect to input models a and b, respectively. Written textually, the signature consists of

$$I = \{\texttt{a} : \texttt{CD}, \texttt{b} : \texttt{CD}, \texttt{r}(\texttt{a}, \texttt{b}) : \texttt{CDrel}(\texttt{CD}, \texttt{CD})\},$$
$$O = \{\texttt{ab} : \texttt{CD}, \texttt{ra}(\texttt{a}, \texttt{ab}) : \texttt{CDrel}(\texttt{CD}, \texttt{CD}), \texttt{rb}(\texttt{b}, \texttt{ab}) : \texttt{CDrel}(\texttt{CD}, \texttt{CD})\}.$$

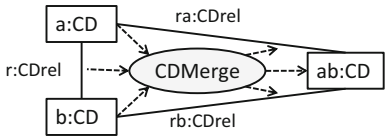

Fig. 5 Signature of a transformation CDMerge for merging class diagrams

Our focus in this paper is the application of a transformation *within* a larger megamodel to transform a portion of it. We can apply a transformation to any portion of a larger megamodel that matches the input signature.

Definition 9 (*Transformation application*) Given megamodel $X = \langle G_X, d_X \rangle$ and transformation F with signature $\langle I, O \rangle$, a *binding of F within X* is an injective mgraph homomorphism $b : I \to G_X$. Transformation F is *applied* to X at b, written $F(b)$, by using artifacts $d_X \circ b$ as the input and computing the output artifacts using F.

That is, the binding b of F within X identifies a subset of nodes in X that match I. An important special type of transformation we consider is one where the output is the same regardless of the order in which the input arguments are bound.

Definition 10 (*Commutative transformation*) Given a transformation F with a signature $\langle I, O \rangle$, we say that F is *commutative* iff for every pair of isomorphic bindings b, b' (i.e., mgraph isomorphisms of I), $F(b)$ is isomorphic to $F(b')$.

CDMerge in Fig. 5 is an example of a commutative transformation—given any two class diagrams M1, M2 related by a relationship R, the merged output is the same regardless of whether we use the binding {a := M1, b := M2, r := R} or {a := M2, b := M1, r := R}.

2.4 Polymorphism

Definition 9 allows the application of specific transformations to a megamodel. But to handle heterogeneity within a megamodel, we require support for *polymorphic* transformation application. For example, if we want to convert all the models in a megamodel to a common type (e.g., state machine, class diagram, etc., to UML models), we would like to use **map** and have it automatically use the correct converter for each model. Similarly, we may want to use **filter** to extract all models that have errors, but the property HasError may be defined differently for different types.

In programming, there are two main kinds of polymorphism: *universal* and *ad hoc* [5]. In universal polymorphism, the same function implementation can be used with a potentially infinite number of types where the type is either taken

as a parameter (*parametric polymorphism*) or is a subtype (*inclusion polymorphism*). A `length` function that computes the length of a list for any list type is an example of parametric polymorphism. Ad hoc polymorphism applies to a finite set of types, and the implementations for different types can vary arbitrarily, e.g., the + operator for adding integers and concatenating strings.

As in programming, in model management, different kinds of polymorphism can be used to define polymorphic transformations. Inclusion polymorphism for transformations has been studied (e.g., [18]) as a way to reuse a transformation for subtypes. For example, a transformation that converts a hierarchical state machine to Java code can safely be used for the state machine subtype that does not allow hierarchy. While inclusion polymorphism permits transformation reuse, it cannot take semantic differences between types into account. For example, a model management merge operation operates differently for state machines than for class diagrams. These situations require ad hoc polymorphism.

In this paper, we support heterogeneity using both inclusion and ad hoc polymorphism. A transformation is applied by binding the signature to artifacts in the repository. Inclusion polymorphism is addressed by allowing a transformation to apply to artifacts with more specialized types than those given in the signature. This is implemented in transformation application according to Definition 9 because mgraph homomorphisms use the type compatibility preorder $\preceq_{\mathrm{TC}}$ (see Definition 3).

Ad hoc polymorphism is supported by allowing the transformation name to be overloaded to support variant implementations of the same transformation with more specialized signatures. When a transformation with a given name is invoked, the *most specific* applicable variant is applied. We formally define what we mean by a polymorphic variant below.

Definition 11 (*Polymorphic variant*) Given a transformation F with a signature $\langle I, O\rangle$, a *polymorphic variant* of F is a pair $\langle F', \alpha_{FF'}\rangle$, where F' is a transformation with a signature $\langle I', O'\rangle$ with $name(F') = name(F)$ and $\alpha_{FF'} : I \cup O \rightarrow I' \cup O'$ is a bijective mgraph homomorphism called the *alignment mapping*. We call the restriction $\alpha_{FF'} : I \rightarrow I'$ of $\alpha_{FF'}$ to the input signature I, the *input alignment mapping*.

Thus, a polymorphic variant of a transformation F has the same name as F, and the alignment mapping shows the correspondence between the arguments. Since the alignment mapping is a bijection, all signatures of polymorphic variants have the same structure but more specialized model and relationship types. Note that a more general formulation of polymorphism can allow polymorphic variants to extend the signature (i.e., the alignment mapping is only injective). In this paper, we consider only the more restricted conception of polymorphism and leave the more general case for future work. In order to formalize the concept of "the most specific polymorphic variant," we define a specialization order relation over polymorphic variants.

Definition 12 (*Polymorphic variant specialization*) Given a transformation F with a signature $\langle I, O\rangle$ and polymorphic variants $\langle F', \alpha_{FF'}\rangle$ and $\langle F'', \alpha_{FF''}\rangle$, we say that $\langle F'', \alpha_{FF''}\rangle$ is *more specialized than* $\langle F', \alpha_{FF'}\rangle$, denoted by $\langle F'', \alpha_{FF''}\rangle \preceq \langle F', \alpha_{FF'}\rangle$, iff there exists a polymorphic variant $\langle F'', \alpha_{F'F''}\rangle$ of F', where $\alpha_{F'F''} \circ \alpha_{FF'} = \alpha_{FF''}$.

Definition 13 (*Most specific variant*) Given a transformation F with a signature $\langle I, O\rangle$, a polymorphic variant $\langle F', \alpha_{FF'}\rangle$, an mgraph G and an mgraph homomorphism $b : I \rightarrow G$, we call $\langle F', \alpha_{FF'}\rangle$ an *applicable variant* of F at b if there exists an mgraph homomorphism $b' : I' \rightarrow G$ such that $b' \circ \alpha_{FF'} = b$. $\langle F', \alpha_{FF'}\rangle$ is called *the most specific variant* of F at b iff there does not exist a different applicable variant $\langle F'', \alpha_{FF''}\rangle$ of F at b such that $F'' \preceq F$.

To illustrate, consider the family of polymorphic variants for the `Match` operation shown in Fig. 6a. This is based on

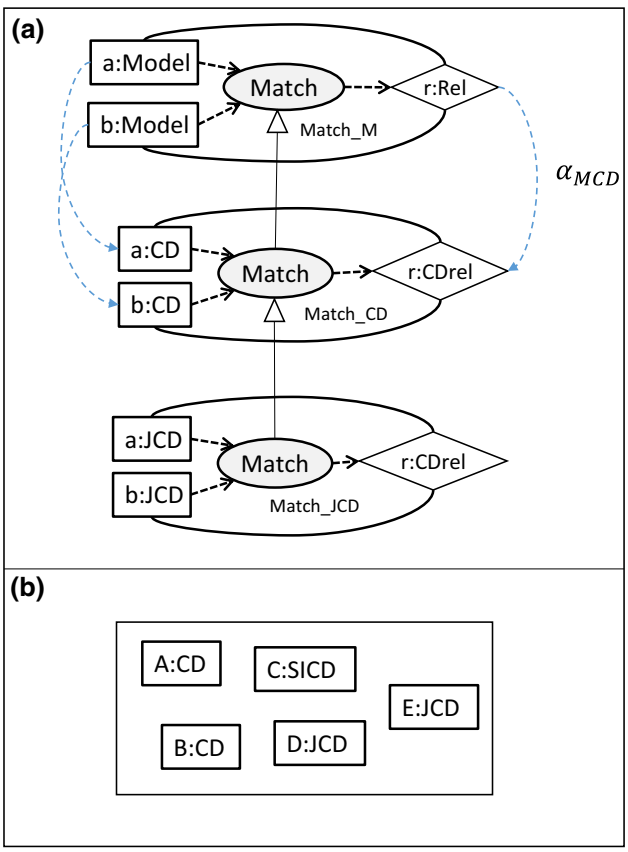

Fig. 6 **a** A family of polymorphic variants of the `Match` operation. To reduce visual clutter, only the alignment mapping α_{MCD} is shown between `Match_M` and its polymorphic variant `Match_CD`. The others are similar. **b** An example mgraph

the type compatibility relation shown in Fig. 3. The hollow arrows in Fig. 6a show the derived specialization relation between the variants. `Match_JCD` is a more specialized variant of `Match` than `Match_CD`. This is the case according to Defn. 12 because we can define `Match_JCD` as a polymorphic variant of `Match_CD` and compose this alignment mapping with the alignment mapping between `Match_CD` and `Match` to get the alignment mapping between `Match_JCD` and `Match`.

If `Match` is applied to the mgraph shown in Fig. 6b, the most specific variant that applies to the bindings {`a` := `D`, `b` := `E`} and {`a` := `E`, `b` := `D`} is `Match_JCD`.[1] For every other binding, the most specific variant is `Match_CD`. For a binding such as {`a` := `B`, `b` := `C`}, we rely both on ad hoc polymorphism since we use the variant `Match_CD` and an inclusion polymorphism since `SICD` is type compatible with `CD`.

In order for the collection operators we define in Sect. 3 to operate deterministically, we require that the polymorphism transformation mechanism selects a unique variant to apply in every situation. Thus, we require that there always be a unique most specific variant. This defines a soundness condition for families of polymorphic variants.

Definition 14 (*Soundness of polymorphism*) Given a transformation F with a signature $\langle I, O\rangle$, the family of polymorphic variants V_F of F is *sound* iff for all mgraphs G and mgraph homomorphisms $b : I \to G$, there exists a unique most specific variant $\langle F', \alpha_{FF'}\rangle \in V_F$ at b.

It is easy to see that the family of polymorphic variants shown in Fig. 6a always has a unique most specific variant for any binding {`a` := x, `b` := y} to models x and y; thus, it is a sound family of variants. Finally, we can generalize Definition 9 to support ad hoc polymorphic transformation application.

Definition 15 (*Polymorphic trans. application*) Given a megamodel $X = \langle G_X, d_X\rangle$, a transformation F with a signature $\langle I, O\rangle$ and a sound family of polymorphic variants V_F of F, a *polymorphic binding of F within X* is a pair $\langle b, \langle F', \alpha_{FF'}\rangle\rangle$, where b is an injective mgraph homomorphism $b : I \to G_X$ and $\langle F', \alpha_{FF'}\rangle \in V_F$ is the most specific polymorphic variant in V_F at b. F is *applied* to X at b, written $F(b)$, using artifacts $d_X \circ b$ as the input and computing the output artifacts using F'.

For example, if we assume that Fig. 6b represents a megamodel (i.e., rather than just an mgraph), an example of a polymorphic binding of `Match` in Fig. 6a is $\langle$ {`a` := `A`, `b` := `B`}, $\langle$`Match_CD`, $\alpha_{\text{MCD}}\rangle\rangle$.

[1] We assume that `Match` is a commutative model management operator; thus, the operator application on either binding will yield the same output relationship.

2.5 Traditional megamodeling operators

As discussed in Sect. 1, a number of model management operators have been defined, with *match*, *merge*, *diff*, and *slice* among them. For the illustrations in this paper, we require only one of them—a simple type of megamodel merge that we call **union**[2] The **union** operator combines the content of a set of megamodels into a single megamodel in which elements that refer to the same artifact are merged into a single element.

There are two possibilities for the set of input megamodels: (a) either they are an mgraph of megamodels and megarels, or (b) they are a set of megarels that share the same endpoints. Figure 7 illustrates both cases. In case (a), the result is a megamodel while in case (b), it is a megarel with the same endpoints as the inputs.

The union process can cause conflicts coming from the following two sources. If megamodel elements refer to the same artifact but the names of these elements differ, it is not clear which name to use for the merged element. To resolve this, we assume that the names in the union are a combination of the original names. Another conflict occurs when different artifacts are referred to by different elements using the same name. In this case, we assume that the names are made distinct in the union. Both of these conflict scenarios are illustrated at the bottom of part (a) in Fig. 7. The dereferencing mappings for these megamodels are not shown, but we assume that both `A` and `D` refer to the same model, and in the union, the name `A_D` is used. In addition, the element `C` in `X2` refers to a different model than `C` in `X3`, and the latter is assigned the name `C_1` in the result.

3 Megamodel collection operators

In this section, we define the set of megamodel collection operators we are proposing in this paper: **map** in Sect. 3.1, **reduce** in Sect. 3.2 and **filter** in Sect. 3.3. Their signatures are **map**$[\mathcal{T}] : \mathscr{P}(\mathcal{M}) \to \mathscr{P}(\mathcal{M})$, **reduce**$[\mathcal{T}] : \mathcal{M} \to \mathcal{M}$, and **filter**$[\mathcal{P}] : \mathcal{M} \to \mathcal{M}$, respectively, where $\mathcal{T}$ is the set of model transformations, $\mathcal{M}$ is the set of megamodels, $\mathcal{P}$ is the set of model properties, and $\mathscr{P}$ is the powerset operator.

All three operators are higher order and accept a transformation or a model property as a parameter (indicated in square brackets). The benefits of using higher-order operators in programming are well-known. In particular, they are useful for encoding common programming idioms (e.g., mapping, filtering, etc.) to allow for clearer and more concise programs. We illustrate this for model management in

[2] In this paper, we use bold font for the megamodel operators **union**, **map**, **reduce** and **filter** defined in this paper and italics for all other operators.

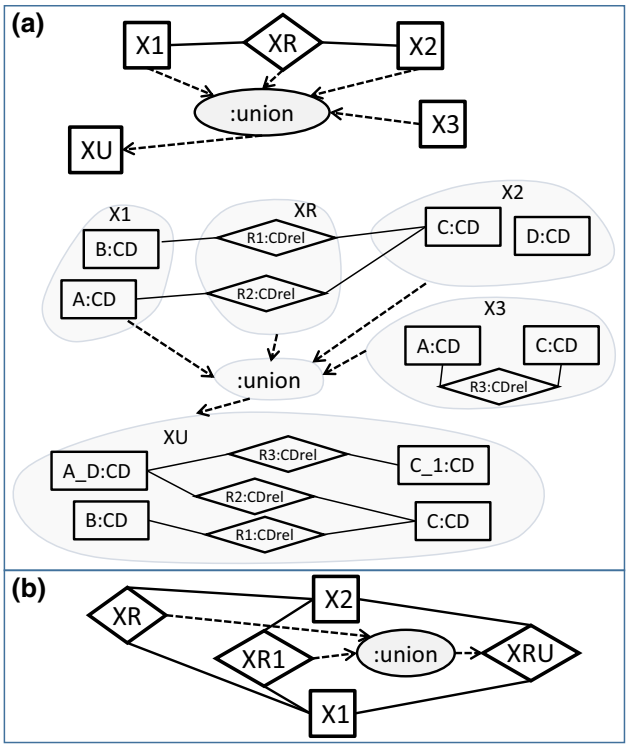

Fig. 7 An illustration of the **union** operator applied to **a** an mgraph of megamodels (megamodel contents shown underneath), and **b** a set of megarels that share the same endpoints

Sect. 4 by presenting scenarios that carry out complex procedures on megamodels using only relatively short workflows. In this section, we given the formal details of the operators, describing each operator as follows: first, the standard usage; second, the special adaptation needed to handle megamodels; third, the behavior defined as an algorithm, and finally, a discussion of how heterogeneity is handled.

