# Declarative Approach for Model Composition

Julia Rubin [1, 2] Marsha Chechik [2] Steve Easterbrook [2]

[1] IBM Haifa Research Labs
Haifa University Campus,
Haifa, 31905, Israel
mjulia@il.ibm.com

[2] Department of Computer Science
University of Toronto
Toronto, ON M5S 3G4, Canada.
{mjulia,chechik,sme}@cs.toronto.edu

## ABSTRACT

*Model-based development involves construction, integration, and maintenance of complex models. One of the key problems in model-based development is composing a set of distributed models into a single seamless model. In this paper we propose a declarative approach for model composition, which augments and strengthens existing structural and heuristic approaches. In our approach, the desired model compositions are constrained by a set of declarative properties, which drive the merge process. Only model compositions that satisfy the specified properties and, possibly, additional model composition restrictions are automatically generated and presented to the model analyst for a review and further modifications. Since our framework is iterative, properties and restrictions can be reviewed and refined as well. We illustrate our ideas by defining a proof-of-concept prototype implementation of the declarative model composition framework using the Alloy Analyzer.*

## Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design tools and techniques.

## General Terms
Design.

## Keywords
Merging, matching, properties, Alloy.

## 1. INTRODUCTION

Due to the increased complexity of software development processes, models take a central role in today's development environments. When dealing with complex systems, it is impractical to describe the whole system by a single model. Instead, analyzing and modelling a software system with separate views is a good practice to deal with complexity and maintainability. This, however, leads to the need to combine the developed views into a single, consistent and coherent model. Automatic model composition still remains one of the key problems in today's model-based development.

Model composition consists of two fundamental phases – *model matching* and *model merging* [1]. In the model matching phase (also sometimes called *model comparison*) correspondences between similar elements of the source models are identified, e.g., an *Employee* in one model can correspond to a *Worker* in another. The merging phase uses these correspondences to produce a unified version of the model, e.g., the one that includes *People* in organization, whether they are called *Workers* in one model or *Employees* in another.

Several approaches to model matching have been proposed, ranging from manual to fully automatic algorithms for detecting correspondences [3,7,8,9,11]. Most existing approaches to (semi)-automatic model matching and merging focus on structural similarities between the models. For example, matching and merging of conceptual database schemata are studied in [9]; a general framework for merging visual design diagrams is proposed in [8]; a generic framework for merging EMOF-based models is presented in [3]; an algebraic approach for merging requirements views is described in [11]; finally, a technique for matching architecture diagrams using machine learning is provided in [7]. In general, these approaches treat models as graphical artefacts while largely ignoring their semantics.

Some recent work on behavioural models has concentrated on establishing semantic relationships between models. For example, [14] proposes the use of refinement relations for merging consistent state-machine models such that their behavioural properties are preserved. An approach to matching and merging hierarchical Statechart models using heuristics for finding terminological, structural, and semantic similarities between models is presented in [10].

All the above techniques to (semi)-automatic model composition use a set of predefined structural or heuristic procedures to drive the match and the merge algorithms. The implementation of these procedures is further supported by several rule-based languages and frameworks for model compositions, such as the Epsilon Merging Language [6] or Kompose [4]. However, all these approaches lack a declarative semantic definition of what the desired model compositions should be. Without a semantically-based declarative definition of such a result, one cannot be sure that the merge algorithm computes exactly what one needs. For example, when merging two business processes due to an acquisition of one company by another, the resulting merge can satisfy a variety of objectives: maintain a fixed number of parties with which the process converses, or maybe maintain the overall process size, or maybe maintain a specific order of activities.

In this paper, we propose to augment the existing structural and heuristic match procedures by a new declarative procedure for model matching and merging. It allows the analyst to specify the properties of the desired model composition, such as "the activities of the merged process must be in the same order as of the original processes“, and uses these properties during the model composition process to produce proper results.

We also propose an iterative framework for property-based construction of model compositions. In this framework, given two source models (we also refer to them as *input models*), the analyst can specify a set of the declarative properties mentioned above. The analyst can also provide additional restrictions by specifying a set of known match relationships between elements of source models, if such information is available, or employ an existing match detection algorithm for identifying such relationships. Our framework uses all this information to construct a set of possible merges. These merges are then presented to the analyst for the evaluation and refinement. Properties and match relationships can be refined as well, which drives the next iteration of the model composition process.

