

# Splitting Models Using Information Retrieval and Model Crawling Techniques

Daniel Strüber<sup>1</sup>, Julia Rubin<sup>2,3</sup>, Gabriele Taentzer<sup>1</sup>, and Marsha Chechik<sup>3</sup>

<sup>1</sup> Philipps-Universität Marburg, Germany

<sup>2</sup> IBM Research – Haifa, Israel

<sup>3</sup> University of Toronto, Canada

strueber@mathematik.uni-marburg.de, mjulia@il.ibm.com  
taentzer@mathematik.uni-marburg.de, chechik@cs.toronto.edu

**Abstract.** In team environments, models are often shared and edited by multiple developers. To allow modularity and facilitate developer independence, we consider the problem of splitting a large monolithic model into sub-models. We propose an approach that assists users in iteratively discovering the set of desired sub-models. Our approach is supported by an automated algorithm that performs model splitting using information retrieval and model crawling techniques. We demonstrate the effectiveness of our approach on a set of real-life case studies, involving UML class models and EMF meta-models.

**Keywords:** model management, model splitting, feature location.

## 1 Introduction

*Model-based engineering* – the use of models as the core artifacts of the development process – has gained increased popularity across various engineering disciplines, and already became an industrially accepted best practice in many application domains. For example, models are used in automotive and aerospace domains to capture the structure and behavior of complex systems, and, in several cases, to generate fully functional implementations. Modeling frameworks themselves, such as UML and EMF, are defined using models – an approach known as *meta-modeling*.

Together with the increased popularity of modeling, models of practical use grow in size and complexity to the point where large monolithic models are difficult to comprehend and maintain. There is a need to *split* such large models into a set of dependent modules (a.k.a. *sub-models*), increasing the overall comprehensibility and allowing multiple distributed teams to focus on each sub-model separately.

Most existing works, e.g., [3], suggest approaches for splitting models based on an analysis of strongly connected components, largely ignoring the semantics of the split and the user intention for performing it. In our work, we propose an alternative, heuristic approach that allows splitting a model into *functional* modules that are explicitly specified by the user using *natural-language descriptions*. It is inspired by code-level feature location techniques [2, 10], which discover implementation artifacts corresponding to a particular, user-defined, functionality.

In the core of our approach is an automated technique that employs *information retrieval (IR)* and *model crawling*. Given an input model and a set of its sub-model

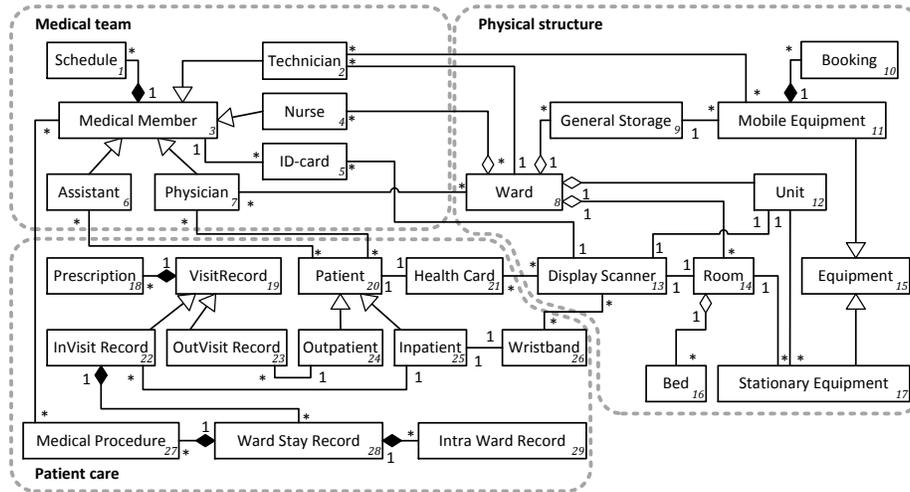


Fig. 1. A UML Class Model of a Hospital System.

descriptions, the technique assigns each element to one of the specified sub-models, effectively producing a partitioning. The technique is applicable to any model for which a split results in sub-models that satisfy the well-formedness constraints of the original one, e.g., UML Class models, EMF models and MOF-based meta-models.

**Motivating Example.** Consider the UML Class Model of a Hospital System (HSM) [7, p. 125] shown in Fig. 1. It describes the organization of the hospital in terms of its medical team (elements #1-7), physical structure (elements #8-17), and patient care (elements #18-29). Each of these concepts corresponds to a desired sub-model, visually encircled by a dashed line for presentation purposes. The goal of our work is to assist the user in determining elements that comprise each sub-model. The user describes the desired sub-models using natural-language text, e.g., using parts of the system documentation. For example, the medical team sub-model in Fig. 1 is described in [7]. A fragment of the description is: “Nurses are affiliated with a single ward, while physicians and technicians can be affiliated with several different wards. All personnel have access to a calendar detailing the hours that they need to be present at the various wards. Nurses record physicians’ decisions. These are written on paper and handed to an administrative assistant to enter. The administrative assistant needs to figure out who needs to be at a particular procedure before they enter it in the system.” The technique uses such descriptions in order to map model elements to desired sub-models. The labels for the sub-models, e.g., “Medical Team”, are assigned manually.

The user can decide whether the list of sub-models describes a *complete* or a *partial* split of the input model. In the former case, each input model element is assigned to exactly one sub-model, like in the example in Fig. 1, where the three sub-models “cover” the entire input model. In the latter case, when the complete set of the desired sub-models is unknown upfront, the technique produces assignments to known sub-models only. The remaining elements are placed in a sub-model called “rest”. The user can inspect the “rest” sub-model in order to discover remaining sub-models in an incremental and iterative fashion, until the desired level of completeness is achieved.