3.1 Operator map

Standard Usage The usual behavior of a map operation is to traverse a collection (e.g., list, tree, etc.) and apply a function to the value at each node in the collection. The result is a collection with the same size and structure as the original with the function output value at each node. For example, given the list of integers $L = [10, 13, 4, 5]$ and the function *Double* that takes an integer and doubles it, applying map with *Double* to L yields the list [20, 26, 8, 10]. If the function has more than one argument, the mapped version can take a collection (with the same size and structure) for each argument, and the function is applied at a given node in the collection using the value at that node in each argument in the collection.

Adaptation for Megamodels Since a transformation input signature is an mgraph, the **map** operator generalizes the standard usage discussed above by applying the transformation to every possible *binding* of the input signature in the input megamodel(s). The collection of outputs from these applications forms the output megamodel. In the special case that the input signature consists of a single model, this is equivalent to the standard usage.

When the transformation signature consists of a single input and output type and uses a single input megamodel which happens to be a set (i.e., no relationships) of instances of the input type, then our **map** operator produces the same result as a standard map operator applied to a set. However, in the general case, **map** is more complex and differs from the behavior of the standard map. In particular,

1. The output megamodel may not have the same structure as the input megamodel since the structure is dependent on the output signature of the transformation.
2. The size of the output may not be equal to the size of the input. For example, if a transformation `FF` takes two models as input and produces one as its output, applying **map** to it on a megamodel with n models will produce as many as $n \times (n-1)$ output models since each pair of input models may be matched in a binding. At the other extreme, if no input models form a binding, then the output will be the empty megamodel.
3. When there are multiple input megamodels, each binding of the input signature is split across the input megamodels in a user-definable way.
4. When the transformation is commutative, we (may) want to avoid replication in the output due to isomorphic bindings. For example, if the transformation `FF` is commutative, we will get each output model twice since there are two ways to apply `FF` to a pair of models.

In what follows, we propose an operator **map** for handling megamodels while avoiding the above problems.

Operator Definition **map**$[F](\{X_e | e \in I\})$ applies a model transformation F with a signature $\langle I, O \rangle$ to a set of input megamodels $\{X_e | e \in I\}$ indexed by the input signature I and satisfying the precondition C1:

(C1) For all relationships $r(a, b) \in I$, either X_r is a megamodel and $X_a = X_b = X_r$ or X_r is a megaRel connecting X_a and X_b.

That is, an input relationship must either be taken from the same megamodel as its endpoint models or from a megaRel connecting the megamodels from which its endpoint models are taken. Note that the megamodels X_e need not be distinct; thus, multiple input arguments can be taken from the same megamodel.

map produces an output megamodel for each element of the output signature O. The behavior is defined by the algorithm in Fig. 8.

Algorithm: Apply map
Input: transformation F with signature $\langle I, O\rangle$,
megamodels $\{X_e | e \in I\}$
Output: set of megamodels $\{Y_e | e \in O\}$
1: **for** $(e \in O)$ { **let** $Y_e := \emptyset$ }
2: **for** (polymorphic binding $\langle b, \langle F', \alpha\rangle\rangle$ in $\{X_e | e \in I\}$) {
3: **if** F is `commutative` **then**
4: **if** isomorphism of b already done **then continue**;
5: **for** $(e \in O)$ { add element e of $F(b)$ to Y_e } }
6: **return** $\{Y_e | e \in O\}$

Fig. 8 Algorithm defining behavior of the **map** operator

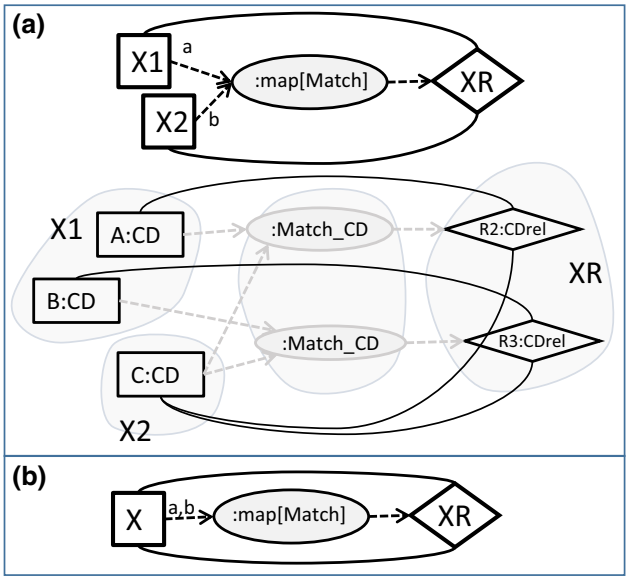

Fig. 9 **a** An illustration of applying **map** to the `Match` transformation using two input megamodels (megamodel content shown underneath); **b** using the same input megamodel for both arguments

We explain the algorithm using the illustration in Fig. 9a of applying **map** to the polymorphic `Match` transformation given in Fig. 6a. The input signature consists of $\{\texttt{a} : \texttt{Model}, \texttt{b} : \texttt{Model}\}$, and the output signature is $\{\texttt{r(a, b)} : \texttt{Rel}\}$. The diagram in Fig. 9a shows **map**(`Match`) applied to megamodels `X1` and `X2` to produce an output megarel `XR`. Thus, the input megamodels to the algorithm are $X_a :=$ `X1` and $X_b :=$ `X2`, and the one output, Y_r, corresponding to the output signature element `r`, produces the value for `XR`.

In line 1, the output megamodels are initialized to the empty megamodel. In our example, $Y_r = \emptyset$. Lines 2–5 iterate over all possible bindings of I in the input megamodels. In line 2, a fresh binding (i.e., previously unmatched) for the input signature of F is found in the input megamodels. Thus, in this example, a binding for `a` is drawn from `X1` and a binding for `b` from `X2`. Assume that this binding is $\langle\{\texttt{a} := \texttt{A}, \texttt{b} := \texttt{C}\}, \langle\texttt{Match\_CD}, {}_{\texttt{MCD}}\rangle\rangle$ since `Match_CD` is the most specific variant of `Match` (see Fig. 6a). Lines 3–4 check whether isomorphic bindings should be ignored because F is commutative. Binding isomorphisms do not occur in this example, so we illustrate them separately below. In line 5, the output of applying the transformation to the combined input binding is added to the output megamodels. Thus, in our example, `Match_CD` is applied at $\{\texttt{a} := \texttt{A}, \texttt{b} := \texttt{C}\}$ and the resulting `CDrel` relationship `R2` is added to Y_r. Line 6 returns the resulting output.

In our example, there are only two matches; thus, the resulting megarel contains two relationships. However, consider the alternative application of **map** to `Match` shown in Fig. 9b. Here, both input elements are taken from the input megamodel `X`. Assume that `X` contains all three models $\{\texttt{A} : \texttt{CD}, \texttt{B} : \texttt{CD}, \texttt{C} : \texttt{CD}\}$. In that case, there are six possible ways to match the input signature. However, since `Match` is designated as `commutative`, a binding

$$\langle\{\texttt{a} := n, \texttt{b} := m\}, \langle\texttt{Match\_CD}, {}_{\texttt{MCD}}\rangle\rangle$$

produces the same output as

$$\langle\{\texttt{a} := m, \texttt{b} := n\}, \langle\texttt{Match\_CD}, {}_{\texttt{MCD}}\rangle\rangle;$$

thus, only three applications of `Match` are used to produce the output.

Heterogeneity Operator **map** applies a transformation to all binding sites to which it is applicable in the input megamodels. Thus, when it is used with a polymorphic transformation, it can be applied to heterogeneous megamodels because different polymorphic variants are automatically chosen depending on the types of the artifacts at each binding site. For example, applying **map**[`Match`] to the megamodel in Fig. 6b yields 10 applications—one for every pair of models. Typically, a model management operator such as `Match`, that must be applicable to any type of model, will have a large number of polymorphic variants, while more type-specific transformations such as state machine flattening will have a more restricted family of variants (e.g., handling only variants of state machines).

3.2 Operator reduce

Standard usage There are different variants of the operator reduce (also called fold, aggregate, etc.) used in programming languages but they typically accept a binary function F and apply it over values $x_1, x_2, \ldots, x_n$ in a recursive collection (e.g., list, tree, etc.) by accumulating the intermediate values, e.g., $F(x_n, F(\ldots, F(x_3, F(x_2, x_1))\ldots))$. For example, applying reduce with the "+" operator to the list [1, 3, 1, 9] produces the sum 14.

Adaptation for megamodels In a similar way, we expect the **reduce** operator to accept a transformation F and use it to combine the elements of the input megamodel. Our approach is to view F as a rewrite rule, by repeatedly applying F in place and deleting the input elements until F can no longer

be applied. We must consider several issues:

1. What should be the criteria that F must satisfy for this process to terminate?
2. Since a megamodel is not a recursively defined structure and has no well-defined ordering on its elements, we cannot rely on a specific traversal path. Thus, F must be *confluent*—the final result of **reduce** should be same regardless of the order in which we apply F to the megamodel.
3. Since the input elements may have relationships to other neighboring elements in the megamodel, we must be careful to preserve this information when the relationships are deleted.

We address issues (1) and (2) in the definition of **reduce** below with appropriate assumptions on F. We address issue (3) by using relationship composition operators to construct new relationships to neighboring elements as needed. As an illustration, assume we are using **reduce** with the `CDMerge` transformation (see Fig. 5) to merge a megamodel of class diagrams and `CDRel` relationships. Figure 11 shows one iteration of the reduction. In step (1), `CDMerge` is applied to an arbitrarily chosen pair of models (in this case, `B` and `C`) to produce a new class diagram `BC`. In step (2), composition operators are invoked to connect `BC` to the neighbors of `B` and `C`. Finally, in step (3), the original models `B` and `C` are deleted together with all of their relationships.

Operator definition We now define a new operator **reduce**$[F](X)$ aimed to apply a transformation F to reduce the content of a megamodel X. We begin by making the following assumptions:

(I) We assume availability of polymorphic variants of the relationship composition operator `Compose` for all relationship combinations we encounter.
(II) In order to achieve confluence, F is required to be commutative and associative with itself and with all relationship composition operators used in item (I).
(III) In order for the reduction process to terminate, we put the constraint on F that it must be strictly reducing in output types: for every model type in the input signature, there must be fewer models of that type in the output signature; and, for relationship type in the input signature, there must be fewer relationships of that type in the output signature that are connected to output models on both (or all, for n-ary relationships) ends.

Figure 10 gives the algorithm for defining the behavior of **reduce**. In line 1, Y is initialized to the same value as the input. Lines 2–8 iterate for each binding of F in Y until no more can be found, and the algorithm terminates returning Y

Algorithm: Apply reduce
Input: transformation F with signature $\langle I, O\rangle$,
megamodel X
Output: megamodel Y
1: **let** $Y := X$
2: **for** (polymorphic binding $\langle b, \langle F, \alpha\rangle\rangle$ in Y) {
3: apply $F(b)$ generating output K';
4: **for** $(m \in K_{\texttt{Mod}}, m' \in K'_{\texttt{Mod}}, r(m, m') \in K'_{\texttt{Rel}})$ {
5: **for** $(m'' \in Y_{\texttt{Mod}}, r'(m'', m) \in Y_{\texttt{Rel}})$ {
6: **let** $r''(m', m'') :=$`Compose`(r', r);
7: add r'' to Y }}
8: delete elements in K from Y }
9: **return** Y

Fig. 10 Algorithm defining behavior of the **reduce** operator

(line 10). In the loop, for a given binding K (line 2), F is first applied to get K' in line 3. Then, lines 4–8 perform the steps as described in Fig. 11 to connect the neighbors of input models in K to the output models in K' using composition operators and then deleting the input models in K. For each output model m' with a relationship r to an input model m (line 4), and for each neighbor model m'' of input model m with relationship r' (line 5), a new relationship r'' is constructed directly from m'' to m' by composing r' and r (line 6) using a polymorphic `Compose` operator.

Heterogeneity Heterogeneity is handled in two ways with the **reduce** operator. First, if the transformation F has polymorphic variants, then these are applied according to the model types encountered. Thus, a single application of **reduce** can be used to handle the reduction of multiple model types within a megamodel. For example, we illustrated **reduce** using a CD-specific transformation `CDMerge`. Instead, we could use a polymorphic operation `Merge`. Second, when composing the relationships to neighboring models, the use of the polymorphic `Compose` operator takes into account the fact that the neighboring models may be of different types. For example, if in Fig. 11, model `A` were a state machine instead of a class diagram, then `f1` would need to be a state machine-class diagram relationship (say, `SMCDrel`) and the composition of `f1` and `fB` in Step (1) would need to use a polymorphic variant that composes an `SMCDrel` with a `CDrel` to produce an `SMCDrel`.

3.3 Operator filter

Standard usage Many languages provide a filtering operation to extract a portion of collection that satisfies some condition. For example, filtering the list [2, 5, 6, 8, 9, 1] using the property `isEven` produces the list [2, 6, 8].

Adaptation for megamodels The **filter** operator is similar and applies to megamodels. A property is given as the filtering condition, and the subset of elements that satisfy the property is used to produce the output. We distinguish between

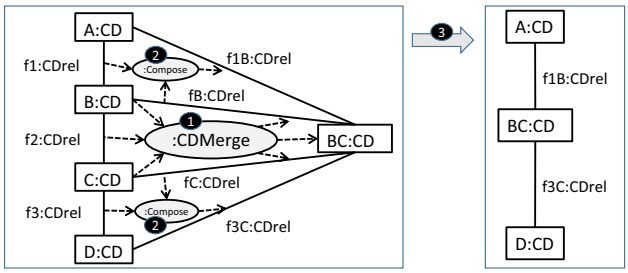

Fig. 11 An illustration of one iteration of **reduce**. First the merge is applied non-deterministically (step 1). Then, the relationships to the neighbors of the merged models are computed using appropriate composition operators. Finally, all input elements are deleted

Algorithm: Apply filter
Input: property P, megamodel X
Output: megamodel Y

```
1:  let Y := ∅;
2:  for (m ∈ X_Mod) {
3:      if P is a model property then
4:          if m ⊨ P then add m to Y;
5:      else add m to Y }
6:  for (r ∈ X_Rel) {
7:      if P is a relationship property then
8:          if r ⊨ P then add r to Y;
9:      else if r.end ∩ Y ≠ ∅ then add r to Y }
10: return Y;
```

Fig. 12 Algorithm defining behavior of the **filter** operator

model and relationship properties and treat them independently. Thus, a *model property* filters only models and keeps all relationships between the remaining models. A *relationship property* filters only relationships and does not affect the models.

filter differs from **map** and **reduce** in that it does not create new models or relationships; it just creates new references to existing models and relationships. Thus, all elements of the output megamodel refer to artifacts that are already referred to by elements of the input megamodel. This aspect of **filter** makes it an inexpensive operation compared to **map** or **reduce**.

If a property P, defined for a model or relationship type T, is used for **filter**, then it selects all elements of type T (or its compatible types) that satisfy the constraints in P (see Definition 7). It is also possible to give a type T as the property which is interpreted as the property *true*, satisfied by any instance of T (or its compatible types).