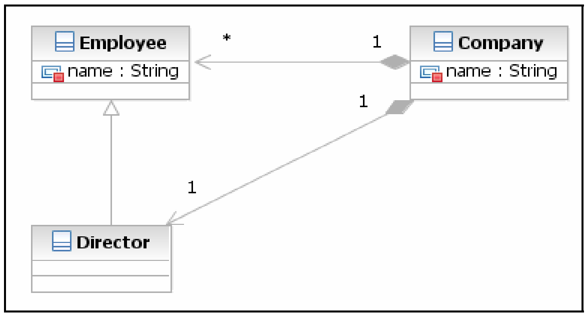
To validate our ideas, we instantiated this framework using the Alloy Analyzer [5] – a software tool which can be used to analyze specifications written in the Alloy language. The Analyzer can generate instances of model invariants; simulate the execution of operations defined as part of the model; and check user-specified properties of a model. In the paper, we describe our proof-of-concept prototype implementation and comment on our experience.

The rest of the paper is organized as follows. In Section 2, we present the working example used throughout the paper. Section 3 describes the generic property-driven model composition framework. In Section 4, we discuss a prototype implementation of the proposed framework using the Alloy Analyzer and demonstrate our prototype on the working example. Section 5 concludes the paper and presents directions for future work.

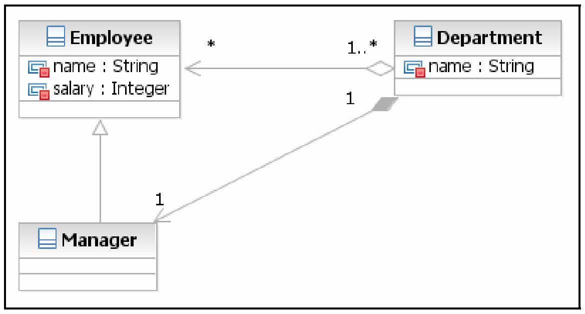
## 2. MOTIVATING EXAMPLE

Figure 1 presents two models depicting relations between entities in an organization. Both models are represented in the UML2 notation [15], with the organizational entities represented by UML2 classes. Model *A* (Figure 1(a)) includes a set of *Employees*, each of whom works for a *Company* (composite association), and one *Director* who also works for a *Company* (composite association). *Director* is a type of *Employee* (generalization relationship). Model *B* (Figure 1(b)) includes a set of *Employees*, each of whom can work for one or more *Department* (shared association). Each *Department* has a *Manager* (composite association). *Manager* is a type of *Employee* (generalization relationship).

These input models capture individual views of two different stakeholders on the organizational structure. The goal of the merge is consolidate these views to get a unified and consistent perspective. There are several different alternatives for the consolidation. Even if we assume that the *Employee* class of model *A* matches the *Employee* class of model *B* (using the name-based correspondence), there are seven merges possible, as presented in Figure 2. For example, Figure 2(a) presents a merge in which, in addition to the *Employee* classes that are combined, the *Director* class from model *A* is combined with the *Manager* class of model *B*, and the *Company* class of model *A* is combined with the *Department* class of model *B*. Figure 2(b)



**(a) Model *A***



**(b) Model *B***

**Figure 1. Input models.**

presents the case in which only the *Employee* classes are combined.

Intuitively, matches in parts (a) – (d) of Figure 2 are reasonable, whereas the remaining ones are not: *Manager* is matched with *Company,* or *Department* is matched with *Director*. However, since the semantics of the model composition is domain-specific and depends on the system being modelled, there cannot be an automatic way to decide, without any additional knowledge, which of the above seven compositions represents the option most preferred by the analyst.

The goal of our work is to assist the analyst in the construction of the desired model compositions based on semantic restrictions that describe them. Once the desired model compositions are characterized by a set of declarative properties, our framework uses these properties and constructs only the compositions that satisfy them.

Declarative model properties can be used for a variety of purposes: some properties are structural and ensure well-formedness of the model compositions; others are specific to the domain being modelled, or even to the concrete application in that domain; others ensure the correctness of the behavioural semantics of the composed model, and so forth.

