HAL
open science

Estimating Footprints of Model Operations

Cédric Jeanneret, Martin Glinz, Benoit Baudry

To cite this version:

Cédric Jeanneret, Martin Glinz, Benoit Baudry. Estimating Footprints of Model Operations. International Conference on Software Engineering, May 2011, Honolulu, United States. hal-00641091

HAL Id: hal-00641091

https://inria.hal.science/hal-00641091v1

Submitted on 14 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Estimating Footprints of Model Operations

Cédric Jeanneret
University of Zurich
Binzmühlestrasse 14
8050 Zurich, Switzerland
jeanneret@ifi.uzh.ch

Martin Glinz
University of Zurich
Binzmühlestrasse 14
8050 Zurich, Switzerland
glinz@ifi.uzh.ch

Benoit Baudry
IRISA
Campus universitaire de Beaulieu
35042 Rennes, France
bbaudry@irisa.fr

ABSTRACT

When performed on a model, a set of operations (e.g., queries or model transformations) rarely uses all the information present in the model. Unintended underuse of a model can indicate various problems: the model may contain more detail than necessary or the operations may be immature or erroneous. Analyzing the *footprints* of the operations – i.e., the part of a model actually used by an operation – is a simple technique to diagnose and analyze such problems. However, precisely calculating the footprint of an operation is expensive, because it requires analyzing the operation's execution trace.

In this paper, we present an automated technique to *estimate* the footprint of an operation without executing it. We evaluate our approach by applying it to 75 models and five operations. Our technique provides software engineers with an efficient, yet precise, evaluation of the usage of their models.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Algorithms, Measurement

Keywords

model footprint, model operation, static analysis

1. INTRODUCTION

A model is an abstract representation of an original for a given purpose. In Software Engineering, many kinds of models are used in various contexts, ranging from informal models sketched on a whiteboard to executable models deployed in a production environment. In this paper, we are interested in models used for analysis and generation purposes. Such models are mainly used as input to various *model operations* like queries, simulations, views extractions or model transformations. Increasingly, model operations are automated to improve both the productivity of engineers and the reliability of the analysis [25]. To support this automation, models must be machine-processable, i.e., models must be expressed in a modeling language with a formal syntax, such as UML [20], ADORA [10] or Petri nets. These modeling languages are defined by a metamodel.

When performed on a model, an operation computes new information based on the information stored in the model. During its execution, it navigates through the content of the model and gathers some information by reading some of its elements. The set of elements touched by a model operation during its execution forms the *footprint* of that operation. Thus, the footprint contains all elements that affect the outcome of the operation, as long as this operation is deterministic and does not use data other than those contained in the input model.

Footprints rarely cover models completely. The ratio between the size of a footprint and the size of the model quantifies the *model usage* of an operation. This measure can be used as a diagnosis to detect problems with the model's scope or its level of detail or the presence of faults in an operation. Identifying the footprint of an operation in a model further helps engineers in solving these problems by highlighting elements in the model that were used by the operation and separating them from the elements that remained unused.

Actual footprints can be computed with a dynamic analysis [5], by tracing the execution of an operation on a model. Unfortunately, the dynamic nature of footprints makes them at least as expensive to compute as performing the operation. To be effective and practical, a diagnosis must be available when it is needed and should be inexpensive to perform. Therefore, obtaining the footprint of an operation without having to execute it would be valuable to software engineers.

In this paper, we motivate the use of footprints in modeling activities and then concentrate on a novel approach for effectively and efficiently *estimating footprints* without executing the operation. In a nutshell, our approach works as follows: By analyzing the formal definition of an operation, we establish its metamodel footprint, the set of modeling constructs involved in this definition. The model footprint can then be estimated by selecting only those model elements that are instances of elements in the metamodel foot-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '11 Waikiki, Honolulu, Hawai
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

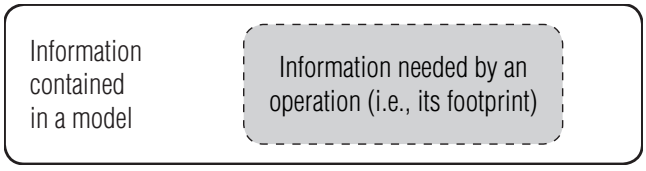

Figure 1: Gap between a model and its usage.

print. We call such a footprint estimate the *static footprint*, whereas the actual footprint is called the *dynamic footprint*. We have chosen this terminology in analogy to the static and dynamic analysis in the program analysis field [9].