Operator definition **filter**$[P](X)$ filters megamodel X to produce the least sub-megamodel of X containing all the elements of X that satisfy property P.

The behavior of **filter** is given by the algorithm in Fig. 12. Line 1 initializes the output to the empty megamodel. Lines 2–5 iterate over the model elements in X. If P is a model property then the model is only added to the output if passes the satisfaction check (line 4). If P is not a model property, all models are added to the output (line 5). A similar algorithm is followed in lines 6–9 that iterate over relationship elements. The only difference is that if P is not a relationship property (and so it must be a model property), only those relationships that already have their endpoints in the output due to the filtering in lines 2–5 are added to the output.

Heterogeneity The use of polymorphic properties allows **filter** to be used with heterogeneous megamodels. For example, consider the model property `IsMinimal` which holds when a model is of the minimum size given its semantics. This check is complex and model type specific—e.g., checking a state machine for minimality is a different algorithm than checking a class diagram for minimality.

4 Application scenarios

In this section, we illustrate our collection-based heterogeneous megamodel management operators using several homogeneous and heterogeneous scenarios.

4.1 Scenario: experiment driver

The goal of this scenario is to apply a transformation on a megamodel and perform an experiment on the result of its application. Specifically, given a megamodel `XUML` containing a set of heterogeneous UML diagrams (namely, class diagrams (CDs) and state diagrams (SDs)), we wish to apply a polymorphic transformation `2Java` that translates a class diagram to its equivalent Java code and produces a `CD2JavaRel` traceability relationship from the CD to the Java code, and similarly, translates a state diagram to its equivalent Java code and produces a `SD2JavaRel` traceability relationship from SD to the Java code. Then, we wish to apply a polymorphic evaluation transformation `ECheck` on each of `CD2JavaRel` and `SD2JavaRel` in the megarels resulting from the transformation application. `ECheck` computes the number of elements in each transformed UML diagram that do not have Java counterparts. Finally, we would like to add these up via a `Sum` operation to learn the total number of incidents where this occurs. If the sum is greater than zero, then there is a problem with the transformation. Figure 13 shows the chain of operators required to accomplish this via the following steps:

1. Apply **map**[`2Java`](`XUML`) to produce `X1` which contains the Java code, and `XR` which is the megarel containing all heterogeneous relationships between `XUML` and `X1`.
2. Apply **map**[`ECheck`](`XR`) to produce a megamodel `X2` which contains the evaluation `ECheck` for each type of relationship in `XR`.
3. Apply **reduce**[`Sum`](`X2`) to produce the final result `X3` containing a single value which is the sum of the results

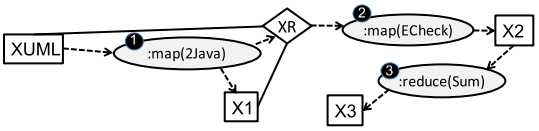

Fig. 13 Illustration of the experiment driver scenario

of **map**[ECheck](XR). A value greater than zero indicates a problem with the transformation application.

4.2 Scenario: IT standard change

Consider the motivating scenario from Sect. 1, where a company wants to change the naming convention across all of its modeling artifacts which are part of a megamodel X1. This is done using a Rename operation which has polymorphic variants for the different model types it is applied to. In addition, the company wants to eliminate the variety of different model types used for the same kind of information by standardizing UML state machines for state-like behavioral models and UML class diagrams for structural models. Finally, they would like to filter out all of the non-compliant models, where compliance is a polymorphic property, and merge the models of the same type to do some further analysis.

Figure 14 shows the chain of operators required to accomplish this scenario:

1. Apply **map**[Rename](X1) to produce X2, which contains all the modeling artifacts with the new naming convention.
2. Apply **map**[ToUML](X2) to map polymorphic ToUML_SM and ToUML_CD transformations on the state-like behavioral models and the structural models, respectively. This can be identified by a property in each model which indicates the model's nature (state-like behavioral vs. structural). This produces a new megamodel X3 containing only UML state machines and class diagrams.
3. Next, **filter**[NonCompliant](X3) is applied using the polymorphic property NonCompliant to produce a megamodel X4, which contains all the non-compliant models of X3.
4. Finally, **reduce**[Merge](X4) is applied using the polymorphic operator Merge to produce a megamodel X4 which contains all of the non-compliant models of X3. The result can then be reviewed and analyzed as needed by the company.

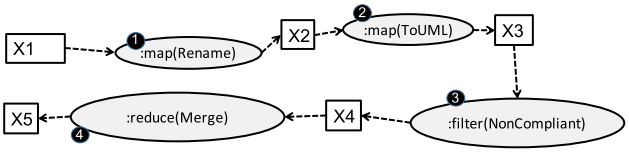

Fig. 14 Illustration of the IT standard change scenario

4.3 Scenario: megamodel transformation

The goal of this scenario is to apply a particular transformation on an entire megamodel. Suppose that we are given an input megamodel XCD consisting of class diagrams (CDs) related by class diagram relationships (CDrels), and we wish to transform it to a megamodel XER consisting of ER diagrams (ERs) related by ER diagram relationships (ERrels). We are also given the transformations CD2ER and CDrel2ERrel (see signatures in Fig. 15) which transform CDs to ERs and CDrels to ERrels, respectively. We would like to use our operators to accomplish this.

This transformation is illustrated in Fig. 16 and involves the following steps:

1. Apply **map**[CD2ER](XCD) which, based on its signature, applies only to the CDs in XCD and produces the megamodel X1 consisting of the ER versions of all the CDs in XCD as well as the megamodel relationship XR1.
2. Apply **map**[CDrel2ERrel](XCD) which, based on its signature, applies only to the CDrels in XCD and produces the megamodel relationship XR2 consisting of a set of ERrels with endpoints in X1. Note that the other arguments come from the megamodel relationship XR1 which contains the applications of the CD2ER transformation.
3. Apply **union**(X1, R) to produce the final megamodel XER which contains the corresponding ERs and the ERrels between them.

4.4 Scenario: mass refactoring

The goal of this scenario is to perform a mass refactoring on an entire megamodel in order to transform it to a more desirable version of that megamodel. Suppose we are given a megamodel XCD that contains unrelated class diagrams, a property PubAtt that represents models with public attributes and its negation NoPubAtt. We wish to find models satisfying PubAtt and refactor them so that public attributes become private attributes with public getter methods using the refactoring transformation PubGet. Figure 17 illustrates this scenario via the following steps:

1. Apply **filter**[PubAtt](XCD) to produce a megamodel X1 containing the sub-megamodel of XCD with models where property PubAtt holds.

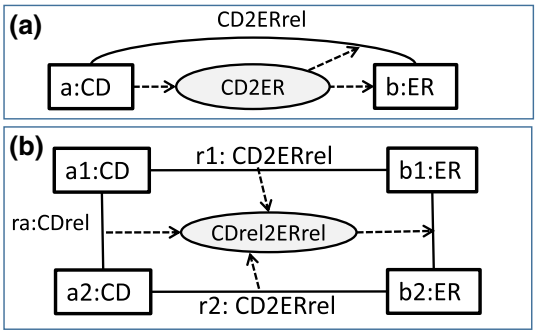

Fig. 15 Illustration of transformation signatures for megamodel transformation scenario. **a** Class diagram (CD) to entity relationship (ER) transformation, **b** CD relationship to ER relationship transformation

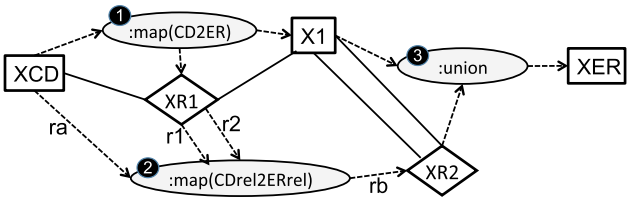

Fig. 16 Illustration of the megamodel transformation scenario

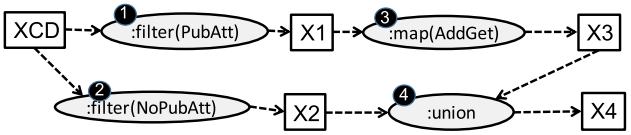

Fig. 17 Illustration of the mass refactoring scenario

2. Apply **filter**[NoPubAtt](XCD) to produce a megamodel X2 containing the sub-megamodel of XCD with models where property PubAtt does not hold.
3. Apply **map**[AddGet](X1) to transform the models with the undesirable property using a refactoring transformation AddGet which produces a megamodel X3.
4. Return a megamodel X4 = **union**(X2, X3) (see Sect. 2) which represents the refactored version of the original s.t. the property PubAtt no longer holds on any of its models.

4.5 Scenario: undesirable property removal

Consider a megamodel XCD which contains UML class diagrams and an undesirable property Mi that represents diagrams with multiple inheritance. In this scenario, we aim to identify all class diagrams that contain this property, refactor them using a predefined transformation to remove the property and merge the modified class diagrams. Figure 18 shows the workflow of operators required to accomplish this scenario:

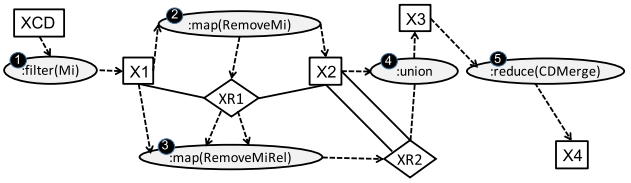

Fig. 18 Illustration of the undesirable property removal scenario

1. Apply **filter**[Mi](XCD) to produce a megamodel X1 containing the sub-megamodel of XCD with models where property Mi holds.
2. Based on the megamodel transformation pattern described in Sect. 4.3: apply **map**[RemoveMi](X1) to produce X2 which is the refactored version of X1 that no longer contains the undesirable property, and
3. apply **map**[RemoveMiRel](XR1) to produce the megamodel XR2 containing the relationships between the refactored models.
4. Apply **union**(X2, XR2) to produce X3 which is the megamodel containing the refactored models and relationships between them.
5. Apply **reduce**[CDMerge](X3) which applies the CDMerge operation described in Sect. 3 on class diagrams with relationship CDRel between them and produces a megamodel X4 where all the related class diagrams are now combined. The final result can now be compared with the result of merging the pre-refactored models which can be achieved by using **reduce**[CDMerge](XCD).

4.6 Scenario: megamodel slicing

As the final scenario, we implement a model management operator for heterogeneous megamodels. In previous work [37], we proposed a generic megamodel slicing approach and gave the algorithm using traditional pseudocode. Here, we show how to re-implement the algorithm using collection operators.

The intent of model slicing is to find all model elements related in a specific way to a given subset of elements, called a *criterion*. There are several types of model slicers (see [30]). In this paper, we focus on the *forward slice* operation that expands the criterion to the smallest subset containing all elements dependent on elements in the criterion. Figure 19a gives the general signature of a polymorphic model Slice operator, where sc is the criterion expressed as a submodel of model m. The output slice sl is another submodel of m. In this context, we use the unary relationship type Sub for expressing submodels of a model. A Sub relationship connected to a model m contains a set of links, each of which connects to an element of m; thus, it identifies a subset of elements in m.

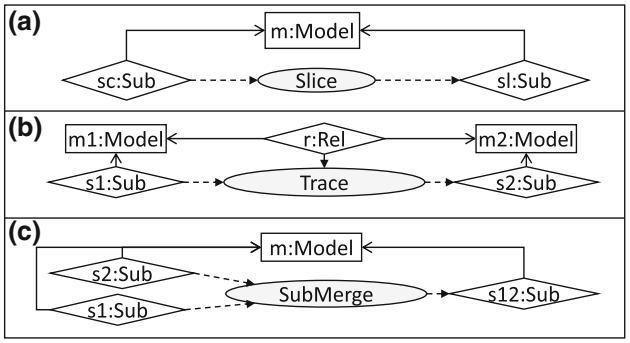

Fig. 19 Signature of polymorphic operators required in the megamodel slicing scenario: **a** `Slice`; **b** `Trace`; and **c** `SubMerge`

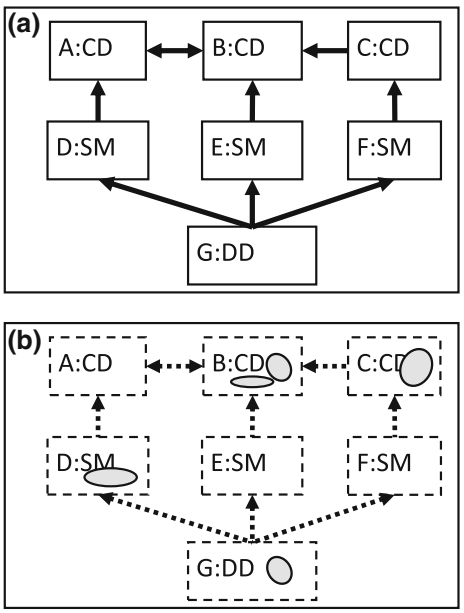

Fig. 20 **a** An example megamodel and **b** its sub-megamodel

The definition of *dependent on* clearly varies according to the model type; thus, `Slice` is a natural example of ad hoc polymorphism.

For megamodel slicing, we assume that a *sub-megamodel* is any set of submodels from a subset of models in the megamodel. For example, Fig. 20a shows a megamodel and Fig. 20b—a sub-megamodel consisting of submodels shown by the shaded ovals. The original megamodel is depicted using dashed lines for clarity, but the sub-megamodel consists only of the submodels. Similarly to submodels, we represent a sub-megamodel of a megamodel `X` as a unary megaRel connected to `X` and consisting of a set of `Sub` relationships connected to the models within `X`.

To slice a megamodel, it is not sufficient to just **map** polymorphic `Slice` over all models in the megamodel because we must take into account the fact that there may be inter-model dependencies across the relationships that connect models. To address these, we assume that we have a polymorphic `Trace` transformation with the signature as shown in Fig. 19b that takes a submodel of model `m1` and propagates it to the dependent submodel of model `m2` across the connecting relationship `r`. Finally, because using `Slice` and `Trace` can produce multiple submodels for the same model (i.e., one from `Slice` and zero or more from propagating using `Trace` for each neighboring model), we also assume that we have a polymorphic submodel merge operation to combine the submodels into a single submodel—this is provided by the `SubMerge` transformation with the signature in Fig. 19c.

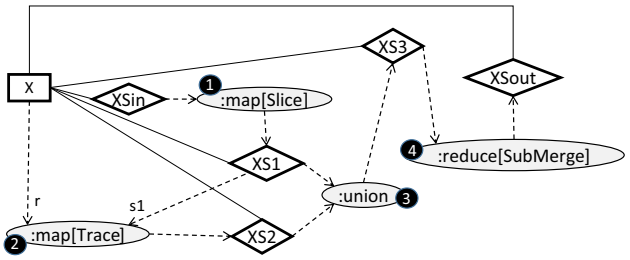

Fig. 21 Illustration of the megamodel slice scenario

Figure 21 shows the core steps of the megamodel slicing operation:

1. Apply **map**[`Slice`](`XSin`) using the submodels in sub-megamodel `XSin` as the criterion to produce a sub-megamodel `XS1` containing the slice submodels.
2. Apply **map**[`Trace`](`XS1`, `X`) to propagate the slice submodels in `XS1` to the corresponding submodels in neighboring models to produce a sub-megamodel `XS2`. Note that the `r` argument of `Trace` is taken from `X`, while the `s1` argument is taken from `XS1`.
3. Apply **union**(`XS1`, `XS2`) to combine the megamodels from steps (1) and (2) to produce `XS3`.
4. Apply **reduce**[`SubMerge`](`XS3`) to merge the multiple submodels of each model of `X` into a single submodel for each model. The result is a sub-megamodel `XSout`.