In general, we differentiate between two types of properties: *meta-model level* and *model level*. Properties specific to all models in a certain domain are usually defined on a meta-model level. In the example on Figure 1, some of these are:

P1. A class cannot be owned by more than one other class (ownership is defined with respect to the composite association relationship);

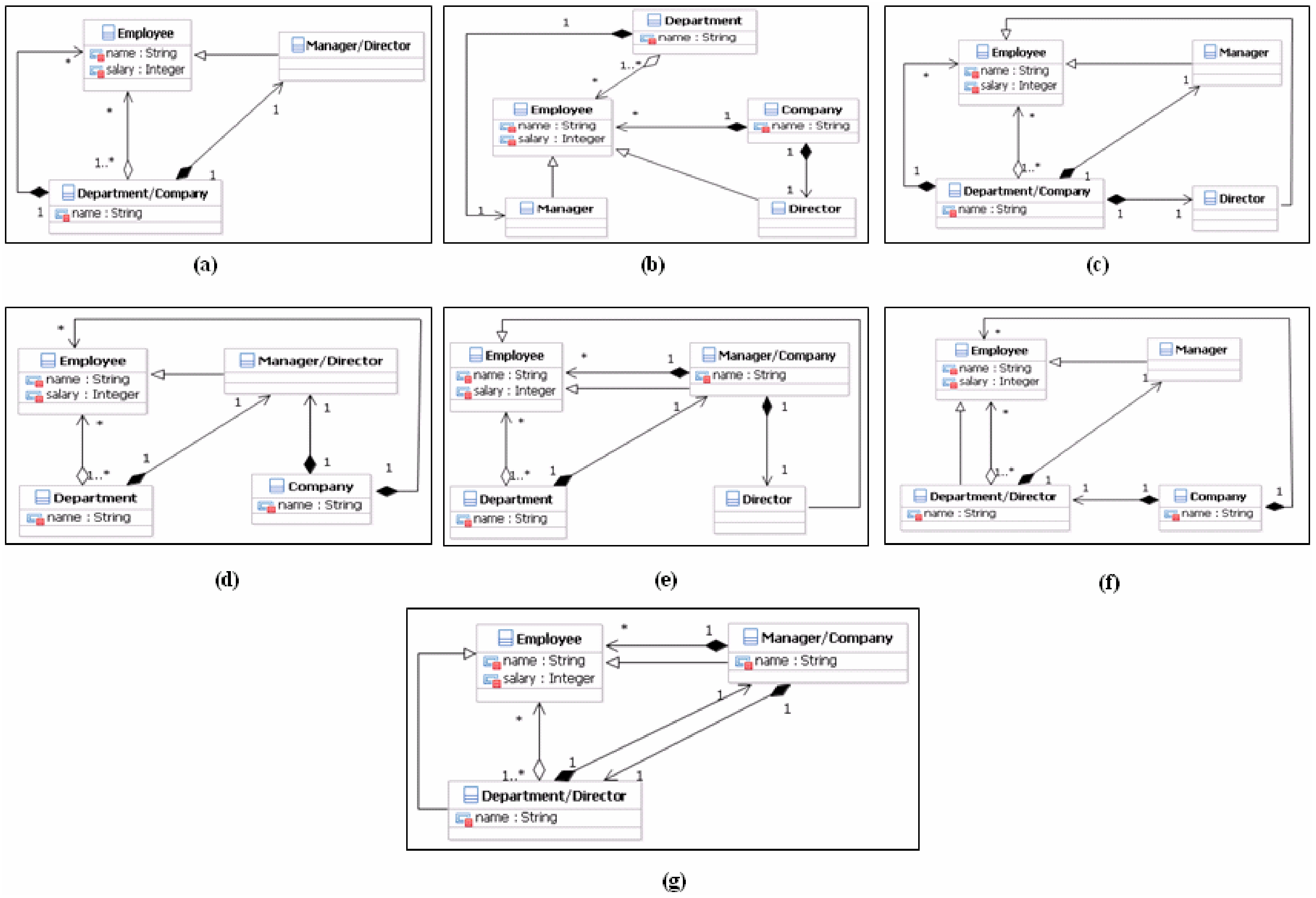


**Figure 2. Possible model compositions.**

P2. A class cannot contain an element that it specializes (containment is defined with respect to the composite or shared association relationship);

P3. Cyclic generalizations are not allowed;

P4. Cyclic ownerships are not allowed.

Properties specific to a concrete system being modelled are defined on a model level, e.g.,

P5. Each *Employee* can work for more than one *Department*;

P6. Each *Department* has one and only one *Manager*;

P7. Each *Company* has one and only one *Director*.

In addition, some properties are *intended*, while others are *accidental*. The former describe features which were deliberately introduced into the model. Properties P1-P7 are in this category. Intended properties are useful for automatic finding of matches. On the other hand, accidental properties, for example, the property

P8. Each model has exactly three classes.

are just side-effects of various design choices and should not necessarily be preserved by the composed model.