We have evaluated our approach with 75 real-world models. First, we illustrate the meaningfulness of footprinting in a case study where we present 5 representative model operations that do not use all the information contained in these models, even when they are executed all together. Then, we evaluate empirically the quality of the estimation made by static footprinting. In experiments involving these 75 models and 5 (+ 1 combining these 5 operations) model operations, we demonstrate that static footprints (a) are indeed conservative estimates of dynamic footprints, (b) are very precise with respect to dynamic footprints and (c) are much faster to compute than dynamic footprints.

The remainder of the paper is organized as follows. In the next Section, we present the various uses of footprinting during modeling activities. In Section 3, we introduce a measure for assessing the model usage of an operation. Section 4 presents static and dynamic footprinting, while Section 5 evaluates our approach with a case study and some experiments. We discuss our contribution in Section 6 before contrasting it with related work in Section 7.

2. MOTIVATION

A typical modeling assignment consists of an original to be modeled, a modeling purpose and a modeling language, e.g., UML [20], which is defined by a metamodel. Frequently, the purpose of a model can be characterized in terms of an operation or a set of operations to be performed on that model, particularly in the context of model-driven development. In this case, a gap between the information contained in the model and the information required by the set of operations that will be executed on the model (cf. Figure 1) is an indicator of a problem.

For example, assume that a modeler needs to create a UML model with the purpose of deriving a performance model based on queueing networks as presented in [4]. Assume further that the modeler actually creates a model that fully documents the architecture of some software according to the "4+1" view model [13]. Among other diagrams, this model includes (1) use case and sequence diagrams for the scenario view, (2) class diagrams for the logical view, (3) component diagrams for the development view, (4) activity diagrams for the process view and (5) deployment diagrams for the physical view. Such a model would contain too much information for the operation presented above, because this operation only uses information from use cases diagrams, sequence diagrams and deployment diagrams.

Models with excessive information with respect to an operation require more time and resources for their creation and their processing by the operation than necessary. As illustrated in Figure 1, the modeler can identify information that is excessive with respect to an operation by establishing the footprint of this operation. An analysis of the size and extent of the actual footprint of an operation (or a set of operations) in comparison to the expected size and extent can reveal several problems: (i) the model may have the wrong scope, (ii) it may contain information that nobody needs or it may be on the wrong level of abstraction (i.e., it contains unnecessary details), (iii) it serves more or other purposes than initially intended, (iv) the operation(s) concerned may be erroneous or immature.

In the example given above, the presence of nodes in the deployment diagram that are not involved in any scenario (interaction) is a problem of scoping (i). Furthermore, the class diagram of the logical view is completely ignored by the operation (ii). Problem (iii) can be illustrated with the presence of comments meant to improve the understandability of the model. Finally, the operation ignores elements from the activity diagrams. Still, these diagrams could provide useful information about workload derivation [4], suggesting a possible improvement to the operation (iv).

Beyond problem identification, calculating the footprint of an operation may also serve for generating a dynamic view of a model that contains only those parts of a model that are relevant for a given operation.

Note that there is another form of gap between a model and its usage: A model may lack some important information for a given operation. In this case, the operation may fail or its result may be imprecise. This paper is concerned with the excessive part of a model with respect to a set of operations and leaves the missing part for future work.

For management purposes, it is sufficient to know the size of a footprint, so that it can be compared to the size of the model. By assessing systematically the model usage of operations, a project manager can locate gaps between models and their usages and undertake appropriate actions. Then, establishing the footprint can further help in reducing these gaps. In the next Section, we propose a measure to quantify the usage of a model by one or more operations.

3. MEASURING MODEL USAGE

In order to quantify the usage of a model by a model operation, we first need a metric for the size of a model or model footprint. For our purpose, it suffices to count the number of elements present in the model or footprint.

To illustrate the further discussion, we introduce a running example: A simple Petri net model is given in Figure 2. Figure 3 shows a metamodel defining Petri nets of the kind given in Figure 2. Basically, such a *metamodel* is a set of types and features. *Types* can either be (meta)classes or data types (enumerations or primitive types like integer). *Features* are divided into *structural features* (attributes within classes or references to other classes) and *behavioral features* (operations). For example, in the metamodel presented in Figure 3, `Transition` and `TransitionKind` are types, `capacity` and `source` are structural features and `fire()` is a behavioral feature.