The slice sub-megamodel of a megamodel `X` is obtained by repeating these steps until the input `XSin` is equal to the output `XSout` (i.e., a fixed point is reached).

4.7 Summary

In this section, we presented scenarios which address specific types of megamodels, transformations and properties. Yet, they can be generalized as design patterns for similar reoccurring problems. For example, the mass refactoring scenario can be generalized for any problem that involves a megamodel which may contain elements with a certain property which should be removed. Similarly, the megamodel transformation scenario can be generalized to any problem that involves a transformation of one type of megamodel

to another, given the appropriate transformations between source and target models and source and target relations. We have observed that in the case of the megamodel transformation pattern, the relationship transformation can be induced from the model transformation; however, further analysis is outside the scope of this paper.

5 Analysis

In this section, we analyze complexity of the three operators we propose—see the summary in Table 1—and discuss the implications of this for scalability. Since these are higher-order operators, we define their complexity in terms of the complexity of the parameters they take. In the following, we assume that a transformation F and a property P are given and their complexity is $C_F(m)$ and $C_P(m)$, respectively, where m is the value of a metric over the input signature measuring the size of the input. For example, if the transformation F is `CDMerge` with signature given in Fig. 5, then m would be the total number of elements in arguments `a` and `b` plus the number of links in `r`. If F (resp. P) is polymorphic then we assume that C_F (resp. C_P) represents the maximum over the complexities of all its polymorphic variants. We assume F (resp. P) has v_F (resp. v_P) polymorphic variants.

Complexity of map $[F]$ The problem of finding bindings for an input signature is an instance of the *subgraph isomorphism* problem and is NP-complete [16]. What makes this tractable in practice is that the size of the input signature of F is typically small relative to the size of the megamodel to which **map**$[F]$ is being applied. For the algorithm in Fig. 8, the iteration in line 2 over possible bindings of the input signature I of F can execute up to n^k times, where $n = \Sigma_{e \in I}|X_e|$ is the number of models in all input megamodels and k is the number of model nodes in I. We assume that there is at most one relationship of each type between any given set of models in input megamodels so that a binding is uniquely determined by the mapping of model nodes in I. If F is commutative and I has q isomorphisms, the loop can execute $(n^k)/q$ times. Furthermore, in each iteration of the loop, F is called which has complexity $C_F(m)$ and up to v_F checks must be made to find the correct polymorphic variant of F. Since, q and v_F are constants, the complexity is $O(n^k \times C_F(m))$.

Table 1 Summary of complexity of the three operators

Operator	Complexity
map$[F](\{X\})$	$O(n^k \times C_F(m))$
reduce$[F](X)$	$O(n^2 \times C_F(m))$
filter$[P](X)$	$O(n^q \times C_P(m))$

Complexity of reduce $[F]$ Similarly to **map**, line 2 of the algorithm in Fig. 10 iterates over all possible bindings of the input signature I, but each time the input models and relationships are deleted. Thus, each input element participates in at most one binding. Furthermore, due to the assumption that F is strictly reducing, each iteration reduces the number of models and relationships. Thus, the number of iterations (and applications of F) is bounded by n, the number of models and relationships in X. The internal loops in lines 4–8 iterate once for every neighbor of a model M in I and a relationship of M to a model in O. This can iterate up to $r \times n$ times, where r is the number of relationships between O and I. In addition, each application of F requires v_F checks to get the correct polymorphic variant. Since r and v_F are constants, the overall complexity is $O(n^2 \times C_F(m))$.

Complexity of filter$[P]$ For a model property, the algorithm in Fig. 12 iterates n times. For a relationship property over a q-ary relationship, it iterates $w \times n^q$ times, where n is the number of models in X and w is the number of q-ary relationship types. Finally, each application of P requires checking through v_P polymorphic variants. Since w and v_P are constants, the complexity is $O(n^q \times C_P(m))$.

Discussion The analysis results in Table 1 show that the operators scale reasonably well for certain classes of application scenarios. Specifically, the complexity is at most quadratic (modulo the transformation/property complexity) in the size of the input megamodel when **map** is applied to a transformation with two or fewer input models, in all cases for **reduce**, and when **filter** is applied to either a model property or to a binary relationship property. Some scenarios exceed these limits (e.g., the megamodel transformation scenario in Sect. 4.3). We discuss future work for addressing scalability in Sect. 10.

6 Tool support

6.1 *MMINT* Overview

We implemented the megamodel collection operators described in this paper in the *MMINT* (Model Management INTeractive) workbench. Implemented in Java, *MMINT*[3] uses the Eclipse Modeling Framework (EMF) [42] to express models and the Eclipse Graphical Modeling Framework (GMF) to create custom editors for editing models and relationships, extending the MMTF model management framework [36]. The overall architecture of *MMINT* is illustrated in Fig. 22.

MMINT uses a distinguished *type megamodel* in which model types, relationship types and transformations are registered. Figure 23 shows a screenshot of a portion of the type

[3] Available at http://github.com/adisandro/MMINT.

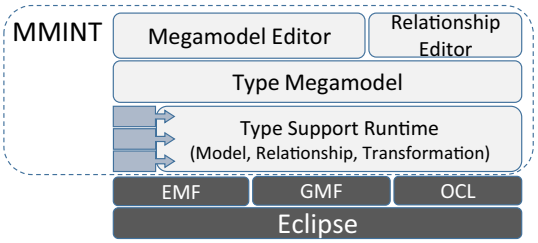

Fig. 22 Architecture of *MMINT*

megamodel used to implement examples in this paper. Here, model types are represented as yellow boxes, and relationship types are shown as rounded blue boxes. The subtyping relationship that implements the type compatibility preorder (see $\preceq_{TC}$ from Definition 3) is shown using the hollow-headed arrows between types. Transformations are ovals connected to their input and output types with named links (names are not shown to avoid clutter). The transformation signature information can be extracted directly from this model. The model also stores additional metadata such as whether a transformation is commutative or whether a relationship represents a composition.

In *MMINT*, a megamodel is referred to as a `MID` (Model Interconnection Diagram) and is managed through the `MID` editor. The runtime operation of *MMINT* is centered around the `MID` editor that allows an engineer to interactively create models and relationships, invoke transformations on them and inspect the results. Implementations for supporting tools such as type-specific editors, validation checkers, solvers and custom transformations can be plugged in and are managed by the type support runtime layer. In addition, *MMINT* includes a generic relationship editor which allows creating sets of links between model elements and editing them manually after they had been constructed by operators such as `Match`. This signature of `Match` is shown in Fig. 6a.

6.2 Support for polymorphism

MMINT includes support for both the inclusion and the ad hoc polymorphism as transformation subtyping in the type megamodel. For example, in Fig. 23, the base transformation called `Rename` (from the IT standard change scenario in Sect. 4.2) takes a `Model` as input and produces one as output. Subtypes that take `ClassDiagram`, `Entity Relationship`, `StateFlow` and `StateMachine` are polymorphic variants of `Rename`. A subtype must have the same signature structure as its super type but can use more specialized argument types.

At runtime, dynamic dispatch (more specifically, multiple dispatch to take all of the actual parameters into account) is used to determine the most specific variant (see Definition 13) when a polymorphic transformation is applied to a specific set of artifacts. When an engineer manually selects one or more MID elements and right-clicks to see which transformations can be applied, the editor shows the most specific variants of polymorphic transformations as well as more general variants, giving the engineer the flexibility to choose which variant to apply. For example, Fig. 24 shows a screenshot of the user manually applying a transformation to a selected model `cd_01` in the megamodel, and the `MID` editor shows that two variants of `Rename` and two variants of `ToUML` can be applied. When collection operators (see below) are used with a polymorphic transformation, the most specific variant is applied.

6.3 Implementation of collection operators

In *MMINT*, all transformations, including higher-order ones, are registered in the type megamodel. The three collection megamodel operators in this paper are shown in Fig. 25 with their inputs and outputs connected to the `MID` type, indicating that they take megamodels as input and produce them as output. In addition, each accepts a parameter shown within the angle brackets. We show the operator **union** as an unparameterized transformation.

MMINT implements the collection operators are implemented as specified in Sect. 3; however, it optimizes the handling of relationships. Specifically, if **map** or **reduce** is applied using a transformation that has relationships in its input signature, *MMINT* first finds bindings for the relationships and then forces them on the endpoint models. For example, the `CDMerge` transformation in Fig. 5 has two models and one relationship in its input signature. Each binding of the relationship also binds the models on its endpoints. Thus, it is sufficient to find all bindings of the relationship to get all bindings of the input signature. This optimization speeds up finding bindings when there are fewer relationships than models in the input signature and this is typically the case. A similar optimization is used with **filter** for relationship properties.

MMINT implements properties as a model or relationship subtype containing additional well-formedness constraints, but it does not change the metamodel of its super type. For example, the `NonCompliant` type in Fig. 23 is used in the IT standard change scenario (see Sect. 4.2). It contains the following OCL code:

```
NotCompliant (UMLSM):
transitions->exists(
    t | t.guard.oclIsUndefined()
)
```

We implemented the algorithms in Figs. 8, 10 and 12 for the three operators (and **union**) in Java and plugged them into the type support runtime layer as transformation definitions. At runtime, when an engineer selects a MID and right-clicks

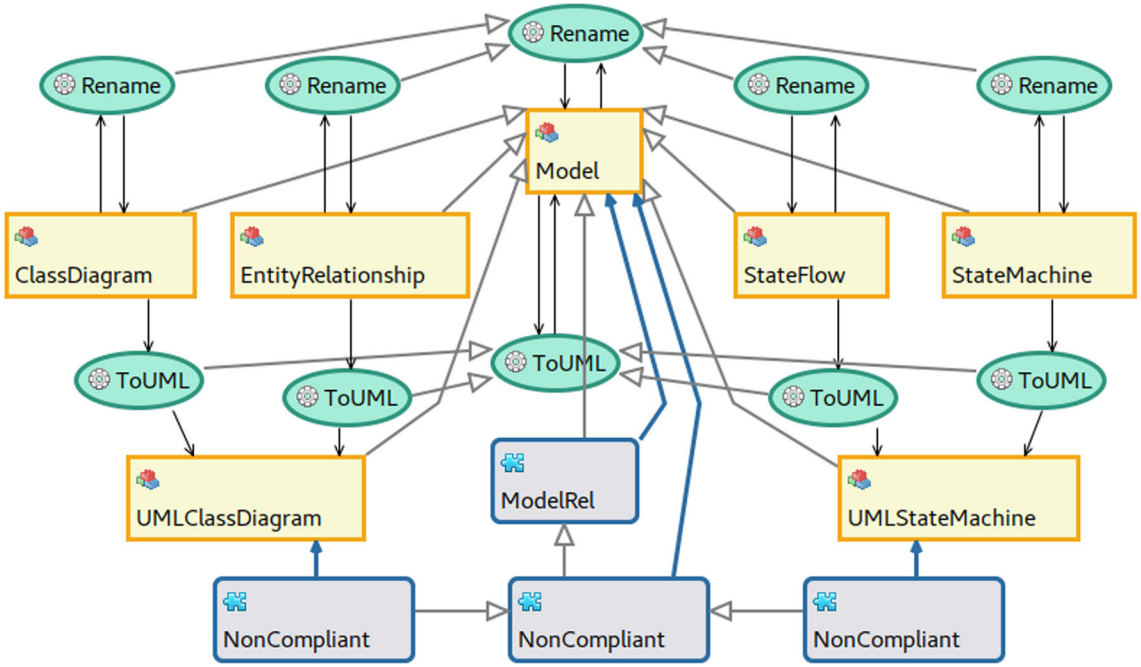

Fig. 23 A fragment of the type megamodel in *MMINT* used for the examples in this paper, showing model types, operators and properties

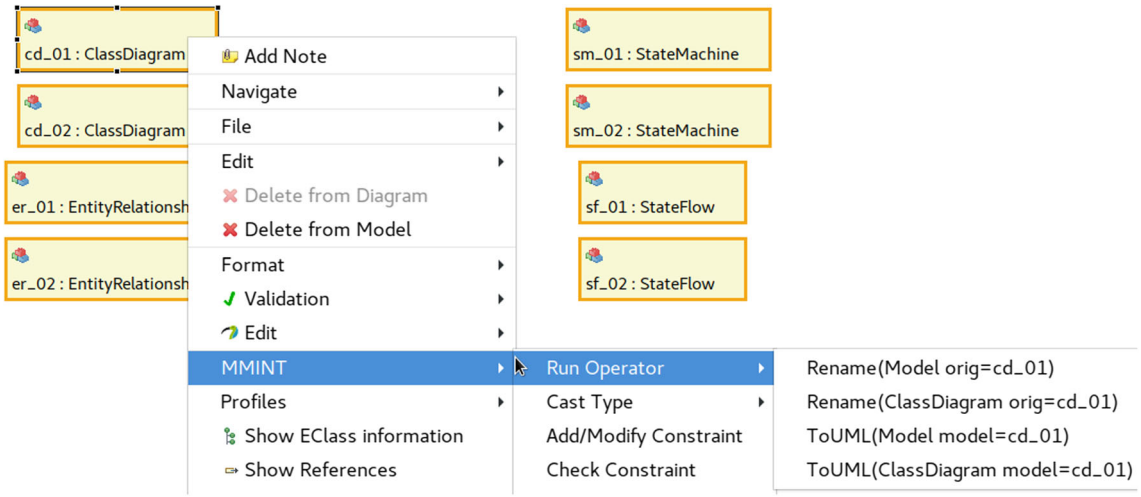

Fig. 24 Screenshot of megamodel for IT standard change scenario in Sect. 4.2 being built in the *MMINT* megamodel editor

to see what transformations are available to apply, he/she sees these collection operators and can select one. If the operator is parameterized, then a second dialog showing the choices for the parameter appears. For example, Fig. 26 gives a screenshot of the IT standard change scenario being built in the megamodel editor. It depicts the engineer in the final step of the IT standard change scenario, invoking the **reduce** operator.

7 Evaluation: scalability

In this section, we report on experimental results for the scalability of the *MMINT* implementation of the megamodel collection operators. Since a collection operator is higher order and takes a transformation (for **map** and **reduce**) or a property (for **filter**) as an argument, the time of operator application is a combination of the collection operator *overhead* and the time for transformation/property execution. In this investigation, we were careful to decouple these two factors and focus on the collection operator overhead.

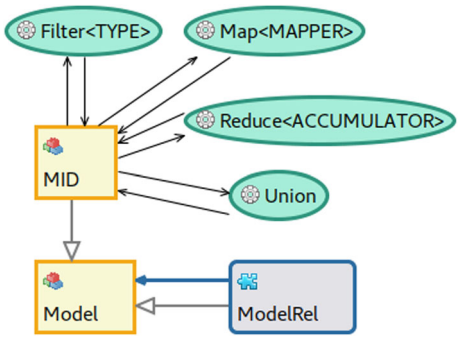

Fig. 25 A fragment of the type megamodel in *MMINT* showing the megamodel operators

All three operators function by first finding binding sites for the transformation/property and then selecting the appropriate polymorphic variant to apply. Thus, we investigate the following four research questions:

(RQ1) How does the overhead cost vary with the number of transformation/property binding sites in the input megamodel?

(RQ2) How does the overhead cost vary with the number of polymorphic variants and with the depth of the polymorphic variant hierarchy?

(RQ3) Is the observed performance of the operators consistent with the complexity analysis in Sect. 5?