While in principle it might be possible to detect an exhaustive list of all properties of the input models, there is no automatic way to distinguish between intended and accidental properties. Therefore, in this work, we assume that the analyst explicitly specifies the list of intended properties that are to be preserved by the composition.

## 3. PROPERTY-DRIVEN FRAMEWORK FOR MODEL COMPOSITION

This section describes the property-driven framework and the process of the property-driven model composition, depicted in Figure 3.

The model composition engine takes input from both the analyst and a central meta-data repository which stores predefined meta-information about model compositions. The information includes the supported merge operation types, such as a “union” merge that produces models containing all elements of both input models; a predefined set of supported modelling domains – either general purpose, like UML, or domain-specific, like business processes or IP telephony services; and, if available, sets of domain-specific properties for each of the stored domains. The information can be added to the repository for each new merge type, new modelling domain or a family of applications in a domain. We assume that the analyst has partial knowledge about the models being composed. Given two source models, the analyst might be able to specify a set of *positive* and *negative match relationships* between elements of the input models – relationships between elements that should be combined into one in the model composition and elements that should not be combined, respectively. Such relationships

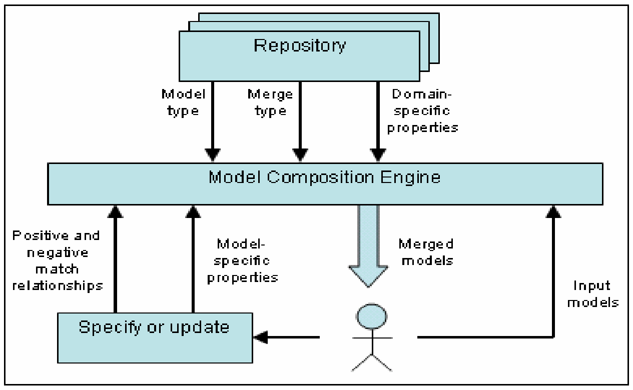


**Figure 3. Property-driven model composition process.**

can be defined by the use of structural of heuristic match detection algorithms, such as in [3] or [10], or done by hand, or skipped altogether.

In the example on Figure 1, the *Employee* class of model *A* has a positive match relationship to the *Employee* class of model *B*, which can be established by a simple name-based correspondence detection. The *Department* and the *Director* classes have a negative match relationship.

Next, the analyst defines a set of semantic properties that describe the desired model composition. Using these properties and an optional set of positive and negative match relationships, our framework constructs the set of match relationships between all elements of the input models and computes a merge by applying an appropriate merge algorithm.

Since creation of the complete list of desired composition properties is a non-trivial task, the model composition process is iterative. At the end of each iteration, possible merges are presented to the analyst for further examination, which may lead to the discovery of new properties and match relationships, or the invalidation of some of the existing ones. The analyst may then want to start a new iteration by revising the properties and executing the subsequent activities.

In our example, assume that the analyst identifies the *Employee* class of model *A* and the *Employee* class of model *B* in Figure 1 as a positive match. For this model composition configuration, which we refer to as ***MCC1***, the system produces all possible model compositions shown in Figure 2. Further, the analyst adds property P2 defined in Section 2 as an additional constraint on the merge (we refer to the obtained configuration as ***MCC2)***. The system automatically produces all possible model compositions corresponding to the cases (a) – (d) in Figure 2. Model compositions (e), (f) and (g) are eliminated because each of them violates the defined property P2: case (e) is eliminated because the *Manager/Company* class contains the *Employee* class; case (f) – because the *Department/Director* class contains the *Employee* class, case (g) – because of both of these reasons. If the analyst had used the property P1 to constrain the composition (we refer to this configuration as ***MCC3)***, cases (b), (d), (e), (f) and (g) would have been eliminated, leaving only cases (a) and (c) to be presented to the analyst. Finally, if, instead of properties P1 or P2, the analyst had defined a negative match relationship between the *Manager* and the *Company* classes, and between the *Department* and the *Director* classes (we refer to this configuration as ***MCC4)***, cases (e), (f) and (g) would have been eliminated as well, leaving only cases (a) – (d) to be presented to the analyst.

It is essential for a model composition framework to provide traceability between the merged model elements and their sources. To keep track of the origins of the elements in a merge, our framework should store proper traceability links in the merged model elements. For example, the *Department/Company* class in Figure 2(a) should be able to trace back to the *Company* class of model *A* and to the *Department* class of model *B* in Figure 1.

We also suggest capturing positive and negative feedback provided by the analyst at the end of each model composition iteration, while she examines the results suggested by the system. A simple form of such feedback could be a set of positive and negative match relationships inferred by observing the presented results. The captured information can be used by the framework in subsequent iterations.