In terms of its metamodel, a model can be regarded as a set of objects and settings. Every *object* is an instance of a class defined in the metamodel. A *setting* represents a value or a set of values held by an object for a structural feature. Settings can be set to a given value, reset to the feature's default value or unset. The set of settings in an object forms

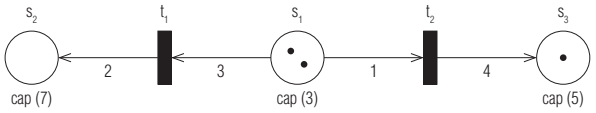

Figure 2: A Petri net.

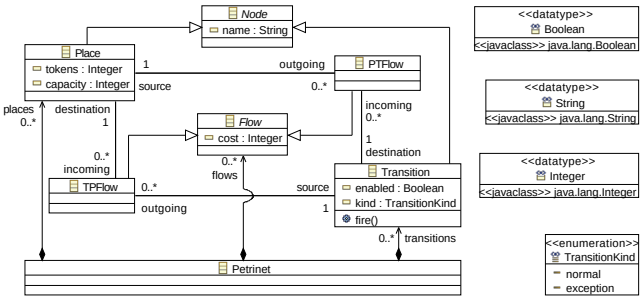

Figure 3: A metamodel for Petri nets.

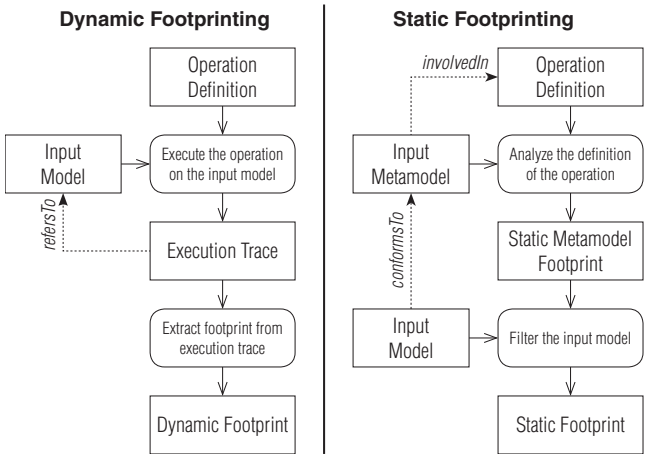

Figure 4: Dynamic and static footprinting of model operations.

the *state* of this object. In our Petri net example (Figure 2), place s_1 has the following five settings: (name = "s_1", tokens = 2, capacity = 3, incoming = nil, outgoing = $\{s_1t_1, s_1t_2\}$).

We define two size metrics by counting (i) the *number of objects* a model or footprint contains and (ii) the *total number of settings* present in the model or footprint. Thus, the size of our example Petri net is 10 objects and 40 settings: it contains 1 object of type `Petrinet` with 3 settings, 3 places (each object of type `Place` has 5 settings), 2 transitions (5 settings per transition) and 4 flows (3 settings per flow).

The usage of a model by a given model operation (or set of operations) can now be formally defined as follows. First, we recall that the footprint of an operation is the set of elements touched by this operation, i.e. the footprint consists of those parts of a model that are actually needed by the operation. If we have a set of operations, we define the footprint of this set as the union of the footprints of each individual operation. We can now define model usage with the *usage ratio* η, which has two components: η_o is based on the number of objects, while η_s is counting the number of settings.

$$\eta_o = \frac{\#\text{ objects in footprint}}{\#\text{ objects in model}}$$
$$\eta_s = \frac{\#\text{ settings in footprint}}{\#\text{ settings in model}}$$

The higher these values, the more a model operation uses the elements in the input model. In the extremes, $\eta = 1$ means that the operation uses the model completely, while $\eta = 0$ indicates an empty footprint. We explain how the footprint of an operation can be determined in the next Section.

4. CALCULATING FOOTPRINTS

The actual footprint of an operation in a model is the *dynamic footprint* (cf. the definition in the introduction). An algorithm for calculating a dynamic footprint is given in Section 4.1. The calculation is not difficult, but it is expensive, because it requires the execution of the operation on the model under analysis. Therefore, we introduce *static footprints* which can be calculated much faster and provide a conservative footprint estimate (i.e., the dynamic footprint is always a subset of the static one). The calculation of static footprints is presented in Section 4.2. Figure 4 illustrates both approaches. Section 4.3 compares static and dynamic footprinting.