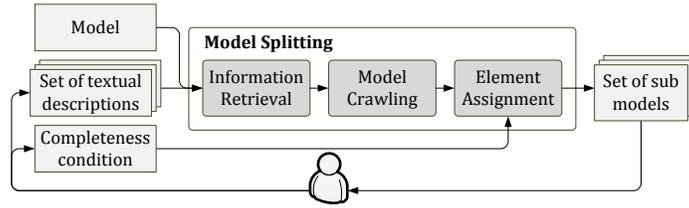


Fig. 2. An overview of the approach.

**Contributions and Organization.** This paper makes the following contributions: (1) we describe an automated model splitting technique which combines information retrieval and model crawling; (2) we propose a computer-supported iterative process for model splitting; (3) we evaluate our approach on a set of benchmark case studies, including real-life UML and EMF models. Our results demonstrate that the proposed approach achieves high accuracy compared to the manually produced results and is able to assist the user in the iterative discovery of the desired sub-models.

The rest of the paper is structured as follows. Sec. 2 gives the high-level overview of our approach. We describe the necessary preliminaries in Sec. 3 and present the automated splitting algorithm in Sec. 4. We report on the results of evaluating our approach in Sec. 5-6. In Sec. 7, we put our contribution in the context of related work, and conclude in Sec. 8 with the summary and an outline of future research directions.

## 2 Overview of the Approach

The high-level overview of our approach is given in Fig. 2. The user provides as input a model that requires splitting, a set of textual descriptions of the desired sub-models, and the completeness configuration parameter that declares whether this set of sub-models is complete or partial. For the example in Fig. 1, the complete set would contain descriptions of all three sub-models – medical team, physical structure, and patient care, while a partial set would contain only some of these descriptions.

**Automated technique.** In the core of our approach is an automated technique that *scores* the model elements w.r.t. their relevance to each of the desired sub-models. The scoring is done in two phases. The first one is based on *Information Retrieval (IR)* and uses sub-model descriptions: it builds a textual query for each model element, e.g., based on its name, measures its relevance to each of the descriptions and identifies those elements that are deemed to be most relevant for each of the descriptions.

The identified elements are used as *seeds* for the second phase, *Model Crawling*. In this phase, structural relationships between model elements are explored in order to identify additional relevant elements that were missed by the IR phase. The additional elements are scored based on their structural proximity to the already scored elements. In HSM, when identifying elements relevant to the medical team sub-model using the description fragment shown in Sec. 1, the IR phase correctly identifies elements #2,4,6,7 as seeds. It misses element #3 though, which is assigned a high score in the first iteration of crawling as it is closely related to the seeds. Once element #3 is scored, it impacts the scoring of elements identified during later iterations of crawling. Eventually, each model element’s relevance to each sub-model is scored.

The third phase, *Element Assignment*, assigns elements to sub-models based on their score. If a complete set of sub-models is given, each element is assigned to a sub-

model for which it has the highest score<sup>1</sup>. In this case, the assignment results in a model partition. If a partial set of sub-models is given as an input, some model elements might not belong to any of these sub-models. Hence, we apply a threshold-based approach and assign elements to sub-models only if their scores are above a certain threshold.

**Iterative process.** A partial set of sub-model descriptions can be further refined in an *iterative* manner, by focusing user attention on the set of remaining elements – those that were not assigned to any of the input sub-models. Additional sub-models identified by the user, as well as the completeness parameter assessing the user’s satisfaction with the set of known sub-models are used as input to the next iteration of the algorithm, until the desired level of completeness is achieved.

Clearly, as additional sub-models are identified, element assignments might change. For example, when only the description of the medical team sub-model is used during a split, element #8 is assigned to that sub-model due to the high similarity between its name and the description: the term *ward* is used in the description multiple times. Yet, when the same input model is split w.r.t. the sub-model descriptions of both the medical team and the physical structure, this element is placed in the latter sub-model: Both its IR score and its structural relevance to that sub-model are higher. In fact, the more detailed information about sub-models and their description is given, the more accurate the results produced by our technique become, as we demonstrate in Sec. 6.

### 3 Preliminaries

In this section, we describe our representation of models and model elements and capture the notion of model splitting. We also introduce IR concepts used in the remainder of the paper and briefly describe the feature-location techniques that we used as an inspiration for our splitting approach.

#### 3.1 Models and Model Splitting

**Definition 1.** A model  $M = (E, R, T, src, tgt, type)$  is a tuple consisting of a set  $E$  of model elements, a set  $R$  of relationships, a set  $T$  of relationship types, functions  $src$  and  $tgt : R \rightarrow E$  assigning source and target elements to relationships, and a function  $type : R \rightarrow T$  assigning types to relationships. Model elements  $x, y \in E$  are related, written  $related(x, y)$ , iff  $\exists r \in R$  s.t. either  $src(r) = x \wedge trg(r) = y$ , or  $src(r) = y \wedge trg(r) = x$ . If  $type(r) = t$ , we further say that  $x$  and  $y$  are related through  $t$ , written  $related_t(x, y)$ .

For example, the HSM in Fig. 1 has three relationship types: an association, a composition, and an inheritance. Further, element #7 is related to elements #3, #8 and #20.

**Definition 2.** Let a model  $M = (E, R, T, src, tgt, type)$  be given.  $S = (E_S, R_S, T, src_S, tgt_S, type_S)$  is a sub-model of  $M$ , written  $S \subseteq M$ , iff  $E_S \subseteq E$ ,  $R_S \subseteq R$ ,  $src_S = src|_{R_S}$  with  $src_S(R_S) \subseteq E_S$ ,  $tgt_S = tgt|_{R_S}$ , and  $type_S = type|_{R_S}$ <sup>2</sup>.