7.1 Experimental setup

In general, a transformation can have an arbitrary signature expressed as a megamodel of models and relationships. This has a systematic blow-up effect on the overhead (as captured by the complexity analysis in Sect. 5) since for a given megamodel, the larger the input signature, the more binding sites are possible and have to be checked. To control for this factor, we limit the input signatures to two kinds: one that takes a single model and one that takes a single relationship with its endpoint models. We assume we can extrapolate the experimental results for these cases to more complex signatures.

We use the following artifacts in the experiments:

1. A set $\texttt{PolyModelType}_0 \ldots \texttt{PolyModelType}_n$ of model types that inherit from an abstract model type `PolyModelType`, and a distinct model type `OtherModelType` (referred to as `PMT` and `OMT`, respectively).
2. A set $\texttt{SleepM}_0 \ldots \texttt{SleepM}_n$ of polymorphic transformations with the signature $\texttt{SleepM}_i(\texttt{m} : \texttt{PMT}_i)$. Each transformation operates by sleeping for a configurable amount of time and producing no output.
3. A set $\texttt{PolyRelType}_0 \ldots \texttt{PolyRelType}_n$ of binary relationship types that inherit from an abstract relationship type `PolyRelType`, and a distinct type `OtherRelType` (referred to as `PRT` and `ORT`, respectively). The $\texttt{PRT}_i$ can only have models of type $\texttt{PMT}_i$ as endpoints, and `ORT` can only have models of type `OMT` as endpoints.
4. A set $\texttt{SleepR}_0 \ldots \texttt{SleepR}_n$ of polymorphic transformations with the signature $\texttt{SleepR}_i(\texttt{m1} : \texttt{PMT}_i, \texttt{m2} : \texttt{PMT}_i, \texttt{r(m1, m2)} : \texttt{PRT}_i(\texttt{PMT}_i, \texttt{PMT}_i))$. Similarly to (2), each of these sleep for a configurable amount of time and produce no output.

We control and isolate the time taken by a transformation by using the `SleepM/R` transformations set to sleep for a fixed amount of time. To control the number of transformation binding sites in a megamodel, recall from Definition 9 that a binding site is a group of models/relationships having the same shape and types as the input signature of a transformation, while a non-binding site has the same shape but not the correct types. Thus, to generate a binding site for `SleepM`, we use a single `PMT` model and for `SleepR` we use a `PRT` relationship with its two `PMT` model endpoints. To gen-

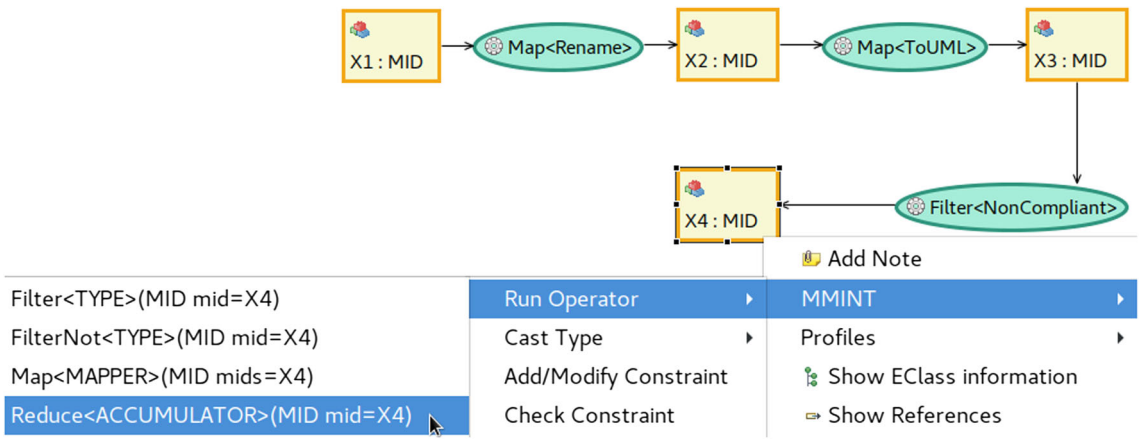

Fig. 26 Screenshot of megamodel for IT standard change in Sect. 4.2 being built in the *MMINT* megamodel editor

erate a non-binding site for `SleepM`, we use an `OMT` model, and for `SleepR` we use an `ORT` relationship.

For example, we can generate a megamodel containing 100 models, 50 of which are of type `OMT`, and 5 each of type $\texttt{PMT}_0$ to $\texttt{PMT}_9$. If we run **map**[`SleepM`], it will ignore all of the models of type `OMT`, and it will apply the polymorphic transformation variants $\texttt{SleepM}_0$ to $\texttt{SleepM}_9$ five times each.

We ran the experiments[4] on a Debian Sid linux system with an Intel Core i7-3720QM CPU and 16 GB of RAM. The relevant software installed includes Eclipse 2018-12 and OpenJDK 11.

7.2 Results

Figures 27, 28 and 29 show the *MMINT* overhead to run each of the megamodel collection operators, when fixing the number of polymorphic variants to 10. The solid lines are used for models (i.e., **map**[`SleepM`], **reduce**[`SleepM`], **filter**[`PMT`]), while the dashed lines are used for relationships (i.e., **map**[`SleepR`], **reduce**[`SleepR`], **filter**[`PRT`]). On the horizontal axis, we vary the number of sites in the megamodel for the `Sleep` operator (binding + non-binding), from 0 to 1000. On the vertical axis, we measure the *MMINT* overhead time of running the collection operators (i.e., with the time taken by the sleep transformation removed). The different colored lines represent different percentages of binding sites, from 0% to 100%. For example, in Fig. 27, the dashed red line represents the *MMINT* overhead for applying **map**[`SleepR`] to megamodels with 25% binding and 75% non-binding sites as the size of the megamodel (i.e., the total number of sites) increases. We ran the experiments multiple times and show the mean overhead time. Confidence intervals are computed using a Student's t-distribution with a confidence level of 95%.

These graphs address RQ1. Figure 27 for **map** indicates that the overhead increases linearly as the size of the input megamodel grows, and changing the ratio of binding sites to non-binding sites seems to have negligible effect. The overhead for `SleepR` is consistently higher than for `SleepM`, which is expected since the binding site for `SleepR` has two models and a relationship. However, the increment is by a constant amount. Note that even for large megamodels with 1000 models, the overhead is still below 1 s. This suggests that the overhead may be a negligible factor when compared to the time taken for the transformation being applied by **map**. For example, a simple transformation `Cap` that capitalizes the names of a model's elements takes 8 s when applied to 1000 models with 100 elements each.

The results for **reduce** and **filter** (Figs. 28, 29) are similar to **map** with some small deviations. Larger megamodels,

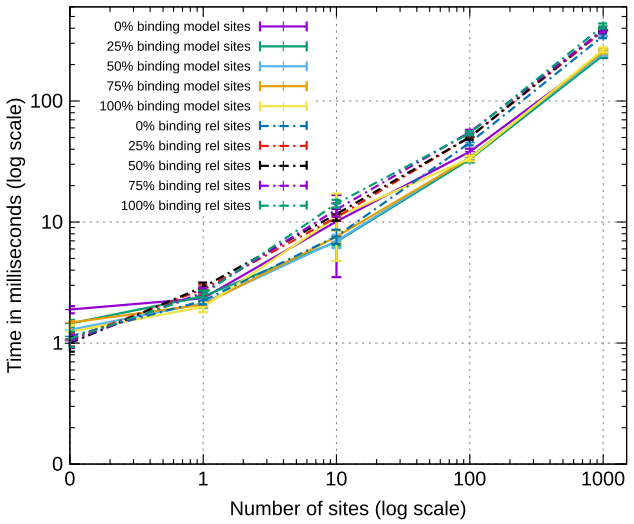

Fig. 27 Overhead in MMINT to run **map**[`SleepM`] (solid lines) and **map**[`SleepR`] (dashed lines) with 10 polymorphic variants

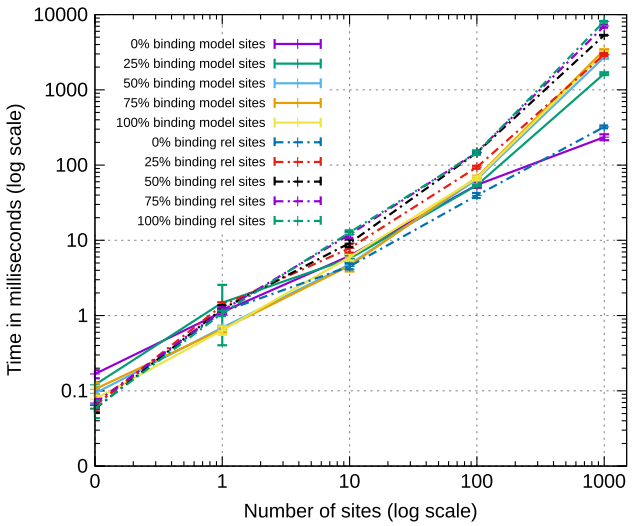

Fig. 28 Overhead in MMINT to run **reduce**[`SleepM`] (solid lines) and **reduce**[`SleepR`] (dashed lines) with 10 polymorphic variants

having more binding sites incurs an increased overhead time in **reduce**. The implementation of **reduce** reuses part of the infrastructure in place for **map**, while it could introduce a number of optimizations tailored to its different looping requirements. We plan to address this in future work.

Figure 30 addresses RQ2 by showing the impact of the number of polymorphic variants on the overhead time, when fixing the number of binding sites to 1000, and non-binding sites to 0. On the horizontal axis, we vary the number of variants, from 1 to 30, while the different lines represent the **map**, **reduce**, **filter** collection operators. As in the previous graphs, the solid lines are used for models, the dashed lines are used for relationships, and the vertical axis plots the average overhead time with same confidence intervals as in the

[4] Detailed instructions to reproduce experiments are available at https://github.com/adisandro/MMINT/wiki/Publications:-SoSyM19.

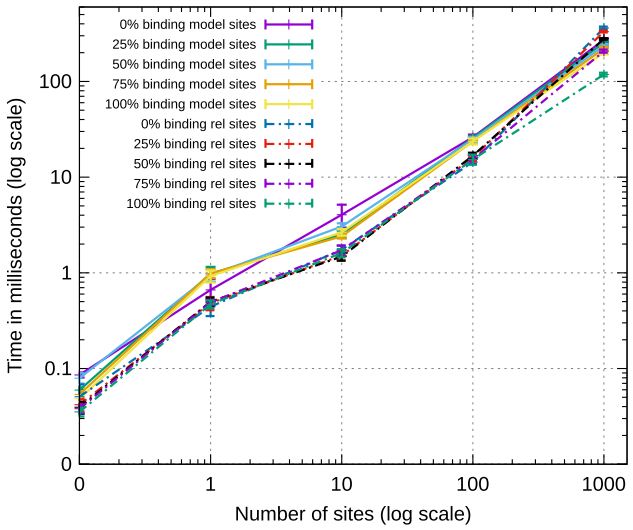

Fig. 29 Overhead in MMINT to run **filter**[PMT] (solid lines) and **filter**[PRT] (dashed lines) with 10 variant types

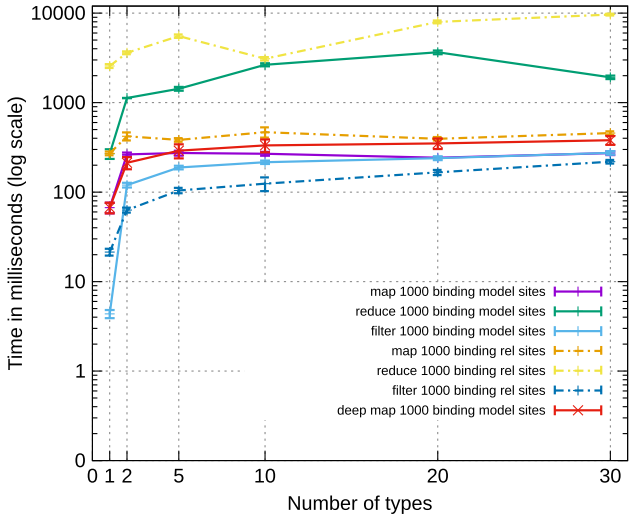

Fig. 30 Overhead in MMINT to run collection operators with different numbers of polymorphic variants

previous graph. An extra **map** line has been added to compare the effect of having a deep variant type hierarchy, i.e., the one where each $\texttt{PMT}_i$ inherits from $\texttt{PMT}_{i-1}$.

Across all cases, we see a similar pattern where the overhead jumps to a higher level when more than one variant is required. This is expected because the polymorphic machinery is invoked only if there is more than one variant. After the jump, the time increases very gradually (within error margins) as the number of variants increases. We only tested up to 30 variants of a polymorphic transformation, but since each variant applies to a different model type, having 30 model types in a single megamodel is a reasonable pragmatic upper bound. As a comparison, even if each diagram type (e.g., sequence diagram, class diagram, etc.) in UML2 was treated as its own model type, and they all occurred in the same megamodel, this would produce only 15 model types.

There is a clear increase in overhead times, in order, for **filter**, **map** and **reduce**. This is expected as these are increasingly complex operators with correspondingly complex implementations. The increase from **filter** to **reduce** is 1000-fold. However, once again, when compared to the time taken for the transformation being applied, this difference is not significant. An experiment was also done to compare the shallow variant hierarchy used in all the experiments (where all variants inherit from a single base type) and a deep variant hierarchy where all variants are in single inheritance chain (red line). This showed a small increase in time as the number of variants increased. We conclude that the structure of the inheritance hierarchy is not a significant factor in overhead time.

We now consider RQ3 and the complexity formulas given in Table 1. Recall that we are only measuring overhead in these experiments; thus, the transformation (resp. property) component $C_F(m)$ (resp. $C_P(m)$) of the complexity is ignored. Note that input megamodel size in the complexity formulas is measured in terms of the number of models, whereas in Figs. 27, 28 and 29 the size is measured as the number of sites (binding + non-binding). We account for this below.

First consider the `SleepM` experiments. Here, the number of sites is the number of models; thus, the complexity formulas are directly applicable. For **map**, since $k = 1$, the complexity should be $O(n)$ which is what is observed. For **reduce**, the complexity is $O(n^2)$, but this is a worst case bound that assumes that each model is connected to each of its neighbors (i.e., a fully connected graph). In our experiment, there are no relationships between the PMT models; thus, we again expect a linear increase. Finally, for **filter**, since we are using a model property, $q = 1$ and this means the complexity is $O(n)$ which is again linear as we observed.

For the `SleepR` experiments, the number of sites is the number of relationships in the input megamodel. The input megamodels are produced by adding a fixed number r of relationships to a randomly generated set of up to $2r$ models. If we assume that $2r$ is the number of models, the formula for **map** yields $O(r^2)$ since $k = 2$ for `SleepR`. Despite this quadratic complexity, we observe a linear time increase for **map**. We can assume that this is due to the optimization discussed in Sect. 6.3 where finding bindings is driven by relationships in the input signature. Since there is only one relationship in `SleepR`, we would expect a linear time increase in the number of relationships. A similar argument applies to **reduce** and **filter**.

We conclude that, subject to the experimental design and *MMINT* optimizations, the observed results are consistent with the analytical complexity results.