4.1 Dynamic Footprinting

Dynamic footprinting (left part of Figure 4) is the most intuitive method to reveal the footprint left by an operation. It consists of executing the operation on the model while recording a trace of this execution. An *execution trace* is a set of events that occurred during the execution of an operation. We are interested in two kinds of events: accesses to the states of objects and invocations of operations on objects. We can then establish the dynamic model footprint of an operation from its execution trace with a simple bookkeeping algorithm.

This algorithm uses an array in two dimensions to keep track of relevant objects and settings. Each row represents an object, while each column represents a structural feature of the input metamodel. This array is initially empty. For every invocation of an operation *op* on an object *obj* in the execution trace, we create a row for *obj* (unless such a row already exists). Similarly, for every access to a structural feature *f* of an object *obj*, we create a row for *obj* (unless such a row already exists) and a column for *f* (unless such a column already exists) and we mark the cell at the intersection of the row and the column. Additionally, if *f* is a reference, we insert all referenced objects into the table. Once the trace has been fully analyzed, the dynamic footprint is the set of objects present in the array and the set of settings marked in the array.

To illustrate dynamic footprinting, we use our running example and calculate the footprint of a query that extracts the names of enabled transitions in Petri nets. This operation can be formally defined in OCL [19] as follows:

```
context
    Petrinet::namesOfEnabledTransitions():
    Set(String)
  body:
    self.transitions->select(t: Transition
        | t.enabled)->collect(t:
```

#	object	feature
1	p: Petrinet	*transitions*
2	t1: Transition	*enabled*
3	t1: Transition	*incoming*
4	s1t1: PTFlow	*source*
5	s1: Place	*tokens*
6	s1t1: PTFlow	*cost*
7	t1: Transition	*outgoing*
8	t1s2: TPFlow	*destination*
9	s2: Place	*capacity*
10	t1s2: TPFlow	*destination*
11	s2: Place	*tokens*
12	t1s2: TPFlow	*cost*
13	t2: Transition	*enabled*
[...]		
24	t2: Transition	*name*

objects \ structural features	*Petrinet::transitions*	*Transition::enabled*	*Transition::incoming*	*PTFlow::source*	*Place::tokens*	*Flow::cost*	*Transition::outgoing*	*TPFlow::destination*	*Place::capacity*	*Node::name*
p: Petrinet	■									
t1: Transition		■	■				■			
s1t1: PTFlow				■		■				
s1: Place					■					
t1s2: TPFlow						■		■		
s2: Place					■				■	
t2: Transition		■	■				■			■
s1t2: PTFlow				■		■				
t2s3: TPFlow						■		■		
s3: Place					■				■	

Figure 5: Dynamic footprinting of the Petri net example.

```
        Transition | t.name)
```

The attribute *enabled* (see metaclass `Transition` in Figure 3) is a structural feature whose value is derived from other values. A transition is enabled if (a) there are enough tokens in source places and (b) destination places have enough capacity to store additional tokens. It can be formally defined as follows:

```
context Transition::enabled: Boolean
  derive:
    self.incoming->forAll(f: PTFlow |
        f.source.tokens >= f.cost) and
        self.outgoing->forAll(f: TPFlow |
        f.destination.capacity -
        f.destination.tokens >= f.cost)
```

Executing this operation on the Petri net of Figure 2 yields an execution trace consisting of 24 accesses to objects (assuming that OCL expressions are not evaluated lazily). An excerpt of this trace is shown on the left side of Figure 5, while the result of the trace analysis is presented on the right side. During its execution, the operation has touched 10 objects and 21 settings. The operation therefore uses all objects in the model ($\eta_o = 1$) but only half of its settings ($\eta_s = 0.52$).

4.2 Static Footprinting

Frequently, model operations concentrate on certain kinds of model elements and do not touch the rest of the model. In metamodeling terms, this means that not all elements in the metamodel will be relevant for the operation. This is particularly the case with modeling languages that are designed to cover a large variety of modeling purposes such as UML. For example, [4] presents an operation which derives a performance model based on queueing networks from a UML model. While such a model may document many aspects of a software system, this operation only considers use case diagrams, sequence diagrams and deployment diagrams.

Static footprinting exploits this observation: it estimates the actual (dynamic) footprint by extracting those elements from the metamodel that are relevant for the operation (yielding a *metamodel footprint*) and then filtering the model by selecting only those model elements that are instances of the metamodel footprint (right part of Figure 4).