Finally, to provide our framework with an additional flexibility, we suggest to also present to the analyst the results that do not satisfy all the defined properties, but are “close enough”. The analyst can then inspect these results and create a desired model composition by applying simple manual modifications on the compositions created automatically. Towards this end, we suggest assigning weights to the properties of the model compositions and defining some threshold on the summary weight of the properties satisfied by the resulting model composition. Only those compositions that pass the threshold are presented to the analyst. Of course, the threshold can always be set high enough, so that any result that does not satisfy all the defined properties will be eliminated.

## 4. PROPERTY-BASED COMPOSITION PROTOTYPE USING ALLOY

In order to demonstrate the viability of the ideas presented in Section 3, we implemented a proof-of-concept prototype of the declarative model composition framework using the Alloy Analyzer [5]. Alloy Analyzer is a tool developed for analyzing models written in the Alloy language – a structural modelling language based on first-order logic. The tool can generate instances of invariants, simulate the execution of operations, and check user-specified properties of a model.

Figure 4 presents an overview of our prototype, as an instance of the generic model composition framework in Figure 3. Several translators are used to express various input artefacts in the Alloy language. *Translate1* specifies an input meta-model in Alloy. This meta-model is then used by *Translate2* and *Translate6* to create a meta-model of possible input compositions (merge meta-model) and to encode the input models. The merge meta-model is used by *Translate4*, *Translate3* and *Translate5* to specify a set of positive and negative match relationships, and a set of domain-specific and model-specific composition properties. Currently all of the above translations are done manually, but they could be automated in the future.

Next, we use the Alloy Analyzer to automatically produce the described model compositions. The results generated by the Alloy Analyzer are processed and presented back to the analyst.

The following sections describe these steps in more details and illustrate them on a working example.

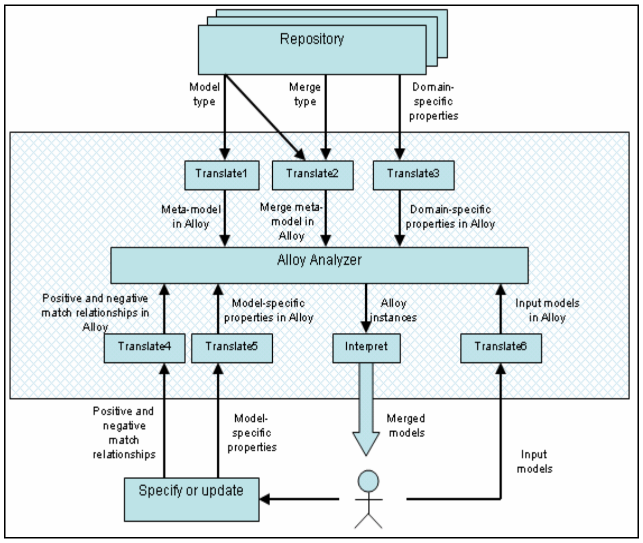

Repository
Model type
Merge type
Domain-specific properties
Translate1
Translate2
Translate3
Meta-model in Alloy
Merge meta-model in Alloy
Domain-specific properties in Alloy
Alloy Analyzer
Positive and negative match relationships in Alloy
Model-specific properties in Alloy
Alloy instances
Input models in Alloy
Translate4
Translate5
Interpret
Translate6
Positive and negative match relationships
Model-specific properties
Merged models
Input models
Specify or update

**Figure 4**. **Model composition process in Alloy.**

## 4.1 Defining Input

Figure 5 presents a simple meta-model for class diagrams in Figure 1, defined in Alloy. It defines an *Element*, which has two sets of properties – those that are associated with it using composite associations and using shared associations. An *Element* can specialize at most one other element. In this meta-model, multiple specializations are not allowed. In addition, the meta-model defines two generic elements – *LeftElement* and *RightElenent*, which are used as a base for all elements of the first and the second input model, respectively. Both *LeftElement* and *RightElement* extend the generic *Element* type.

The meta-model in Figure 5 does not represent the complete UML2 class diagram meta-model, but it is expressive enough to illustrate our ideas. In the future, we plan to explore using and extending the UML2Alloy project [1] to define and process general UML2 models.