<sup>1</sup> An element that has the highest score for two or more sub-models is assigned to one of them randomly.

<sup>2</sup> For a function  $f : M \rightarrow M'$  with  $S \subseteq M$ ,  $f|_S : S \rightarrow M'$  denotes the restriction of  $f$  to  $S$ .

That is, while sources of all of a sub-model’s relationship are elements within the model, it does not have to be true about the targets. For example, each dashed frame in the example in Fig. 1 denotes a valid sub-model of HSM. All elements inside each frame form the element set of the corresponding sub-model. There are two types of relationships between these elements: those with the source and the target within the sub-model, e.g., all inheritance relations within the medical team sub-model, and those spanning two different sub-models (often, these are association relationships).

**Definition 3.** Given a model  $M$ , a model split  $Split(M) = \{S | S \subseteq M\}$  is a set of sub-models s.t.  $\forall S_1, S_2 \in Split(M) : (S_1 \neq S_2) \Rightarrow (E_{S_1} \cap E_{S_2} = \emptyset)$ .

By Def. 2, if  $\bigcup_{S \in Split(M)} E_S = E$ , then  $\bigcup_{S \in Split(M)} R_S = R$ . The split of HSM, consisting of three sub-models, is shown in Fig. 1.

**Definition 4.** A model  $M$  satisfying a constraint  $\varphi$  is splittable iff every sub-model of  $M$  satisfies  $\varphi$ .

All UML class models (without packages) are splittable since we can take any set of classes with their relationships and obtain a class model. Models with packages have a constraint “every class belongs to some package”. To make them splittable, we either relax the constraint or remove the packages first and then reintroduce them after the splitting is performed, in a new form.

### 3.2 Relevant Information Retrieval Techniques

Below, we introduce the basic IR techniques used by our approach.

**Term Frequency - Inverse Document Frequency Metric (TF-IDF) [8].** *Tf-idf* is a statistical measure often used by IR techniques to evaluate how important a term is to a specific document in the context of a set of documents (*corpus*). It is calculated by combining two metrics: *term frequency* and *inverse document frequency*. The first one measures the relevance of a specific document  $d$  to a term  $t$  ( $tf(t, d)$ ) by calculating the number of occurrences of  $t$  in  $d$ . Intuitively, the more frequently a term occurs in the document, the more relevant the document is. For the HSM example where documents are descriptions of the desired sub-models, the term `nurse` appears in the description  $d$  of the medical team sub-model in Sec. 1 twice, so  $tf(nurse, d) = 2$ .

The drawback of term frequency is that uninformative terms appearing throughout the set  $D$  of all documents can distract from less frequent, but relevant, terms. Intuitively, the more documents include a term, the less this term discriminates between documents. The *inverse document frequency*,  $idf(t)$ , is calculated as follows:  $idf(t) = \log(\frac{|D|}{|\{d \in D | t \in d\}|})$ . This metric is higher for terms that are included in a smaller number of documents.

The total *tf-idf* score for a term  $t$  and a document  $d$  is calculated by multiplying its *tf* and *idf* scores:  $tf-idf(t, d) = tf(t, d) \times idf(t)$ . In our example, since the term `nurse` appears neither in the description of the physical structure nor in patient care,  $idf(nurse) = \log(\frac{3}{1}) = 0.47$  and  $tf-idf(nurse, d) = 2 \times 0.47 = 0.94$ .

Given a query which contains multiple terms, the *tf-idf* score of a document w.r.t. the query is commonly calculated by adding the *tf-idf* scores of all query terms. For

example, the *tf-idf* score of the query “*medical member*” w.r.t. the description of the medical team sub-model is  $0 + 0 = 0$  as none of the terms appear in the description and thus their *tf* score is 0. The latent semantic analysis (LSA) technique described below is used to “normalize” scores produced by *tf-idf*.

**Latent Semantic Analysis (LSA) [4].** LSA is an automatic mathematical/statistical technique that analyzes the relationships between queries and passages in large bodies of text. It constructs vector representations of both a user query and a corpus of text documents by encoding them as a *term-by-document co-occurrence matrix*. It is a sparse matrix whose rows correspond to terms and whose columns correspond to documents and the query. The weighing of the elements of the matrix is typically done using the *tf-idf* metric.

Vector representations of the documents and the query are obtained by normalizing and decomposing the term-by-document co-occurrence matrix using a matrix factorization technique called *singular value decomposition* [4]. The similarity between a document and a query is then measured by calculating the cosine between their corresponding vectors, yielding a value between 0 and 1. The similarity increases as the vectors point “in the same general direction”, i.e., as more terms are shared between the documents. For example, the queries “*assistant*”, “*nurse*” and “*physician*” result in the highest score w.r.t. the description of the medical team sub-model. Intuitively, this happens because all these queries only have a single term, and each of the terms has the highest *tf-idf* score w.r.t. the description. The query “*medical member*” results in the lowest score: none of the terms comprising that query appear in the description.

### 3.3 Feature Location Techniques

*Feature location* techniques aim at locating pieces of code that implement a specific program functionality, a.k.a. a *feature*. A number of feature location techniques for code have been proposed and extensively studied in the literature [2, 10]. The techniques are based on static or dynamic program analysis, IR, change set analysis, or some combination of the above.