The static metamodel footprint of an operation is extracted through a static analysis of its formal definition. This analysis consists of collecting metamodel elements involved in the definition of the operation. For operations written using declarative languages (such as triple graph grammar [24]), this analysis is performed on the left-hand side part of every rule. If the operation is defined in an imperative language, metamodel elements are collected along the *control flow graph* of the operation, that is, the set of all its possible execution paths.

Table 1: Static metamodel footprint of the Petri net example.

Expression	Feature	Types
self.transitions	*Petrinet::transitions*	Petrinet, Transition
t.enabled	*Transition::enabled*	Transition, Boolean
self.incoming	*Transition::incoming*	Transition, PTFlow
f.source	*PTFlow::source*	PTFlow, Place
f.source.tokens	*Place::tokens*	Place, Integer
f.cost	*Flow::cost*	PTFlow, Integer
self.outgoing	*Transition::outgoing*	Transition, TPFlow
f.destination	*TPFlow::destination*	TPFlow, Place
f.destination.capacity	*Place::capacity*	Place, Integer
f.destination	*TPFlow::destination*	TPFlow, Place
f.destination.tokens	*Place::tokens*	Place, Integer
t.name	*Node::name*	Transition, String

More precisely, for every expression involving features, we add the feature and the type of the object on which this feature is "applied" (accessed or invoked) to the static metamodel footprint (unless these metamodel elements come from the language's library and not the input metamodel). For invocations of behavioral features (operations defined in classes), we also add the types of their parameters and their return types. For accesses to structural features, we include the type of this feature in the static metamodel footprint. Furthermore, when a class is added to the static metamodel footprint, we insert all its subclasses as well to make subsequent model filtering simpler.

Table 1 illustrates the analysis of the query introduced in Section 4.1 which extracts names of enabled transitions in Petri nets. The left column lists all expressions found in its control flow graph that involve a feature from the Petri net metamodel. The second column lists the features involved in these expressions while the right column lists the types involved in the corresponding expression. Thus, the static metamodel footprint in our example is the set of features and types in Table 1.

To visualize this static metamodel footprint, we can create a view of the metamodel specifically tailored for the query by pruning the complete metamodel [26]. Figure 6 presents such a view of the Petri net metamodel (the complete metamodel is illustrated in Figure 3). This view contains all metamodel elements from Table 1, and, additionally, the classes `Node` and `Flow`. They have been included in this view, because some of their features — `name` and `cost` — are part of the static metamodel footprint.

Once the static metamodel footprint has been computed, it can be used to create static (model) footprints by filtering input models. This filtering is straightforward: we first remove objects whose (meta)class is not part of the static metamodel footprint and, for every remaining object, we unset all its settings whose feature is not part of the static

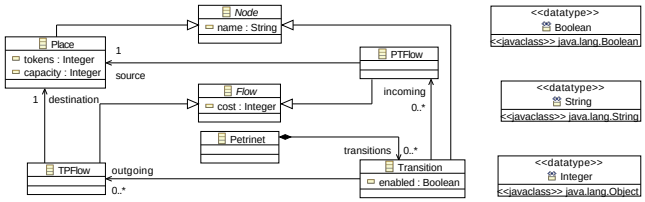

Figure 6: Static footprint metamodel for the Petri net example.

metamodel footprint. In our example, the static (model) footprint contains 10 objects and 26 settings: 1 petrinet (1 setting), 3 places (3 settings each), 2 transitions (4 settings each) and 4 flows (2 settings each). For comparison, the complete model has 10 objects and 40 settings. Thus, the model usage of the Petri net example is estimated to $\widehat{\eta_o} = 1$ and $\widehat{\eta_s} = 0.65$.

4.3 Comparison

In this Section, we compare dynamic and static footprinting and discuss their limitations. The major difference between them is that dynamic footprints can be obtained only *after* the execution of the operation, while static footprint can be computed *without* executing the operation. This difference has major impacts on both the effort required by these approaches and their precision.

Static footprinting requires less effort than dynamic footprinting. The static metamodel footprint must only be computed once for a given operation definition. It can then be used to filter the static footprint for any model. On the opposite, dynamic footprinting requires executing the operation on every model. Thus, it is preferable to use static footprinting if footprints are to be computed regularly (for example, when monitoring the model usage along the evolution of a model).