Now we can encode input models in Figure 1 as instances of the above defined meta-model. Figure 6 presents two fragments taken from the definition of the input model in Figure 1(a).[1] The fragment in Figure 6(a) defines input model classes *Employee*, *Company* and *Director*. Further, it defines the internals of each input model class via an instantiation of the meta-model *Element*. The *Company* class, for example, does not specialize any other input model classes, and thus does not have any generalization relations. It does not have any shared association relations either. It does have a composite association relation to a set of *Employees* and a composite association relation to exactly one *Director*.

This fragment specifies the multiplicity of only one end of association relations. For example, it says that *Company* has exactly one *Director*. We use an additional Alloy fact to capture the multiplicity of the other relation's end. The fragment shown in Figure 6(b) further specifies the model and states that each *Director* works for exactly one *Company*.

[1] The complete definition is available at http://www.cs.toronto.edu/~mjulia/DMCinAlloy/

```
module metaModel

abstract sig Element {
    compositeAssociations: set Element,
    sharedAssociations: set Element,
    generalization: lone Element
}

abstract sig LeftElement extends Element {
}

abstract sig RightElement extends Element {
}
```

**Figure 5**. **Expressing a simple class meta-model.**

```
module leftModel
open metaModel

sig Employee extends LeftElement { }
{
    no compositeAssociations
    no sharedAssociations
    no generalization
}

sig Director extends LeftElement { }
{
    generalization in Employee
    one e: Employee | e in generalization
    no compositeAssociations
    no sharedAssociations
}

sig Company extends LeftElement { }
{
    compositeAssociations in Employee + Director
    one d: Director | d in compositeAssociations
    no sharedAssociations
    no generalization
 }
```

**(a) Input model classes**

```
fact eachDirectorWorksForOneCompany {
    all d: Director | some c:Company | d in c.compositeAssociations
    all c1, c2: Company, d: Director |
        c1 != c2 => (d in c1.compositeAssociations =>
                                d not in c2.compositeAssociations)
}
```

**(b) Multiplicity definition**

**Figure 6**. **Definition of an input model.**

## 4.2 Defining Merge Meta-Model

Using the input meta-model, we construct the meta-model of possible input compositions. Figure 7 presents a fragment of the composition model, each element of which is essentially a merge of two matching input model elements. A composition model element traces to a *LeftElement* and a *RightElement* which it combines – elements of the first and the second input models, respectively. We also allow the composition model to contain elements that do not have correspondences in the other model, but rather represent elements of only one of the input models. This is supported via composing left or right elements with *None*. In addition, the first fact in Figure 7 states that each element of an input model should be represented in the composition model; thus, the composition model is a "union" of two input models. The second fact states that all relationships between *CombinedElements* are "unions" of the corresponding relationships between the original elements.

```
sig None { }

sig CombinedElement extends Element{
            left: one LeftElement + None,
            right: one RightElement + None
}
{
            generalization in CombinedElement
            compositeAssociations in CombinedElement
            sharedAssociations in CombinedElement
            not (left in None && right in None)
}


fact combinationsAreValid {
            //each original element should belong to one combined element
            all l: LeftElement | one ce: CombinedElement | l in ce.left
            all r: RightElement | one ce: CombinedElement | r in ce.right
}

fact compositeRelations {
    all ce1, ce2: CombinedElement |
       //all generalizations go to combined element which represents
       //left.generalization and right.generalization
       (ce2 in ce1.generalization <=>
           (ce2.left in ce1.left.generalization || ce2.right in ce1.right.generalization)) &&
       (ce2 in ce1.compositeAssociations <=>
                         (ce2.left in ce1.left.compositeAssociations ||
                          ce2.right in ce1.right.compositeAssociations) ) &&
       (ce2 in ce1.sharedAssociations <=>
                         (ce2.left in ce1.left.sharedAssociations ||
                          ce2.right in ce1.right.sharedAssociations))
}
```

**Figure 7. Merge meta-model.**

At this stage, if we ask the Alloy Analyzer to produce instances of the meta-model we just defined, we get arbitrary merges which match any element of the first input model with any element of the second. So, the next step is to specify restrictions on the model compositions.

## 4.3 Restricting Model Compositions

Restricting the model composition using the specified merge meta-model is quite straightforward. Figure 8(a) and Figure 8(b), for example, specify in Alloy the model composition configuration ***MCC2,*** defined in Section 3. The former defines a positive match relationship between the *Employee* classes of models *A* and *B*. The later presents a definition of a declarative property P2 from Section 2. Figure 8(c) specifies two negative match relationships (between the *Manager* and the *Company* classes, and between the *Department* and the *Director* classes). This, together with the positive match relationship shown in Figure 8(a), represents the model composition configuration ***MCC4*** from Section 3.