While the IR phase of our technique is fairly standard and is used by several existing feature location techniques, e.g., SNI AFL [17], our model crawling phase is heavily inspired by a code crawling approach proposed by Suade [9]. Suade leverages static program analysis to find elements that are related to an initial *set of interest* provided by the user – a set of functions and data fields that the user considers relevant to the feature of interest. Given that set, the system explores the *program dependance graph* whose nodes are functions or data fields and edges are function calls or data access links, to find all neighbors of the elements in the set of interest. The discovered neighbors are scored based on their *specificity* – an element is specific if it relates to few other elements, and *reinforcement* – an element is reinforced if it is related to other elements of interest. The set of all elements related to those in the initial set of interest is scored and returned to the user as a sorted *suggestion set*. The user browses the result, adds additional elements to the set of interest and reiterates.

Our modifications to this algorithm, including those that allow it to operate on models rather than code and automatically perform multiple iterations until a certain “fixed point” is achieved, are described in Sec. 4.

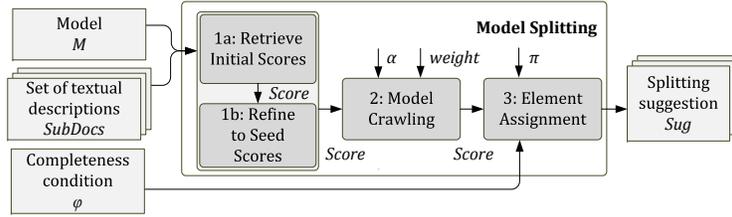


Fig. 3. An outline of the algorithm for creating a splitting suggestion.

## 4 Splitting Algorithm

Fig. 3 shows a refined outline of the algorithm introduced in Fig. 2. The algorithm receives a model  $M$  to be split, a set of textual descriptions of desired sub-models  $SubDocs$ , and a completeness condition  $\phi$  which is *true* if the set of descriptions represents a desired partitioning of  $M$  and *false* if this set is partial. The algorithm is based on scoring the relevance of model elements for each target sub-model (steps 1-2), and then assigning each element to the most relevant sub-model (step 3). The relevance scoring is done by first applying the IR technique and then using the scored sets of elements as seeds for model crawling. The latter scores the relevance of all model elements w.r.t. specificity, reinforcement, and cohesiveness of their relations. The algorithm also uses parameters  $w$ ,  $\alpha$  and  $\pi$  which can be user adjusted for the models being analyzed. Our experience adjusting them for class model splitting is given in Sec. 5.

**Step 1a: Retrieve Initial Scores Using LSA.** The user provides the input model  $M$  and natural-language sub-model descriptions  $SubDocs$  as unrelated artifacts. They need to be preprocessed before LSA can establish connections between them.  $SubDocs$  are textual and can be used as input documents directly. Textual queries are retrieved from elements of  $M$  by extracting a description – in class models, the element’s name. LSA then scores the relevance of each sub-model description to each model element description. The resulting scores are stored in *Score*, a data structure that maintains a map from  $(sub\text{-}model\ number, element)$  pairs to scores between 0 and 1.

**Step 1b: Refine initial scores to seed scores.** Some scored elements may not be suited as starting points for model crawling. If a model element description occurred in many different sub-model descriptions, its score might be too low. In this step, we use the technique proposed in [17] which involves inspecting the scores in descending order. The first gap greater than the previous is determined to be a *separation point* and all scores below it are discarded. The remaining scores are normalized for each sub-model to take the entire  $(0, 1]$  range.

**Step 2: Model crawling.** The aim of model crawling is to score the relevance of each model element for each target sub-model. Intuitively, model crawling is a breadth-first search: beginning with a set of seeds, it scores the neighbors of the seeds, then the neighbors’ neighbors, et cetera.

This step is outlined in Fig. 4: An exhaustive crawl is performed for each target sub-model. While there exists a scored element with unscored neighbors, we determine for each of these elements  $x$  and each relationship type  $t$  the set of directly related elements, calling it *OneHop* (lines 5-7). To score each unscored element in *OneHop*, the *TwoHop* set comprising *their* related elements is obtained (lines 8-9). The score is computed at line 10 as a product of  $x$ ’s score, a fraction quantifying specificity and reinforcement,

**Input:**  $M = (E, R, T, src, trg, type) : \text{Model}$   
**Input:**  $SubDocs$ : A set of  $i$  target sub-model descriptions  
**Input:**  $Score : ((1..i) \times E) \rightarrow [0, 1]$ : Map of (*sub-model number, element*) pairs to scores  
**Constant:**  $w : T \rightarrow (0, 1]$ : Weighting parameters for relationship types  
**Constant:**  $\alpha \in (0, 1]$ : Calibration parameter  
**Output:**  $Score : ((1..i) \times E) \rightarrow [0, 1]$

```

1 function CRAWLMODEL( $M, SubDocs, Score$ )
2   for each  $1 \leq j \leq i$  do
3     while  $\exists x, y \in E$  s.t.  $related(x, y) \wedge (Score(j, x) > 0) \wedge (Score(j, y) = 0)$  do
4       for each  $t \in T$  do
5         Var  $Scored \leftarrow \{x \in E \mid Score(j, x) > 0\}$ 
6         for each  $x \in Scored$  do
7           Var  $OneHop \leftarrow \{y \in E \mid related_t(x, y)\}$ 
8           for  $y \in OneHop \setminus Scored$  do
9             Var  $TwoHop \leftarrow \{z \in E \mid related_t(z, y)\}$ 
10             $Score.\underline{put}((j, y), (Score(j, x) * \frac{|TwoHop \cap Scored|}{|OneHop| * |TwoHop|} * w(t))^\alpha)$ 
11   return  $Score$ 

```

**Fig. 4.** Algorithm 1: Crawl model.

and a weighting factor. A constant exponent  $\alpha$  is applied to fine-tune the scoring distribution. Finally, we use a special operator, proposed by [9], to account for elements related to already scored elements through multiple relations. The operator, denoted by the underlined *put* command, merges the scores obtained for each relationship. It assigns a value higher than the maximum of these scores, but lower than 1.