8 Evaluation: safety case change impact assessment case study

In this section, we evaluate the applicability of the collection operators to implement a model management workflow with industrial relevance. In safety critical domains, such as automotive, companies are beginning to use an artifact, called a *safety case* [22], to present an argument showing that a developed system meets its safety requirements. As a system changes due to a variety of reasons (e.g., bug fixes, adding features, etc.), its safety case needs to evolve as well. To support the safety engineer in this costly and labor-intensive activity, we have, in other work [25], proposed a change impact assessment (CIA) model management workflow to partially automate determining the impact of system model changes on the safety case and have validated the approach with our industrial partners [15,24,26]. The CIA workflow was originally defined without considering the megamodel collection operators and does not use them.

In this case study, we re-implement the CIA workflow using megamodel collection operators in order to qualitatively assess the usability and effectiveness of the operators as well as their implementation in *MMINT*.

8.1 Example: the power sliding door (PSD)

We illustrate the CIA workflow using a simple example of a power sliding door (PSD) system, presented in Part 10 of the ISO 26262 functional safety standard [20] for the automotive domain. PSD is an automotive subsystem that controls the behavior of a power sliding door in a car. The system has an *Actuator* that is triggered on demand by a *Driver Switch*. As per the standard, the power sliding door system is considered an *item*, with an architecture shown in Fig. 31. The *Driver Switch* input is read by a dedicated electronic control unit (ECU), referred to as *AC ECU*, which powers the *Actuator* through a dedicated power line. The vehicle equipped with the item is also fitted with an ECU, referred to as *VS ECU*, which is able to provide the vehicle speed. The system includes a safety element, namely a *Redundant Switch*. Including this element ensures a higher level of integrity for the overall system.

As shown in Fig. 31, the *VS ECU* provides the *AC ECU* with the vehicle speed. The *AC ECU* monitors the driver's requests, tests if the vehicle speed is less than or equal to 15 km/h, and if so, commands the *Actuator*. The *Redundant Switch* is located on the power line between the *AC ECU* and the *Actuator*. It switches on if the speed is less than or equal to 15 km/h, and off whenever the speed is greater than 15 km/h. It does this regardless of the state of the power line (its power supply is independent). The *Actuator* operates only when it is powered.

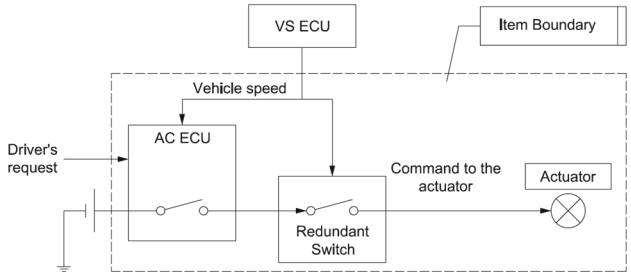

Fig. 31 Power sliding door system with redundancy [20]

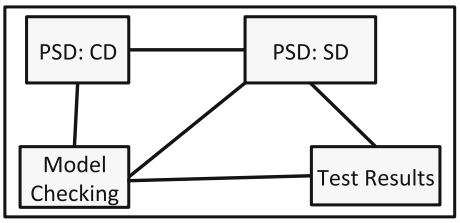

Fig. 32 System megamodel for the power sliding door example [26]

The PSD system software design consists of a UML class diagram, a sequence diagram, analysis models and traceability relationships connecting these shown as a megamodel in Fig. 32. The full details of these models are available in [25]—we omit them here as they are not relevant for our purpose.

Accompanying the system models is the safety case for the PSD system, shown in Fig. 33. It is represented using Goal Structured Notation (GSN) [17], a commonly used modeling language for safety cases. A safety argument in GSN is organized into a tree which includes element types *goal* (box), *strategy* (parallelogram) and *solution* (circle). The root element is the overall goal to be satisfied by the system, and it is gradually decomposed (possibly via strategies) into subgoals and finally into solutions, which are the leaves of the safety case representing the types of evidence obtained from analyzing the system. We extend GSN with an additional notation (small square boxes on the bottom right of the goals) used to reflect the Automotive Safety Integrity Level (ASIL) of the goal. The ASIL is a risk classification scheme defined in the ISO 26262 [20] and is used to indicate the risk associated with a goal. The values can be QM, A, B, C or D, ordered from the lowest to the highest risk.

For example, in Fig. 33, the top level goal **G1** is "Avoid activating the actuator while the vehicle speed is greater than 15km/hr." This is decomposed into sub-goals **G1.1**–**G1.4** via an "AND decomposition" strategy, **S1**, which means that all sub-goals need to hold in order for the parent goal to hold. **G1.1** is assigned an ASIL level C (which would have been determined during the hazard analysis activity) and is linked to a supporting solution **Sn1.1** "Software Verification Report (9.5.3)—Unit Testing Methods 1a, 1b, 1e" via strategy **S1.1**.

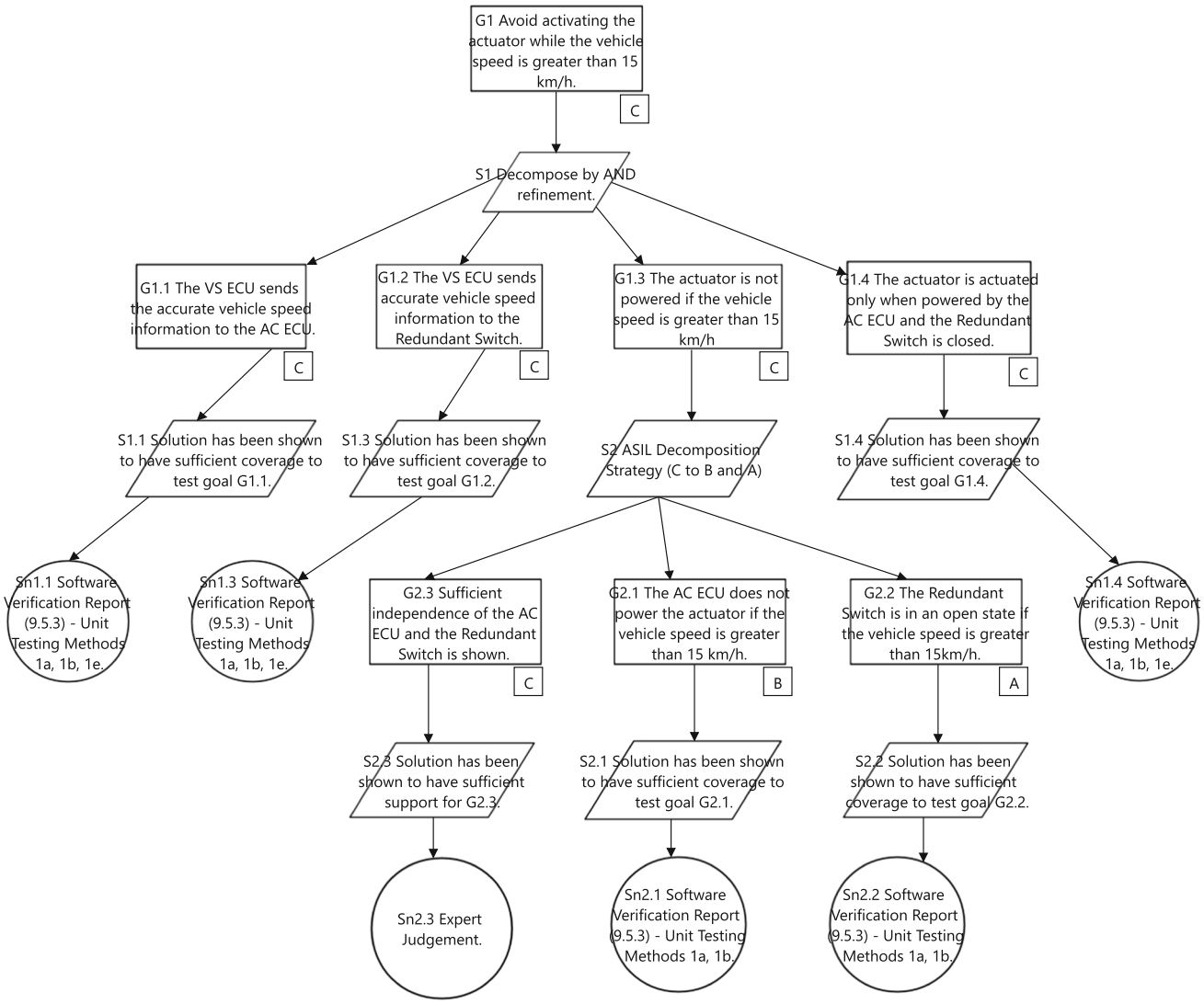

Fig. 33 Safety case for power sliding door [15]

8.2 CIA workflow

The CIA workflow assumes that the system is represented as a heterogeneous megamodel consisting of various models connected by relationships (e.g., Fig. 32). In addition, it assumes that there is a traceability relationship between the safety case and the system model showing what parts of the system are mentioned in each goal. For example, goal **G1.2** in Fig. 33 mentions the system elements *VS ECU*, *Redundant Switch* and the attribute *vehicle speed* which the traceability relationship should map to the corresponding parts of the class diagram and sequence diagram represented by the megamodel in Fig. 32.

The CIA workflow takes as input a safety case with a traceability relationship to the system model and the set of changes in the system model. This version of the CIA workflow assumes that changes consist of deleting or modifying elements.[5] The output of the CIA workflow is an *annotation* of the safety case indicating where and how it may be impacted by the system changes.

As an example of an application of CIA, Fig 34 shows the output when the system change consists of the removal of the redundant switch. At a high level, it shows which parts need to be *revised* (due to direct linkage to affected system elements), *rechecked* (due to indirect linkage to affected system elements), or can be *reused* (not affected by the changes). Figure 34 shows that all safety case elements that refer directly to the redundant switch must be revised, while any related elements must be rechecked for their content (and/or state) validity. Also, by removing the redundancy mechanism, the ASIL decomposition strategy is no longer valid as per ISO

[5] See [25] for issues concerning handling added elements.

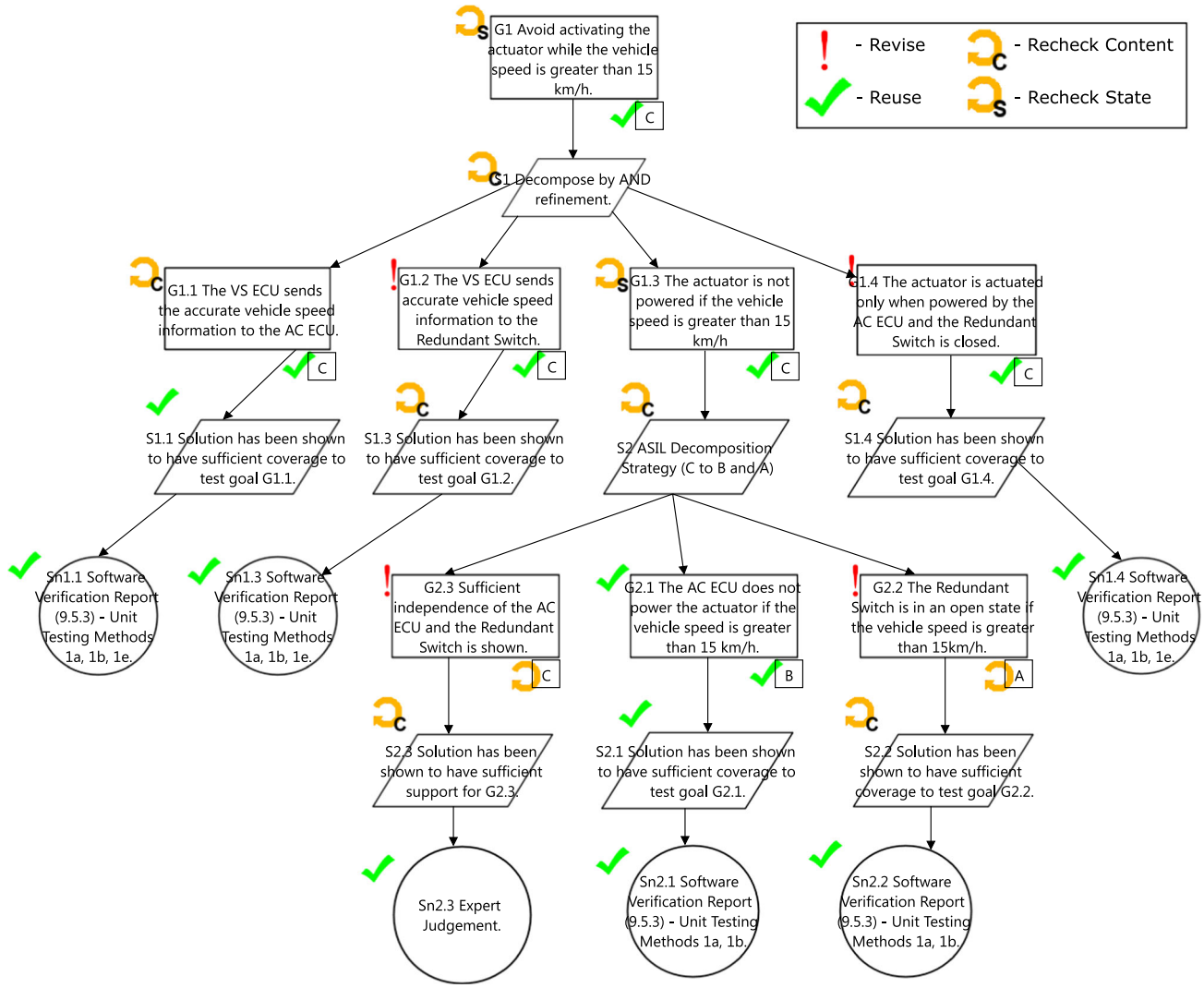

Fig. 34 Result of CIA workflow: annotated safety case for power sliding door due to removing the redundant switch [15]

26262; thus, the ASILs of the corresponding goals must also be revised.

Conceptually, the CIA workflow consists of the following steps:

1. Perform model slicing on the system megamodel in order to identify which other model elements are impacted by the changes. This is a heterogeneous operation since it uses different slicers for different types of models; its own workflow is discussed in Sect. 4.6.
2. Trace the modified system elements across the traceability relationship to identify the safety case elements requiring *revision* (i.e., those with content that definitely need to be modified given the system changes).
3. Trace the impacted system elements from step (1) across the traceability relationship to identify the safety case elements that must be rechecked for *content validity* (i.e., the textual content of the element needs to be rechecked manually in order to validate that it still holds given the changes).
4. Perform safety case model slicing[6] on results of step (2) in order to get the initial set of safety case elements that needed to be rechecked for *state validity* (i.e., the state of the element (e.g., a goal or solution) needs to be rechecked as it may no longer be supported by underlying sub-goals or evidence).
5. Perform safety case model slicing on the results of steps (3) and (4) in order to identify the complete set of safety case elements that must be rechecked for *state validity*.
6. Perform safety case model slicing on the result of step (2) and merge with the result of step (3) in order to identify the complete set of safety case elements that must be rechecked for *content validity*.

[6] See [15] for the set of rules used for safety case slicing.

7. Annotate the safety case. The results of steps (2), (5) and (6) are marked for revision, content recheck and state recheck, respectively.

8.3 CIA workflow implemented with collection operators

Figure 35 shows the MMINT implementation of the CIA workflow using the collection operators. It accepts as input two megamodels and two megarels, namely the system megamodel (`SysMega`), the safety case and its traceability relationship to `SysMega` (`SafetyMega`), the set of modified system elements (`SysMod`) and the set of deleted system elements (`SysDel`). `SysMod` and `SysDel` are sub-megamodels (defined as in Sect. 4.6) of `SysMega`—i.e., they consist of submodels of models in `SysMega` where a submodel is expressed as a unary relationship. The workflow outputs an annotated copy of the input safety case (`AnnotatedSC`), indicating for each safety case element whether it can be reused, must be revised or must be rechecked for state or content validity.