## 4.4 Analyzing the Results

We ran the Alloy Analyzer on various model composition configurations and analyzed the obtained results. Alloy model which corresponds to the ***MCC2*** configuration produces four non-equivalent model composition instances. As expected, all composition instances satisfy the defined properties and correspond to one of the possible model compositions in Figure 2 – either one of the cases (a) – (d).

An example of a produced composition instance is shown in Figure 9(a). Each *CombinedElement* traces to the input model elements which it represents: *CombinedElement1* represents the *Company* class; *CombinedElement3* represents the *Department* class; *CombinedElement2* represents the *Manager/Director* class; *CombinedElement0* and *CombinedElement2* both represent the *Employee* class.

```
fact EmployeeToEmployee {
    some ce: CombinedElement |
      ce.left in Employee && ce.right in Employee
}
```

**(a) Positive match**

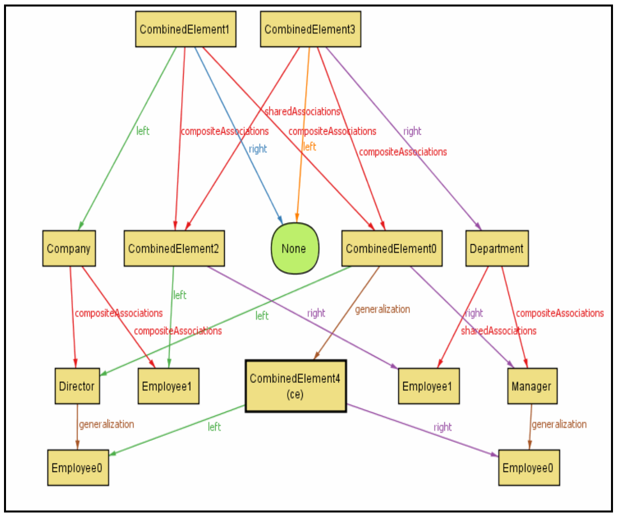
```
fact canNotOweGeneralizedElements {
  all ce1, ce2: CombinedElement | (ce2 in ce1.generalization =>
       (no ce3: CombinedElement | sameCombinations[ce2, ce3] &&
         (ce3 in ce1.compositeAssociations || ce3 in ce1.sharedAssociations)))
}
```

**(b) Declarative property**

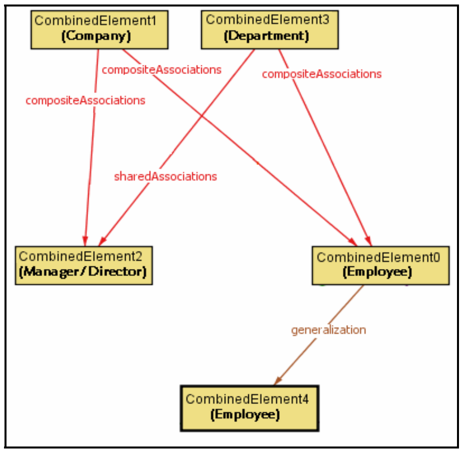
```
fact noManagerToCompany {
    no ce: CombinedElement |
        ce.left in Company && ce.right in Manager
}

fact noDepartmentToDirector {
    no ce: CombinedElement |
        ce.left in Director && ce.right in Department
}
```

**(c) Negative match**

**Figure 8. Composition model restrictions.**



**(a)**



**(b)**

**Figure 9. Generated model compositions.**

Figure 9(b) shows the same composition model instance in which we filtered out the input model elements and augmented *CombinedElements* with the names of the source elements they represent. This composition instance corresponds to the model composition in Figure 2(d). Multiplicity of association ends is also preserved by the generated instance model, and can be obtained by querying it. However, the process of deducing the composition model from its instance, produced by Alloy, is currently done manually. Automating it is our next step.

Model composition configuration ***MCC4***, similarly to ***MCC2,*** produces four non-equivalent configurations, corresponding to cases (a) – (d) in Figure 2 as well. ***MCC1*** produces all cases in Figure 2, as expected.

The performance of the Alloy Analyzer is tightly coupled with the specified number of model element instances. For example, for the ***MCC2*** configuration, we restricted the scope of the Alloy Analyzer model to exactly one *Company* and one *Department*, exactly three *Employees* from the first input model and three *Employees* of the second input model. A simple calculation shows that the minimal total number of elements required to accommodate all possible model compositions is 18. Generation of results then took around 4 seconds on a 1.59GHz, 2.00GB of RAM machine. The same configuration with the total number of up to 30 model elements took Alloy around 35 seconds. A configuration with exactly two *Companies*, two *Departments*, exactly six *Employees* in each model and a total number of up to 36 model elements took Alloy around 56 seconds.