This procedure adjusts the feature location algorithm proposed in [9] in three respects: (A1) We perceive neighborhood as being undirected; relations are navigated in both directions. Not considering directionality is powerful: It allows to eventually access *all* model elements, provided the model is connected. (A2) The weighting factor embodies the intuition that some relations imply a stronger coherence than others. An example is *composition* in UML, which binds the life cycles of elements together. (A3) We modified the scoring formula to reflect our intuition of reinforcement and specificity. The enumerator rewards a large overlap of the set of scored elements and those related to the element being scored, promoting high specificity and high reinforcement. The denominator punishes high connectivity of elements being analyzed, i.e., low specificity, and elements being scored, i.e., low reinforcement.

**Step 3: Element Assignment.** A splitting suggestion  $Sug$  is constructed by assigning suggested model elements to sub-models. When the complete split is desired, i.e.,  $\phi = true$ , each element is put into the sub-model for which it has the highest score. Ties are broken by selecting one at random. This guarantees that each element is assigned to exactly one sub-model. For a partial split, i.e.,  $\phi = false$ , an element is assigned to a sub-model only if its score exceeds the user-provided threshold value  $\pi$ . As a result, each element is assigned to zero or one sub-models.

**Proposition 1.** *For a splittable model  $M$ , the algorithm described in this section computes a model split  $Split(M)$  as defined in Def. 3.*

*Proof sketch:* In step 3, each model element is assigned to at most one sub-model. Thus, all pairs of sub-models eventually have disjoint sets of model elements, as required by Def. 3. The resulting sub-models satisfy all constraints satisfied by  $M$  because  $M$  is splittable (Def. 4).

Example	Decomposition Type	Sub-Models	Classes, Interfaces	Assoc.	Comp.	Aggr.	Gener.	Interf. Real.
HSM	Diagram split	3	28	10	5	4	16	0
GMF	Sub-model decomposition	4	156	62	101	0	70	65
UML	Package decomposition	14	242	283	213	0	205	0
WASL	Package decomposition	4	30	18	13	0	14	0
WebML	Package decomposition	2	23	11	13	0	12	0
R2ML	Package decomposition	6	104	96	27	0	76	0

Table 1. Subject models.

## 5 Experimental Settings

Our goal is to study the applicability and the accuracy of model splitting techniques when applied to real-life models. In this section, we describe our experimental setting. We focus the evaluation on two research questions: **RQ1**: How useful is the incremental approach for model splitting? and **RQ2**: How accurate is the automatic splitting?

### 5.1 Subjects

We chose subject models for our evaluation based on the following criteria: (1) the models should be splittable, as per Def. 4, modulo trivial pre- and post-processing; (2) we have access to an existing, hand-made splitting of the model which can be used for assessing our results; and (3) the splitting is documented, so that we can extract descriptions of the desired sub-models without introducing evaluator bias.

We selected six models that satisfy these criteria. The first four of these were known to the authors (convenience sampling); the last two were obtained by scanning the AtlanMod Zoo on-line collection of meta-models<sup>3</sup>. All models were either initially captured in UML or transformed from EMF to UML. The subjects are shown in Table 1 along with their particular decomposition types and metrics: The number of sub-models, classes and interfaces, associations, compositions, aggregations, generalizations, and interface realizations.

The first model, HSM [7], comprises three different diagrams and was already described in Sec. 1. Textual descriptions of the sub-models were extracted from [7]. The second, GMF<sup>4</sup>, is a meta-model for the specification of graphical editors, consisting of four viewpoint-specific sub-models. Three out of four textual descriptions of the sub-models were obtained from the user documentation on the GMF website. One missing description was taken from a tutorial web site for Eclipse developers<sup>5</sup>. The UML meta-model<sup>6</sup> is organized into 14 packages. The descriptions of these packages were extracted from the overview sections in the UML specification. The description of the four WASL packages was extracted from [16]. The description of the two WebML packages was obtained from the online documentation. Finally, R2ML is a markup language designed for rule interchange between systems and tools. It comprises six packages, each documented in [15].

<sup>3</sup> <http://www.emn.fr/z-info/atlanmod/index.php/Zoos>

<sup>4</sup> <http://www.eclipse.org/modeling/gmp/>

<sup>5</sup> <http://www.vogella.com/articles/EclipseEMF/article.html>

<sup>6</sup> <http://www.omg.org/spec/UML/2.5/Beta1/>

Association	Aggregation	Composition	Generalization	Interface Realization	$\alpha$
0.04	0.13	0.26	0.44	0.13	0.86

**Table 2.** Parameter assignment for class models.

The second and the third columns in Table 1 list the decomposition type and the number of target sub-models for each of the subjects. The last four columns present the size of the subject models in terms of the number of classes and relationships.

## 5.2 Methodology and Measurement

To investigate **RQ1**, we performed a qualitative analysis using a case study (Sec. 6.1) while for **RQ2**, we performed a set of quantitative experiments (Sec. 6.2). To evaluate the accuracy of our splitting technique, we used the following metrics:

1. *Expected*: the number of elements in the predetermined result, i.e., sub-model.
2. *Reported*: the number of elements assigned to the sub-model.
3. *Correct*: the number of elements correctly assigned to the sub-model.
4. *Precision*: the fraction of relevant elements among those reported, calculated as  $\frac{Correct}{Reported}$ .
5. *Recall*: the fraction of all relevant elements reported, calculated as  $\frac{Correct}{Expected}$ .
6. *F-measure*: a harmonized measure combining precision and recall, whose value is high if both precision and recall are high, calculated as  $\frac{2 \times Precision \times Recall}{Precision + Recall}$ . This measure is usually used to evaluate the accuracy of a technique as it does not allow trading-off precision for recall and vice versa.