On the other hand, dynamic footprinting reveals the actual footprint of an operation with respect to an input model, while static footprinting estimates footprints based on two approximations. First, it considers all possible execution paths of an operation, not only the actual execution path. For operations implemented as graph transformations, it takes all rules into account, including those that are not actually used. Second, it essentially works with types, not with individual instances. This means that static footprinting overlooks all conditions defined in terms of object states. In our example, the names of all transitions are part of the static footprint, while only the names of enabled transitions are included in the dynamic footprint. Even worse, `name` being an attribute of `Node`, names of places are also part of the static footprint. Therefore, it may happen that static footprinting is not precise enough for assessment purposes. In the next Section, we evaluate our approach with a sample of models.

5. EVALUATION

In the previous section, we have presented two approaches to determine the footprint of an operation using a simple example to illustrate them. In this section, we report on experiments and case studies designed to evaluate our work. First, establishing footprints of model operation only makes sense if there exist a class of operations whose model usage is not 100%, that is, whose footprints do not cover the model completely. We investigated the existence of such operations in a case study (Section 5.3). Static footprinting is then only usable when it is (a) valid, (b) precise and (c) efficient. We evaluated static footprinting along these 3 directions with some experiments in Sections 5.4 and following. Finally, we discuss the threats to the validity of our evaluation in Section 5.7.

5.1 Implementation

To evaluate our approach, we have chosen Kermeta [17] as the language for the definition of operations. This choice is accidental to our contribution; the notion of footprint is not bound to a particular technical space [3] or technology to implement model operations. Actually, estimating footprints may have been easier for operations defined with graph grammars [28]. Nevertheless, the imperative and object-oriented nature of Kermeta makes it more accessible, and thus more relevant, to industrial practice. Furthermore, Kermeta is an executable metamodeling language compatible with the Eclipse Modeling Framework[1] (EMF), a popular implementation of EMOF based on Eclipse.

We have implemented both dynamic and static footprinting for operations written in Kermeta. For dynamic footprinting, we have extended the intepreter of Kermeta to save execution traces and have written a Java program to analyze them. The static analysis of Kermeta code has been implemented in Kermeta as a visitor on (type checked) abstract syntax trees of Kermeta programs. The filtering of models has been implemented in Java using the reflective API provided by EMF. Our implementation supports any input metamodels (examples include the metamodel for Petri nets in Figure 3, UML or any Domain Specific Language), as long as these metamodels are defined in Kermeta or in Ecore, which is the metamodeling language used in EMF. Key elements of this implementation are available on [11].

5.2 Models and Operations

The unit of analysis in our empirical work is the execution of an operation on an input model. For this evaluation, the models studied are actually metamodels written in Ecore. We decided to use metamodels instead of models of systems (i.e., we use M2 models instead of M1 models in the OMG's four layered metamodel architecture [18]) because we had not enough real-world models at our disposal whereas there are many real-world metamodels available for researchers in the public domain. We have selected a sample of 75 Ecore metamodels that were packaged in Eclipse plugins or available in online repositories, such as the *AtlanMod* metamodel zoo[2]. These models are accessible on [11].

Furthermore, we have defined 5 model operations to be executed on (meta)models. Since metamodels describe modeling languages, their typical use includes the documentation of modeling languages and storage and manipulation of models expressed in the modeling languages.

E2KV generates Kermeta code implementing a visitor for the input metamodel. This visitor can be used to implement operations on models conforming to the input metamodel.

[1] http://www.eclipse.org/emf
[2] http://www.emn.fr/z-info/atlanmod/index.php/Zoos

Table 2: Average model usage per operation.

	E2KV	E2SQL	E2HTML	E2DOT	E2GEN	E2* Suite
η_o	16.58%	49.43%	55.56%	51.70%	51.70%	55.86%
η_s	4.46%	25.43%	21.47%	22.57%	8.38%	39.25%

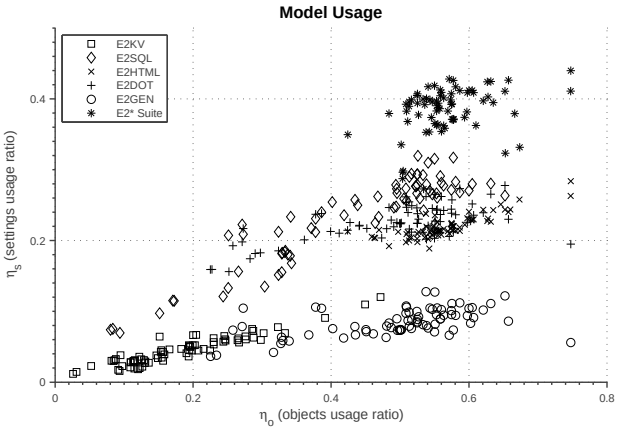

Figure 7: Model Usage of 75 models by 6 operations.