## 5. DISCUSSION AND FUTURE WORK

In this paper, we sketched a property-based model composition framework. The input to our framework consists of two models being merged, a set of positive and negative matches between input model elements, and a set of declarative properties that constrain and drive model compositions. The output is an automatically generated set of possible model compositions, each of which satisfies the properties and restrictions defined by the analyst. These model compositions can be reviewed and further modified by the analyst. Properties and restrictions can be reviewed and refined as well. We have also presented a prototype implementation developed using Alloy.

The composition approach described in this paper currently handles only homogeneous models, those that share the same meta-model. It would be interesting to extend this approach to handle heterogeneous input models as well. Also, we are currently dealing only with one-to-one match relationships between input model elements. We plan to investigate the extension of our approach for handling many-to-many match relationships. We also plan to extend our framework to provide a better support for the iterative match and merge process, and captures positive and negative results of previous iterations.

An additional issue that we would like to investigate involves providing the analyst with an ability to explicitly specify how elements of different input models relate to each other. In our example, the analyst could define a containment relation between the *Company* class of one input model and the *Department* class of the other. This relation could restrict a set of possible matches and produce results that are closer to the analyst's intention.

Our current Alloy implementation also has several limitations. First, we still need to automate deduction of the composition model from the generated composition instance. We also need to refine and extend the process of converting input models into Alloy. We plan to evaluate using UML2Alloy for that purpose.

In addition, it is not clear yet how to define the minimal sufficient scope required for the Alloy Analyzer to find all the desired results. Since the Alloy Analyzer performs model-finding over a finite, user-defined scope (i.e., the number of instances which Alloy manipulates), it is *incomplete* by design, meaning that it can miss results that are out of the defined scope. Automatic deduction of the minimal required scope is still a research challenge.

We also want to look beyond Alloy as an implementation engine for our property-based composition framework.

## 6. REFERENCES

[1] K. Anastasakis, B. Bordbar, G. Georg and I. Ray. "UML2Alloy: A Challenging Model Transformation". In *Proceedings of MoDELS'07,* 2007.

[2] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. "A Manifesto for Model Merging". In *Proceedings of Wkshp. on Global Integrated Model Mgmt.* (GAMMA'06), 2006.

[3] F. Fleurey, B. Baudry, R. France and S. Ghosh. "A Generic Approach for Automatic Model Composition". In *Proceedings of 11th Int'l Workshop on Aspect-Oriented Modeling (AOM@MoDELS'07)*, 2007.

[4] F. Fleurey, R. Reddy, R. France, B. Baudry and S.Ghosh. "Kompose: a Generic Model Composition Tool". http://www.kermeta.org/kompose/

[5] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press. Cambridge, MA. March 2006.

[6] D. S. Kolovos, R. F. Paige and F. A.C. Polack. "Merging Models with the Epsilon Merging Language (EML)". In *Proceedings of MoDELS'06*, volume 4199 of LNCS, pp. 215-229, 2006.

[7] D. Mandelin, D. Kimelman, and D. Yellin. "A Bayesian Approach to Diagram Matching with Application to Architectural Models". In *Proceedings of ICSE'06*, pp. 222–231, 2006.

[8] A. Mehra, J. Grundy, and J. Hosking. "A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design". In *Proceedings of ASE'05*, pp. 204–213, 2005.

[9] S. Melnik. *Generic Model Management: Concepts And Algorithms*. Volume 2967 of LNCS. Springer, 2004.

[10] S. Nejati, M. Sabetzadeh, M.Chechik, S. Easterbrook and P. Zave. "Matching and Merging of Statechart Specifications". In *Proceedings of ICSE'07*, pp. 54-64, 2007.

[11] M. Sabetzadeh and S. Easterbrook. "View Merging in the Presence of Incompleteness and Inconsistency". *Requirements Engineering Journal*, 11(3), pp. 174-193 2006.

[12] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik. "A Relationship-Driven Approach to ModelMerging". In *Proceedings of MiSE'07*, May 2007.

[13] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. "Consistency Checking of Conceptual Models via Model Merging". In *Proceedings of RE'07*, 2007.

[14] S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In *Proceedings of SIGSOFT FSE'04*, pp. 43–52, 2004.

[15] Unified Modelling Language (UML). http://www.omg.org/technology/documents/formal/uml.hml