## 5.3 Implementation

We implemented the proposed splitting algorithm for UML class models, considering the relationship kinds shown in table 1. Our prototype implementation is written in Java. As input, it receives an input model and text files providing the sub-model descriptions and configuration parameters.

For the IR phase, we used the LSA implementation from the open-source SemanticVectors library<sup>7</sup>, treating class and interface names as queries, and sub-model descriptions as documents. The crawling phase is performed using a model-type agnostic graph-based representation allowing us to analyze models of different types. We thus transformed the input UML models into that internal representation, focusing only on the elements of interest described above. We disregarded existing package structures in order to compare our results against them. The output sub-models were then transformed back to UML by creating a designated package for each.

Our technique relies on a number of configuration parameters described in Sec. 4: the calibration parameter  $\alpha$  shaping the distribution of scores and the *weight* map  $w$  balancing weights of specific relationship types. We fine-tuned these parameters using the hill climbing optimization technique [6]. Our goal was to find a *single* combination of parameter values yielding the best average accuracy for all cases. The motivation for doing so was the premise that a configuration that achieved good results on most members of a set of unrelated class models might produce good results on other class models, too. The results are summarized in Table 2.

<sup>7</sup> <http://code.google.com/p/semanticvectors/>

## 6 Results

In this section, we discuss our evaluation results, individually addressing each of the research questions.

### 6.1 RQ1: How useful is the incremental approach for model splitting?

We evaluate this research question on a case study based on the Graphical Modeling Framework (GMF). GMF comprises four sub-models: *Domain*, *Graphical*, *Tooling*, and *Mapping*. While the sub-models of GMF are already known, they may not necessarily be explicitly present in historically grown meta-models comparable to GMF. We assume that the person in charge of splitting the model is aware of two major viewpoints, *Domain* and *Graphical*, and wants to discover the remaining ones. She provides the meta-model and describes the sub-models as follows: “**Sub-model Domain** contains the information about the defined classes. It shows a root object representing the whole model. This model has children which represent the packages, whose children represent the classes, while the children of the classes represent the attributes of these classes. **Sub-model Graphical** is used to describe composition of figures forming diagram elements: node, connection, compartment and label.”

The user decides to begin with an incomplete splitting, since her goal is discovery of potential candidates for new sub-models. An incomplete splitting creates suggestions for sub-models *Domain*, *Graphical* as well as a “*Rest*” – for elements that were not assigned to either of the first two because they did not score above a predefined threshold value. The user can control the size of the *Rest* part by adjusting the threshold value according to her understanding of the model. After a suitable splitting is obtained, the *Rest* part contains the following elements: *ContributionItem*, *AuditedMetricTarget*, *DomainElementTarget*, *Image*, *Palette*, *BundleImage*, *DefaultImage*, *ToolGroup*, *MenuAction*, *MetricRule*, *NotationElementTarget*, *ToolRegistry*. From the inspection of these, the user concludes that a portion of the monolithic model seems to be concerned with tooling aspects of graphical editors comprising different kinds of toolbars, menu items, and palettes aligned around the graphical canvas. She describes this intuition: “**Sub-model Tooling** includes the definitions of a *Palette*, *MenuActions*, and other UI actions. The palette consists of basic tools being organized in *ToolGroups* and assigned to a *ToolRegistry*.”

A next iteration of splitting is performed. This time, the *Rest* comprises only four items: *MetricRule*, *DomainElementTarget*, *NotationElementTarget*, *AuditedMetricTarget*. Three out of these four elements signify a notion of defining relationships between elements of already known sub-models. She concludes that a separate sub-model is required for defining the integration and interrelation of individual sub-models. She performs a third and last splitting after providing a final sub-model description: “**Sub-model Mapping** binds the aspects of editor specification together. To define a mapping, the user creates elements such as *NotationElementTarget* and *DomainElementTarget* establishing an assignment between domain and notational elements.”

To investigate **RQ1** we further split it into two research questions: **RQ1.1**: Does the accuracy of splitting improve with each iteration? and **RQ1.2**: Does the approach assist the user in identifying missing sub-models?

**RQ1.1**: This question can be explored by considering the delta of each sub-model’s F-measure during multiple incremental splitting steps. As shown in Table 3, the increase

Run	Domain	Graphical	Tooling	Mapping
1	80%	77%	–	–
2	80%	84%	90%	–
3	86%	94%	90%	68%

**Table 3.** F-Measure during three runs of incremental splitting.

	1: IR Only			2: IR + Plain			3: IR + Undirected			4: Overall		
	Prec.	Recall	F-M.	Prec.	Recall	F-M.	Prec.	Recall	F-M.	Prec.	Recall	F-M.
HSM	93%	42%	<b>56%</b>	93%	53%	<b>67%</b>	78%	78%	<b>75%</b>	90%	92%	<b>89%</b>
GMF	100%	9%	<b>17%</b>	99%	30%	<b>38%</b>	68%	72%	<b>68%</b>	86%	87%	<b>86%</b>
UML	57%	21%	<b>24%</b>	37%	20%	<b>22%</b>	34%	38%	<b>30%</b>	50%	58%	<b>48%</b>
WASL	88%	48%	<b>61%</b>	72%	29%	<b>38%</b>	68%	64%	<b>63%</b>	92%	91%	<b>89%</b>
WebML	100%	37%	<b>52%</b>	100%	40%	<b>56%</b>	88%	94%	<b>90%</b>	93%	97%	<b>95%</b>
R2ML	81%	22%	<b>32%</b>	74%	30%	<b>30%</b>	46%	49%	<b>42%</b>	75%	77%	<b>74%</b>
UML <sub>funct</sub>	67%	22%	<b>30%</b>	76%	24%	<b>33%</b>	64%	66%	<b>61%</b>	84%	80%	<b>80%</b>

**Table 4.** Accuracy of model splitting.

of accuracy is monotonic in all sub-models! The same threshold value was used for all splits. The discovery process not only helped the user to discover the desired sub-models but also to create short sub-model descriptions which can later be used for documentation.