The implementation of the CIA workflow consists of 13 megamodel operations, of which 11 utilize **map** to operate on the models and relationships within the input megamodels and megarels. The numbered groupings in Fig. 35 show the parts of the implementation corresponding to the seven steps of the workflow described in Sect. 8.2:

1. Apply megamodel slice $\texttt{Slice}_{\texttt{MMINTA}}$ (See Sect. 4.6) on the modified `SysMod` and deleted `SysDel` system elements to identify the sub-megamodel `C1dm` of system elements that may be impacted by the change.
2. Apply **map**[`ModelRelPropagation`] to propagate the deleted system elements `SysDel` to the safety case across the traceability relationship.[7] `ModelRelPropagation` (See the signature in Fig. 36b) propagates the elements of submodel `s1` across relationship `r` to produce submodel `s2`. Thus, the result of **map**[`ModelRelPropagate`] here is a sub-megamodel of `SafetyMega` containing a submodel of the safety case obtained from each submodel in `SysDel`. These submodels are then merged into a single submodel of the safety case using **map**[`ModelRelMerge`] (See the signature in Fig. 36c) to produce `C2Revise`.
3. Apply **map**[`ModelRelPropagation`] to propagate submegamodel `C1dm` to the safety case. This step is similar to (2), and the result, `C2Content1`, is a sub-megamodel of `SafetyMega` containing a single submodel of the safety case representing the initial set of safety case elements to be rechecked for content validity.
4. Slice the safety case elements in `C2Revise` using **map** [`GSNSliceRevise2State`]. `GSNSliceRevise2State` (see Fig. 36d) is a safety case slicing transformation that takes a submodel of the safety case as input containing elements that must be revised and expands it to the submodel containing all dependent elements that must be rechecked for state validity. Here, it is used to expand `C2Revise` to `C2State`.
5. Slice `C2State` and `C2Content1` using **map** [`GSNSliceRecheck`] to produce `C3State`. `GSNSliceRecheck` (See Fig. 36e) is another safety case slicing transformation that takes a submodel of the safety case as input containing elements that needed to be rechecked and expands it to the submodel containing all dependent elements that must be rechecked for state validity.
6. Apply **map**[`GSNSliceRevise2Content`] on `C2Revise` and merge the results with `C2Content1` to produce `C3Content`. Like `GSNSliceRevise2State`, transformation `GSNSliceRevise2Content` (see Fig. 36f) is a safety case slicer that takes a submodel of revised elements as input, but it expands this to the set of dependent elements that need to be rechecked for *content* validity rather than *state* validity.
7. Produce the annotated safety case `AnnotatedSC` by applying `GSNAnnotate` on `C2Revise`, `C3Content` and `C3State`. `GSNAnnotate` (see Fig. 36g) creates a copy of the original safety case with each element annotated according to its presence in each of the three megamodel relations. For example, if an element is present in `C2Revise`, it is marked for revision, etc.

8.4 Discussion

We make several observations from the exercise of implementing the CIA workflow.

We first note that the collection operators were *sufficient* to implement the workflow despite its complexity. Furthermore, the use of collection operators allowed a near direct translation of the conceptual workflow in Sect. 8.2 to the implementation workflow (apart from the need for using the `ModelRelMerge` helper transformation discussed below). Interestingly, **map** was the only collection operator needed. Its usefulness likely follows from the fact that it implements a basic iteration control flow construct over megamodels—a (stateless) *for-loop*. Both **reduce** and **filter** are also looping constructs, but their more specialized semantics are applicable only in special situations that did not arise in the CIA workflow.

A noticeable characteristic of the workflow is the repeated use of **map**[`ModelRelMerge`]. `ModelRelMerge` merges two submodels of the same model into a single submodel. The CIA workflow repeatedly generates information from mul-

[7] The transformation `SliceCriterionDecorate` (See the signature in Fig. 36a) is first used to add metadata to the deleted elements to support tooling for queries over the results of CIA. We omit further discussion of this because it is outside the scope of the conceptual workflow.

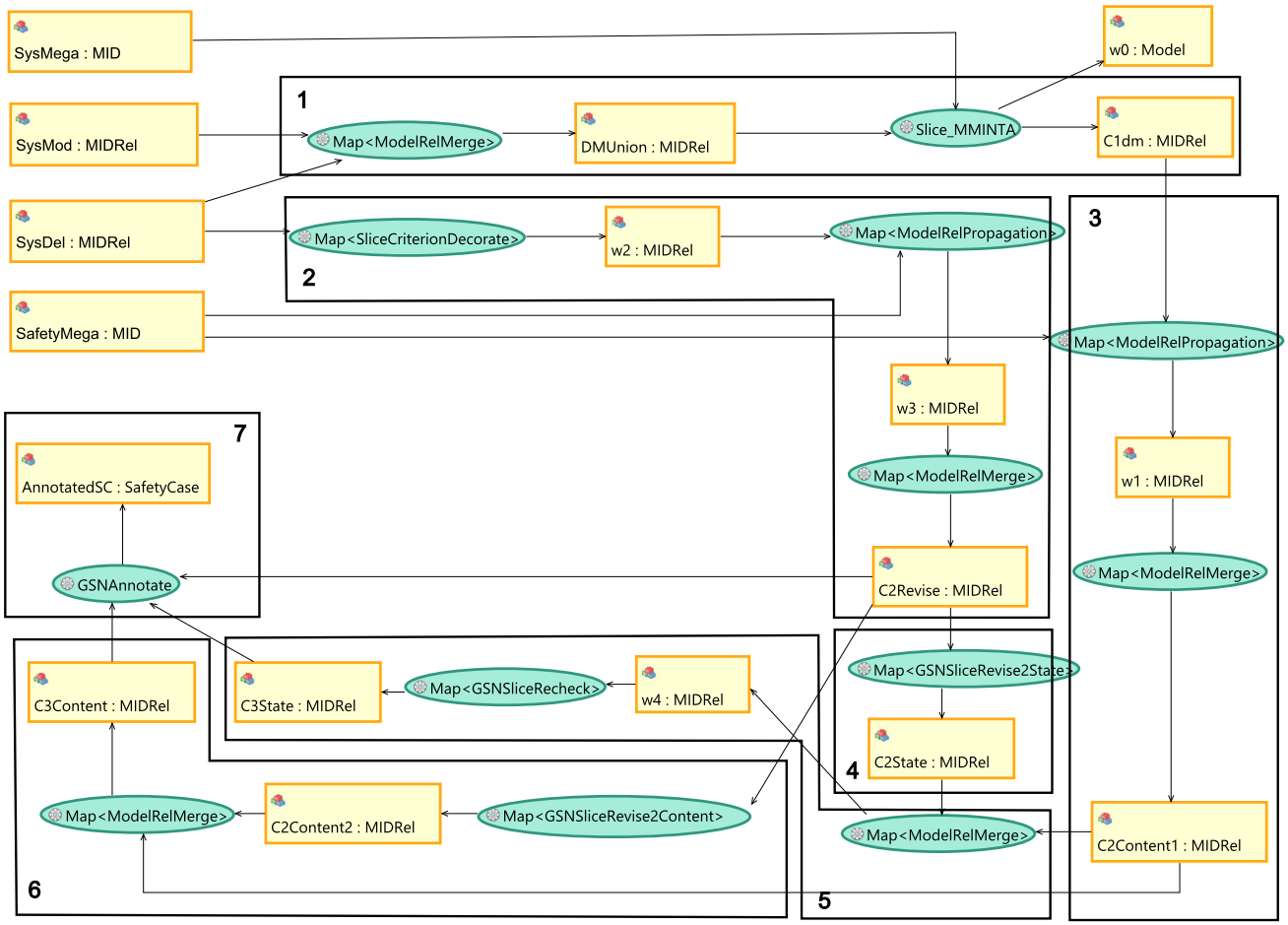

Fig. 35 MMINT workflow for assessing impact of system changes on a GSN safety case

tiple sources and applies it to the same model (e.g., the safety case). This information needs to be merged. Steps (1), (5) and (6) use **map**[`ModelRelMerge`] with two input arguments because this operation merges the content of two submodels, each of which are in different megamodels. In contrast, steps (2) and (3) use **map**[`ModelRelMerge`] with a single input argument (i.e., taking both arguments of `ModelRelMerge` from the same input megamodel) since the output of **map**[`ModelRelPropagate`] produces one megamodel containing multiple submodels of the safety case. These cases show the importance of the **map** feature that allows the input arguments of the mapped transformation to be allocated to input megamodels in different ways.

A drawback of using collection operators is that they may obscure the design intent the workflow. For example, it is impossible to deduce that `C2Revise` should only contain a single safety case submodel without prior knowledge of the semantics of `GSNSliceRevise2State`. Another problem is that staying at the megamodel level may limit the possibility of performing static analyses on the workflow since the models are "buried" in megamodels. Unless operators such as `GSNAnnotate` are used to convert megamodels into models which can then be analyzed, the onus is completely on the user to detect and diagnose unexpected results.

Finally, the fact that in Fig. 35 we had to parse the workflow into seven steps by superimposing numbered boxes onto it seems to suggest that some hierarchical decomposition mechanism is needed in *MMINT* to help manage complexity in large workflows. However, this issue is orthogonal to the functionality of collection operators since it would apply to any model management workflow.

9 Related work

Megamodel management Many model management approaches have been proposed. For example, Rondo [34] represents models as directed labeled graphs and supports traditional model management operations (e.g., *match* and *merge*) that work directly on models but not on the megamodels containing them. Maudeling[8] offers advanced query

[8] Maudeling: http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Maudeling.

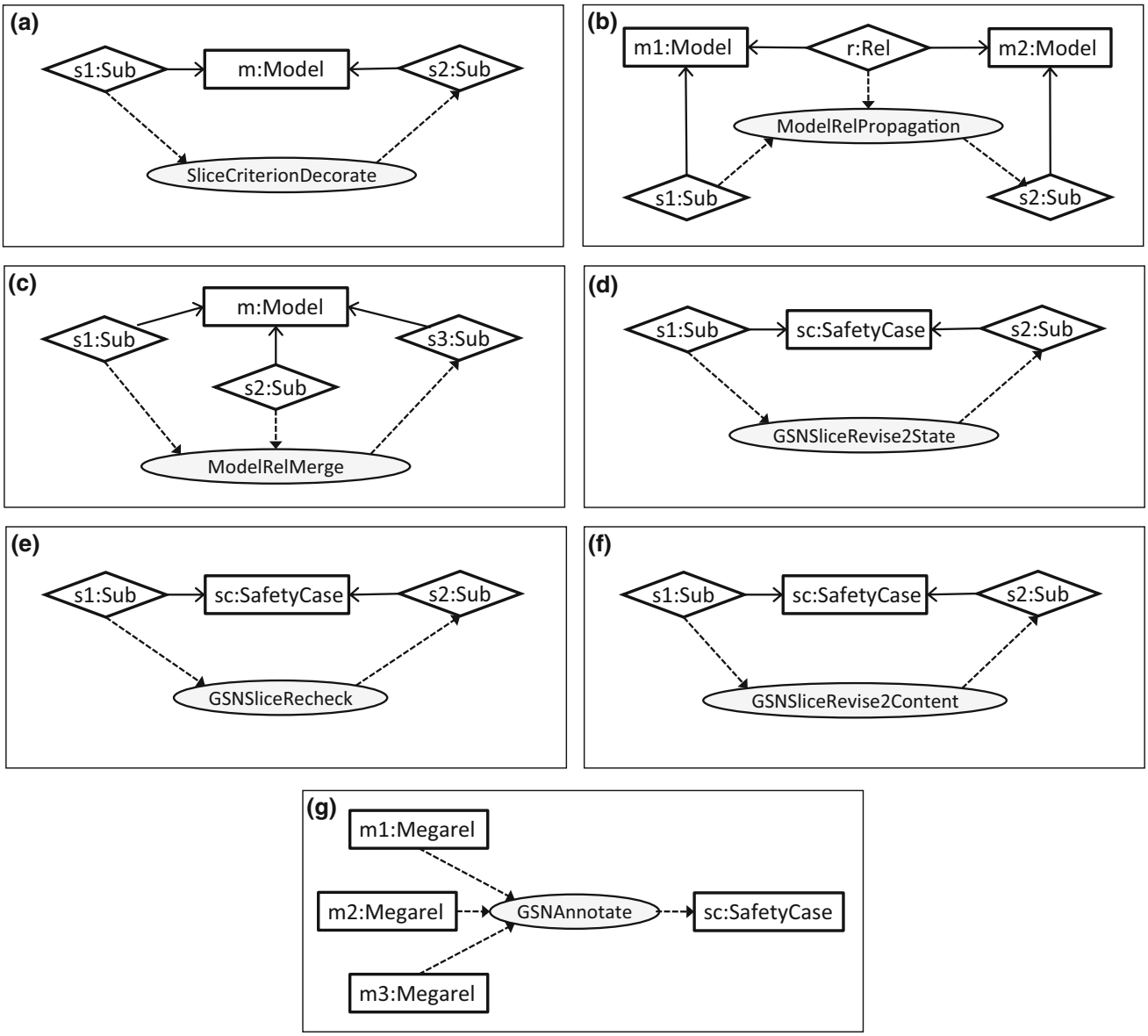

Fig. 36 Signatures of operators required in the change impact assessment workflow: **a** `SliceCriterionDecorate`; **b** `ModelRelPropagation`; **c** `ModelRelMerge`; **d** `GSNSliceRevise2State`; **e** `GSNSliceRecheck`; **f** `GSNSliceRevise2Content`; and **g** `GSNAnnotate`

services; however, these are on the modeling artifacts themselves and not on megamodels. Epsilon [27] provides a set of domain specific languages for specific model management operations such as *match* and *merge*; however, no special support is provided for megamodels.

The Atlas Model Management Architecture (AMMA) [2] has a component AM3 [21] for expressing megamodels and an OCL-based scripting language MoScript for general model management scripts including limited support for megamodel manipulation. Specifically, MoScript [23] provides support for *map* by using the OCL ApplyTo and Collect operations and support for *filter* using the OCL Select operation; however, these versions of *map* and *reduce* are more limited than what we propose because MoScript does not treat relationships between models as first class citizens and the support for map and reduce is limited to sets of models rather than graph-like collections in megamodels. In addition, MoScript does not provide support for the *reduce* operation. Other MDE workflow languages such UniTI [45], and TraCo [19] can also be used to manipulate megamodels. These languages are more general purpose and do not have the megamodel-specific abstractions that we focus on. More recently, the megamodeling language MegaL [14] has been proposed for editing megamodels graphically. But the focus of this work is on the megamodeling language itself rather than on its use for model management. We see MDE

workflow languages such as MoScript, UniTI and TraCo as complementary to our approach and believe they can benefit from incorporating our megamodel manipulation operations into the language.

Model search engines such as MOOGLE [33] and IncQuery [44] perform queries of model contents. Our **filter** operation does not limit which languages or engines can be used for defining model and relationship properties. Thus, model search engines are complementary to our approach.

Collection operators Graph-based languages and frameworks that provide collection-based operations on graphs have been proposed. The *map* and *fold* (i.e., reduce) algorithms in [12] generalize the classic list-based versions of these to graphs, but the assumptions made by these algorithms make them inapplicable to the megamodel case. Specifically, the *map* algorithm does not allow for a "graph" of input arguments to the transformation as **map** does with transformation input signatures, and the *fold* algorithm only aggregates values on nodes and edges rather than collapsing the graph structure itself as **reduce** does. The *MapReduce* approaches of Google and others [6] are intended for the efficient processing of big data; yet, these operate differently from the *map* and *reduce* functions found in many programming languages [29].