E2SQL creates a SQL schema for storing, in a database, models expressed in the input metamodel.

E2HTML creates an HTML document presenting the metamodel (as does Javadoc for Java code).

E2DOT visualizes the input metamodel with the help of GraphViz[3], a tool for rendering graphs.

E2GEN generates a generator model of the metamodel, which contains additional information for code generation. This generator model decorates the input model, that is, it contains reference to the input model.

More details about these operations can be found in [11], including their source code and their static metamodel footprint. In addition, we consider the operation combining these 5 operations to illustrate footprints left by a set of operations:

E2* Suite executes sequentially E2KV, E2SQL, E2HTML, E2GEN and E2DOT.

In total, our evaluation is based on 450 executions of operations ($5 + 1$ operations and 75 models). We have produced dynamic and static footprints for these 450 executions.

5.3 Model Usage

Table 2 presents the average (median) model usage while Figure 7 displays the usage ratio measured in our sample of models with respect to the operations (based on dynamic footprints). Every point represents the usage of one model by one operation. On the horizontal axis, we measure the usage in terms of objects (η_o), while the usage in terms of settings (η_s) is measured on the vertical axis. The plot is to be interpreted as follows: the higher / more right a point, the more an operation uses the elements of a model (in terms of settings respectively in terms of objects).

[3]http://www.graphviz.org/

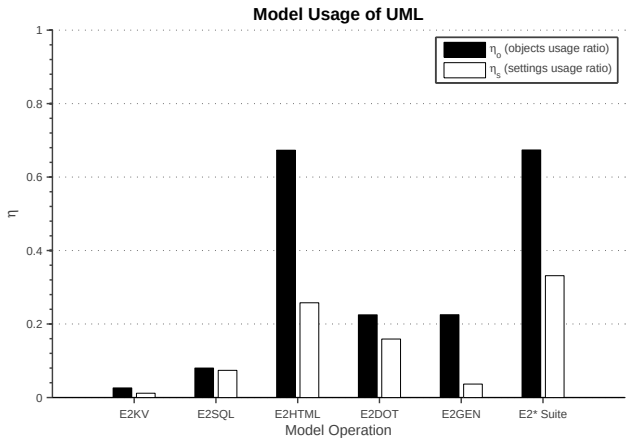

Figure 8: Usage of UML by 6 operations.

For example, when measuring the representation of UML in Ecore (Figure 8), we found that only 3% of its objects and 1% of its settings are used when creating a visitor in Kermeta for UML (E2KV). Note that many of the unused elements are used by other operations. Indeed, with respect to E2HTML, the model usage of UML increases to 67% in terms of objects and to 26% in terms of settings.

In average (median), 56% of the objects and 36% of the settings in a model are used with respect to our operation suite E2* (see Table 2). The low usage of E2KV can be explained by the fact that this operation is only interested in `EClass` objects and the inheritance relationships among them but not their content (such as `EAttribute` or `EOperation` objects). On the opposite, E2HTML considers almost every kind of model element (including some of their annotations). Nevertheless, we found no footprint that completely covers a model, because none of our operation uses `EGenericType` objects.

5.4 Validity of Static Footprints

To be of any use, static footprints must be conservative estimates of dynamic footprints. This property should hold by construction, but we nevertheless verified that our implementation satisfies this requirement by testing it with 450 test cases (1 test case per footprint). We executed the operations on both static footprints and complete models and compared the outcome of these executions. We found no difference between the outputs of these executions. In other words, all elements needed by the operations were indeed included in the static footprints. Note that E2GEN (and, consequently, E2*) created slightly different outputs, because the operation creates decorator models containing references to the input models. Thus, when E2GEN is executed on a static footprint, the output model refers to the static footprint rather than the complete model. This difference is insignificant and does not argue against the validity of static footprints.

5.5 Precision of Static Footprints

Since static footprints estimate dynamic footprints, we can assess the pertinence of this estimation by using the precision measure from the information retrieval field. For this purpose, we consider elements of the dynamic footprint

Table 3: Average precision of static footprints.