**RQ1.2:** In the first query, the *Rest* part has 12 elements, whereas in the original model, its size was 139. All 12 elements actually belong to the yet undiscovered sub-models, *Tooling* and *Mapping*. Thus, we are able to conclude that the user was successfully guided to concentrate on discovering these sub-models without being distracted by contents of those sub-models she knew about upfront.

## 6.2 RQ2: How accurate is the automatic splitting?

We investigate **RQ2** by answering two research questions: **RQ2.1:** What is the overall accuracy of the splitting approach? and **RQ2.2:** What is the relative contribution of individual aspects of the splitting algorithm on the overall quality of the results?

**RQ2.1:** Column 4 in Table 4 presents average precision, recall and F-measure of our automated technique for each of the subject models. For five out of the six models, the achieved level of accuracy in terms of F-measure was good to excellent (74%-95%). However, the result for UML was not as good (48%). Detailed inspection of this model revealed that package organization of UML has a special, centralized structure: it is based on a set of global hub packages such as *CommonStructure* or *CommonBehavior* that provide basic elements to packages with more specific functionality such as *Use-Case* or *StateMachine*. Hub packages are *strongly coupled* with most other packages, i.e., they have a low ratio of inter- to intra-relations. For example, the class *Element* is a transitive superclass for all model elements. This violation of the software engineering principle of *low coupling* hinders our topology-based approach for splitting.

To evaluate whether our algorithm produces meaningful results *except for hubs*, we derived a sub-model of UML which is restricted only to the functional packages. This sub-model, `umlfunct`, comprises 10 out of 14 packages and 188 out of 242 model elements of UML. As shown in Table 4, the accuracy results of `umlfunct` were similar to the five successful case studies (80%).

**RQ2.2:** Columns 1, 2 and 3 of Table 4 list contributions of individual steps of the algorithm and of the adjustments (A1-3) described in Sec. 4. The results after the IR phase are shown in column 1. Compared to the overall quality of the algorithm (column 4), the results are constantly worse in terms of the F-measure, due to low recall values. That is, IR alone is unable to find a sufficient number of relevant elements.

In column 2, we present the results of IR augmented with basic crawling which respects directionality, i.e., does not navigate relations from their inverse end. This version is similar to the crawling technique proposed by Suade but adjusted to operate on models rather than on code-level artifacts. The results are again worse than those of the overall technique due to low recall values. Interestingly, in some cases, e.g., WASL, the results are also worse than those of the plain IR technique in terms of both precision and recall, making the scoring schema related to this crawling strategy really inefficient.

Column 3 shows the results when crawling discards directionality, i.e., applies A1. This strategy results in a significant improvement in recall and the overall F-measure compared to the previous approach, but comes together with some decrease in precision.

Column 4 shows the results when the previous approach is extended with scoring modifications (A2-A3). This approach is clearly superior to the previous ones in terms of both precision and recall, and, as a consequence, of the overall F-measure.

We conclude that the basic crawling technique that worked well for code in case of Suade is not directly applicable to models, while our improvements allowed the algorithm to reach high accuracy in terms of both precision and recall.

### 6.3 Threats to Validity

Threats to external validity are most significant for our work: the results of our study might not generalize to other cases. Moreover, because we used a limited number of subjects, the configuration parameters might not generalize without an appropriate tuning. We attempted to mitigate this threat by using real-life case studies of considerable size from various application domains. The ability to select appropriate sub-model descriptions also influences the general applicability of our results. We attempted to mitigate this threat by using descriptions publicly available in online documentation.

## 7 Related Work

In this section, we compare our approach with related work.

**Formal approaches to model decomposition.** A formally founded approach to model splitting was investigated in [3]. This approach uses strongly connected components (SCCs) to calculate the space of possible decompositions. The user of the technique may either examine the set of all SCCs or try to find reasonable unions of SCCs according to her needs. In a recent publication [5], the same authors rule out invalid decompositions by providing precise conditions for the splitting of models conforming to arbitrary meta-models. This work is orthogonal to ours: Our technique requires a basic notion of a model being splittable, mostly motivated by the need to split class models and meta-models.

Another formal approach to model splitting is discussed in [13]: The authors show that a split of a monolithic meta-model into a set of model components with export and

import interfaces can be propagated for the automatic split of instances of the meta-model. However, this technique does not automate the meta-model splitting; the user has to assign model elements to target components by hand.

**Graph clustering for meta-models and architecture models.** *Graph clustering* is the activity of finding a partition of a given graph into a set of sub-graphs based on a given objective. Voigt [14] uses graph clustering to provide a divide-and-conquer approach for the matching of meta-models of 1 million elements. Of the different graph clustering algorithms, the author chose Planar Edge Separator (PES) for its run-time performance, and then adapted it to meta-model matching. Like us, he provides weighting constants for specific relationships kinds; yet [14] only presents the values of these constants and does not evaluate their impact on the quality of the match. From a software engineering perspective, the main drawback of this approach is that the user cannot influence the decomposition in terms of the concepts represented in the resulting sub-models. The same objection may be raised for the meta-model splitting tool proposed in [12]. Our approach bases the decomposition on user description of the desired sub-models, thus avoiding the need for the user to comprehend and classify the resulting components.