Model heterogeneity Several approaches [8,10,11,28,40,43] for addressing the problem of consistency checking and management among heterogeneous models have been proposed. The work in [8,9,28] treats the collection of heterogeneous models (multimodels) as a graph and considers a categorical approach to formalize the overlaps between them. The work in [11] addresses the same problem by organizing the different partial models as a network of related models, which provides a global view of the heterogeneous system through a correspondence model. As changes occur and inconsistency emerges, a semi-automatic process based on the correspondence model allows detecting changes, calculating their impacts, and proposing modifications to maintain the consistency among them. The thesis in [40] focuses on describing, modeling and verifying inter-model constraints and relationships between preexisting heterogeneous models used in a system engineering process. Finally, in [43], a formalization of software development build processes is used to enable consistency management between collections of heterogeneous models represented using a megamodel. While all these approaches deal with heterogeneous model management, they focus on the specific subproblem of consistency management. In contrast, our collection operators are generic building blocks for implementing arbitrary heterogeneous model management tasks.

Model typing and polymorphism Model typing plays an important role in our work. It has been studied from different perspectives. A formal type system cGMM [46] has been integrated with AM3 megamodels discussed above to provide support for type checking and type inference. However, there is no discussion of polymorphism in that work.

To the best of our knowledge, there are no works addressing ad hoc polymorphism for models; however, various approaches to universal polymorphism have been proposed as a way of supporting *transformation reuse*. Model subtyping techniques (e.g., [18,41]) establish rules for subtyping relationships between metamodels that can be applied automatically to judge whether a transformation expressed over one metamodel can be executed over another metamodel. Model concepts and related techniques [32,35] require that developers wanting to reuse a model transformation provide an explicit, and potentially complex, mapping between the two metamodels. This is more powerful than subtyping because the mapping can allow for richer relationships between metamodels. Other approaches [31,47] use constraints to express minimal typing requirements on the input/output model types of a transformation. Other input/output types that satisfy the requirements can also be safely used with the transformation.

In our work, the type compatibility relation is the abstraction that allows universal polymorphism (via inclusion polymorphism). Thus, any of the subtyping approaches in the literature can be used as the type compatibility relationship. Currently, we do not support the additional metadata (e.g., mappings) required by the more powerful universal polymorphism approaches described above and consider this future work.

10 Conclusion and future work

In this paper, we have proposed three new megamodel collection operators: **map**, **reduce** and **filter**. These operators are inspired by similar collection manipulation operators found in many programming languages, but are adapted to address the special characteristics of megamodels and MDE environments. Specifically, the operators treat model relationships as first class entities and address the graph-like structure of megamodels and of the signatures for model transformations. To address the application of these operators to heterogeneous megamodels, they are designed to work with polymorphic transformations and properties. Both inclusion and ad hoc polymorphism are supported.

Our future work will explore several issues. First, we plan to extend our operators to take hierarchical structure of megamodels into account. The current operators are shallow—i.e., they do not penetrate into referenced megamodels; however, complexities arise with deep versions. For example, in a deep version of **map** when there is more than one input megamodel and they have different hierarchical structures, it is not clear what hierarchical structure the output should have. Second, since the combinatorial nature of **map**

limits its scalability, we intend to investigate ways to mitigate this problem. For example, it may be possible to adapt the highly parallelizable *MapReduce* framework used in big data scenarios. We want to add optimizations to make **reduce** more scalable as well. Third, we plan to extend our approach to address a more general formulation of polymorphism where the polymorphic variants can extend the signature with new elements. That is, we will relax the requirement that the alignment mapping be bijective and require it to only be injective. In addition, we will investigate supporting the richer forms of universal polymorphism that have been presented in the literature. Fourth, we plan to experimentally evaluate how more complex signatures affect the overhead time of the collection operators. Finally, we want to further validate the usefulness of the collection operators by doing more case studies both in software development and beyond. Although in this paper we have taken our motivation from the software domain, there is nothing about the collection operators that limits their use to software models. We have hinted at this in Sect. 8 where we included GSN models in the case study. Our overall objective in all these investigations is to produce a set of scalable megamodel manipulation operators that are needed in heterogeneous model management scenarios.

Acknowledgements This work is funded by NSERC in collaboration with General Motors.

References

1. Bernstein, P.A.: Applying model management to classical meta data problems. In: Proceedings of CIDR'03, vol. 2003, pp. 209–220 (2003)
2. Bézivin, J., Jouault, F., Touzet, D.: An introduction to the atlas model management architecture. Tech. Rep. 05.01, Labratoire d'Informatique de Nantes-Atlantique (2005)
3. Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proceedings of OOPSLA/GPCE Workshops (2004)
4. Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A manifesto for model merging. In: Proceedings of GAMMA at ICSE'06, pp. 5–12 (2006)
5. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. (CSUR) **17**(4), 471–523 (1985)
6. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
7. Diskin, Z., Kokaly, S., Maibaum, T.: Mapping-aware megamodeling: design patterns and laws. In: Proceedings of SLE'13, pp. 322–343 (2013)
8. Diskin, Z., König, H.: Incremental consistency checking of heterogeneous multimodels. In: Proceedings of STAF'16 Workshops, pp. 274–288. Springer (2016)
9. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In: Proceedings of MDI@MODELS'10, pp. 42–51. ACM (2010)
10. El Hadji Bassirou Toure, B., Fall, I., Bah, A., Camara, M.S.: Megamodel consistency management at runtime. In: Proceedings of CNRIA'17, vol. 204, p. 257. Springer (2018)
11. El Hamlaoui, M., Ebersold, S., Coulette, B., Nassar, M., Anwar, A.: Heterogeneous model matching for consistency management. In: Proceedings of RCIS'14, pp. 1–12. IEEE (2014)
12. Erwig, M.: Functional programming with graphs. ACM SIGPLAN Not. **32**(8), 52–65 (1997)
13. Favre, J.M., Lämmel, R., Varanovich, A.: Modeling the Linguistic Architecture of Software Products. Springer, Berlin (2012)
14. Favre, J.M., Lämmel, R., Varanovich, A.: Modeling the linguistic architecture of software products. In: Proceedings of MoDELS'12, pp. 151–167 (2012)
15. Fung, N.L., Kokaly, S., Di Sandro, A., Salay, R., Chechik, M.: MMINT-A: a tool for automated change impact assessment on assurance cases. In: Proceedings of ASSURE@SAFECOMP'18, pp. 60–70. Springer (2018)
16. Garey, M.R., Johnson, D.S.: Computers and intractability, vol. 29. WH freeman, New York (2002)
17. GSN: Goal Structuring Notation Working Group. GSN Community Standard Version 1. http://www.goalstructuringnotation.info/ (2011)
18. Guy, C., Combemale, B., Derrien, S., Steel, J.R., Jézéquel, J.M.: On model subtyping. In: Proceedings of ECMFA'12, pp. 400–415 (2012)
19. Heidenreich, F., Kopcsek, J., Aßmann, U.: Safe composition of transformations. J. Obj. Technol. **7**(10) (2011)
20. ISO: ISO 26262: Road Vehicles—Functional Safety. International Organization for Standardization (2011). 1st version
21. Jouault, F., Vanhooff, B., Bruneliere, H., Doux, G., Berbers, Y., Bézivin, J.: Inter-DSL coordination support by combining megamodeling and model weaving. In: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 2011–2018. ACM (2010)
22. Kelly, T., Weaver, R.: The goal structuring notation—a safety argument notation. In: Proceedings of DSN'04 (2004)
23. Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., Cabot, J.: MoScript: A DSL for querying and manipulating model repositories. In: Proceedings of SLE'12, pp. 180–200. Springer (2012)
24. Kokaly, S.: Managing assurance cases in model based software systems. Ph.D. thesis, McMaster University (2019)
25. Kokaly, S., Salay, R., Cassano, V., Maibaum, T., Chechik, M.: A model management approach for assurance case reuse due to system evolution. In: Proceedings of MODELS'16, pp. 196–206. ACM (2016)
26. Kokaly, S., Salay, R., Chechik, M., Lawford, M., Maibaum, T.: Safety case impact assessment in automotive software systems: an improved model-based approach. In: Proceedings of SAFECOMP'17, pp. 69–85. Springer (2017)
27. Kolovos, D.S., Rose, L.M., Garcia-Dominguez, A., Paige, R.F.: The Epsilon Book. Eclipse, Cairo (2015)
28. König, H., Diskin, Z.: Advanced local checking of global consistency in heterogeneous multimodeling. In: Proceedings of ECMFA'16, pp. 19–35. Springer (2016)
29. Lämmel, R.: Google's mapreduce programming model—revisited. Sci. Comput. Program. **70**(1), 1–30 (2008)
30. Lano, K., Rahimi, S.K.: Slicing of UML models. In: Proceedings of ICSOFT'10 Vol. 2, pp. 259–262 (2010)
31. de Lara, J., Di Rocco, J., Di Ruscio, D., Guerra, E., Iovino, L., Pierantonio, A., Cuadrado, J.S.: Reusing model transformations through typing requirements models. In: International Conference on Fundamental Approaches to Software Engineering, pp. 264–282. Springer, Berlin (2017)
32. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. SoSyM **12**(3), 453–474 (2013)
33. Lucrédio, D., Fortes, R.P.d.M., Whittle, J.: MOOGLE: A model search engine. In: Proceedings of MoDELS'08, pp. 296–310 (2008)

34. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: a programming platform for generic model management. In: Proceedings of SIGMOD'03, pp. 193–204. ACM (2003)
35. Rose, L., Guerra, E., de Lara, J., Etien, A., Kolovos, D., Paige, R.: Genericity for model management operations. SoSyM (2011)
36. Salay, R., Chechik, M., Easterbrook, S., Diskin, Z., McCormick, P., Nejati, S., Sabetzadeh, M., Viriyakattiyaporn, P.: An eclipse-based tool framework for software model management. In: Proceedings of Eclipse Workshop @ OOPSLA'07, pp. 55–59 (2007)
37. Salay, R., Kokaly, S., Chechik, M., Maibaum, T.: Heterogeneous megamodel slicing for model evolution. In: Proceedings of ME@MoDELS'16, pp. 50–59 (2016)
38. Salay, R., Kokaly, S., Di Sandro, A., Chechik, M.: Enriching megamodel management with collection-based operators. In: Proceedings of MoDELS'15, pp. 236–245 (2015)
39. Salay, R., Mylopoulos, J., Easterbrook, S.: Using macromodels to manage collections of related models. In: Proceedings of CaiSE'09, pp. 141–155. Springer (2009)
40. Simon-Zayas, D.: A framework for the management of heterogeneous models in systems engineering. Ph.D. thesis, ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique-Poitiers (2012)
41. Steel, J., Jézéquel, J.M.: On model typing. SoSyM **6**(4), 401–413 (2007)
42. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education, London (2008)
43. Stevens, P.: Towards sound, optimal, and flexible building from megamodels. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 301–311. ACM (2018)
44. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: an integrated development environment for live model queries. Sci. Comput. Program. **98**, 80–99 (2015)
45. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: Uniti: A unified transformation infrastructure. In: Proceedings of MODELS'07, pp. 31–45. Springer (2007)
46. Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H.: Typing artifacts in megamodeling. J. Softw. Syst. Model. **12**(1), 105–119 (2013)
47. Zschaler, S.: Towards constraint-based model types: A generalised formal foundation for model genericity. In: Proceedings of the 2nd workshop on view-based, aspect-oriented and orthographic software modelling, p. 11. ACM (2014)

Dr. Rick Salay is a researcher in software modeling with over 40 peer-reviewed papers in the area. He has conducted and led internationally recognized research in modeling on topics including safety assurance modeling, model management, model uncertainty and model transformations. He played a senior role in NECSIS, a five year 10.5 million pan-Canadian research network with industrial partners General Motors and IBM, focused on improving Model Driven Engineering (MDE) practice. Currently he plays senior roles in projects related to the safety of automated driving systems and machine learning. He regularly participates in program committees for international conferences related to software engineering, modeling and safety. Prior to his research career, he had a 15 year career in advanced software product development holding senior software design roles, most recently as chief architect at InSystems Technologies Inc. (now Oracle).

Dr. Sahar Kokaly is a Research Associate in the Department of Computer Science at the University of Toronto. She is also a part-time Researcher at General Motors, working with the R&D group on collaborative projects with academia in the areas of safety, model-based engineering and feature modeling. Sahar completed her Ph.D. in Software Engineering in 2019 from McMaster University. Prior to that, Sahar worked as a Research Engineer on the NECSIS (Network for the Engineering of Complex Software Intensive Systems) project in Canada, and as an IT specialist at IBM Canada. She has been involved in organizing numerous workshops (e.g., MiSE at ICSE, AMT and MPM at MODELS), served on MODELS conference organizing committees (MODELS 2015, 2017, 2018) and was an invited panelist at MODELS 2016. Sahar regularly acts as a program committee member on workshops, most recently MiSE 2019 and SASSUR 2019, and as a reviewer for journals including Journal of Systems and Software, Empirical Software Engineering Journal, Software & Systems Modeling Journal and IEEE Software. Sahar's main research interests are in safety assurance, model-driven engineering and improving the state-of-the art in software development in industry through automation and reuse.

Alessio Di Sandro is a Software Engineer based in Pisa, Italy. He received his degree (with honors) in Computer Engineering from the University of Pisa in 2009, working for Ericsson Research in Stockholm, Sweden, as part of his Master's thesis. He has been a researcher at CNR-ISTI in Pisa, Italy, and at the University of Toronto, Canada. His research interests include topics from Model-Driven Engineering and Visual Technologies, such as model management, model validation, automated code generation, design of graphical interfaces and web technologies.

Nick L. S. Fung is a research assistant in the Software Engineering group at the University of Toronto, wherein he also obtained a Master's degree in Computer Science. His research interests revolve around model-driven engineering, with a special focus on the tools and techniques for designing and developing safety critical systems. Particular application domains for his research are healthcare and automotive.

Marsha Chechik is Professor in the Department of Computer Science at the University of Toronto. She received her Ph.D. from the University of Maryland in 1996. Prof. Chechik's research interests are in the application of formal methods to improve the quality of software. She has authored numerous papers in formal methods, software specification and verification, computer safety and security and requirements engineering. In 2002–2003, Prof. Chechik was a visiting scientist at Lucent Technologies in Murray Hill, NY and at Imperial College, London UK, and in 2013—at Stonybrook University. She is a member of IFIP WG 2.9 on Requirements Engineering and an Associate Editor in Chief of Journal on Software and Systems Modeling. She is has been an associate editor of IEEE Transactions on Software Engineering 2003–2007, 2010–2013. She regularly serves on program committees of international conferences in the areas of software engineering and automated verification. Marsha Chechik has been Program Committee Co-Chair of the 2018 International Conference in Software Engineering (ICSE18), 2016 International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16), the 2016 Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE16), the 2014 International Conference on Automated Software Engineering (ASE'14), the 2008 International Conference on Concurrency Theory (CONCUR'08), the 2008 International Conference on Computer Science and Software Engineering (CASCON'08), and the 2009 International Conference on Formal Aspects of Software Engineering (FASE'09). She will be PC Co-Chair of ESEC/FSE'2021. She is a Member of ACM SIGSOFT and the IEEE Computer Society.