The architecture restructuring technique by Streekmann [11] is most similar to our approach. This technique assumes a legacy architecture that is to be replaced by an improved one. Similar to our technique, the starting point for the new organization is a set of target components together with a set of seeds ([11] calls them *initial mappings*) from which the content is derived. Yet, unlike in our approach, these seeds are specified manually by the developer. The clustering is performed by applying a traditional hierarchical clustering algorithm assigning model elements to components. The algorithm supports the weighting of different types of relationships; tuning these strongly impacts the quality of the decomposition. For the scenarios given in [11], the weighting differs from case to case significantly. In this work, in turn, we were able to find a specific setting of values that produced good results for an (albeit small) selection of unrelated class models. Streekmann also discusses algorithm stability w.r.t. arbitrary decisions made by it. During hierarchical clustering, once two elements are assigned to the same cluster (which, in the case of multiple assignment options, may be an arbitrary choice), this decision is not reversible. Arbitrary decisions in this style do not occur in our approach since we calculate relevance scorings for each sub-model individually.

**Model slicing.** *Model slicing* is a technique that computes a fragment of the model specified by a property. In the approach in [1], slicing of a UML class model results in a sub-model which is either *strict*, i.e., it satisfies all structural constraints imposed by the meta-model, or *soft*, if conformity constraints are relaxed in exchange of additional features. For example, slicing a class model can select a class and all of its subclasses, or a class and its supertypes within radius 1, etc. Compared to model splitting, model slicing concentrates on computing a sub-model of interest, ignoring the remainder of the model. In contrast, we use textual descriptions as input to IR to identify sub-models. The techniques are orthogonal and can be combined, as we plan to do in the future.

## 8 Conclusions and Future Work

Splitting large monolithic models into disjoint sub-models can improve comprehensibility and facilitate distributed development. In this paper, we proposed an incremental

approach for model splitting, supported by an automated technique that relies on information retrieval and model crawling. Our technique was inspired by code-level feature location approaches which we extended and adapted to operate on model-level artifacts. We demonstrated the feasibility of our approach and a high accuracy of the automated model splitting technique on a number of real-life case studies.

As part of future work, we intend to enhance our technique with additional types of analysis, e.g., considering cohesion and coupling metrics when building sub-models. We also plan to extend our approach to consider information obtained by analyzing additional model elements such as class attributes, methods and their behavior.

In addition, we are interested in further investigating the impact of sub-model descriptions on the overall accuracy of our approach and in suggesting strategies both for identifying good descriptions and for improving existing ones. Further involving the user in the splitting process, e.g., allowing her to provide partial lists of elements to be present (absent) in particular sub-models, might improve the results of the automated analysis significantly. We aim to explore this direction in the future.

**Acknowledgements.** We wish to thank Martin Robillard for making the source code of Suade available to us and the anonymous reviewers for their valuable comments.

## References

1. A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling Model Slicers. In *Proc. of MODELS'11*, volume 6981 of *LNCS*, pages 62–76, 2011.
2. B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.
3. P. Kelsen, Q. Ma, and C. Glodt. Models Within Models: Taming Model Complexity Using the Sub-Model Lattice. In *Proc. of FASE'11*, pages 171–185, 2011.
4. T. K. Landauer, P. W. Foltz, and D. Laham. An Introduction to Latent Semantic Analysis. *Discourse Processes*, (25):259–284, 1998.
5. Q. Ma, P. Kelsen, and C. Glodt. A Generic Model Decomposition Technique and Its Application to the Eclipse Modeling Framework. *J. Soft. & Sys. Modeling*, pages 1–32, 2013.
6. L. Pitsoulis and M. Resende. *Handbook of Applied Optimization*. Oxford Univ. Press, 2002.
7. Y. T. Rad and R. Jabbari. Use of Global Consistency Checking for Exploring and Refining Relationships between Distributed Models: A Case Study. Master's thesis, Blekinge Institute of Technology, School of Computing, January 2012.
8. A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge Univ. Press, 2011.
9. M. P. Robillard. Automatic Generation of Suggestions for Program Investigation. In *Proc. of ESEC/FSE-13*, pages 11–20, 2005.
10. J. Rubin and M. Chechik. A Survey of Feature Location Techniques. In I. Reinhartz-Berger et al., editor, *Domain Engineering: Product Lines, Conceptual Models, and Languages*. Springer, 2013.
11. N. Streekmann. *Clustering-Based Support for Software Architecture Restructuring*. Springer, 2011.
12. D. Strüber, M. Selter, and G. Taentzer. Tool Support for Clustering Large Meta-models. In *Proc. of BigMDE'13*, 2013.
13. D. Strüber, G. Taentzer, S. Jurack, and T. Schäfer. Towards a Distributed Modeling Process Based on Composite Models. In *Proc. of FASE'13*, pages 6–20, 2013.
14. K. Voigt. *Structural Graph-based Metamodel Matching*. PhD thesis, Univ. of Dresden, 2011.
15. G. Wagner, A. Giurca, and S. Lukichev. A Usable Interchange Format for Rich Syntax Rules Integrating OCL, RuleML and SWRL. *Proc. of WSh. Reasoning on the Web*, 2006.
16. U. Wolfgang. Multi-platform Model-driven Software Development of Web Applications. In *Proc. of ICISOFT'11 (Vol. 2)*, pages 162–171, 2011.
17. W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a Static Noninteractive Approach to Feature Location. *ACM TOSEM*, 15:195–226, 2006.