	E2KV	E2SQL	E2HTML	E2DOT	E2GEN	E2* Suite
σ_o	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
σ_s	89.26%	92.48%	92.89%	95.80%	65.71%	94.12%

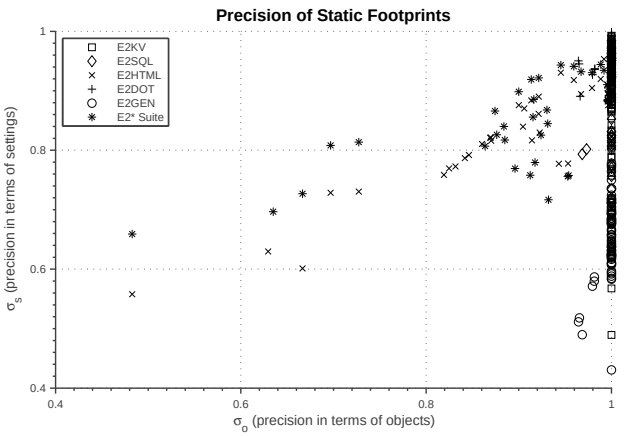

Figure 9: Precision of static footprints with respect to dynamic footprints.

as *relevant* while elements from the static footprint form the set of *retrieved* elements. *Precision* measures the proportion of retrieved elements that are indeed relevant. Note that the dual of precision, *recall* (the proportion of relevant statements that have been retrieved), is always trivial in this context (100%), because a static footprint is always a superset of the dynamic footprint it estimates. The precision of the estimation is measured by considering objects (σ_o) and settings (σ_s). The larger the measure σ, the closer is the static footprint to the dynamic footprint and the more accurate is the measure of model usage when it is based on static footprinting ($\widehat{\eta}$).

$$\sigma_o = \frac{\text{\# objects in dynamic footprint}}{\text{\# objects in static footprint}}$$
$$\sigma_s = \frac{\text{\# settings in dynamic footprint}}{\text{\# settings in static footprint}}$$

In our explanatory example (see Section 4), both footprints contain 10 objects. In terms of settings, the dynamic footprint has 21 settings while the static footprint contains 26 settings. Therefore, the precision of the static footprint is $\sigma_o = 1$ and $\sigma_s = 0.81$.

Figure 9 presents the precision of our 450 static footprints in comparison with their dynamic counterparts. On the horizontal axis, we measure the precision in terms of objects (σ_o), while the precision in terms of settings (σ_s) is measured on the vertical axis. Therefore, the higher / more right a point, the closer is the static footprint to the dynamic one.

In average, static footprints are very precise: the majority of static footprints contain no irrelevant objects (the median of σ_o is 100%) while still containing some irrelevant settings (see Table 3).

On the left part of the plot in Figure 9, there are 10 static footprints with a precision $\sigma_o < 0.8$. These are the footprints of five models (friends, BPMN, filesystem, flowchart and fsmStatic) with respect to two operations (E2HTML and E2*). The imprecision of these static footprints is due to `EAnnotation` objects. E2HTML (and consequently E2*) reads annotations whose *source* is "genmodel". These five models have a lot of annotations, but these annotations have different sources (they are destinated to other operations or relevant to other purposes). Dynamic footprints do not include the entries of these `EAnnotation` objects while static footprints include them. These outliers reveal a limitation of static footprinting: its precision can be rather low for operations whose usage depends on conditions defined in terms of object states.

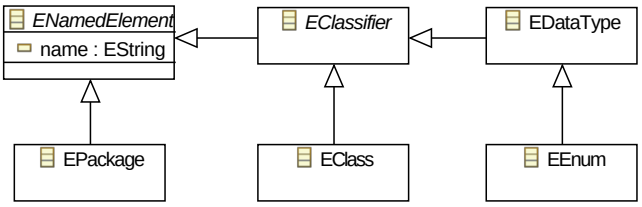

Figure 10: Excerpt of the Ecore metamodel: ENamedElement and some of its subtypes.

Many static footprints suffers from a lack of precision in terms of settings $\sigma_s < 0.6$ (lower left part of Figure 9). This lack of precision is due to the inheritance relationships among the types of the Ecore metamodel. An excerpt of this metametamodel is depicted in Figure 10: The feature *name* is defined in a class `ENamedElement`, which is the supertype of many other classes.

Two static footprints of E2KV are impacted by this problem. The static metamodel footprint of E2KV contains all classes depicted in Figure 10. Still, the operation only reads the *name* of `EClass` objects, but not the *name* of `EDataType` objects. Since Ecore and BPMN contain a lot of `EDataType` objects, their static footprints get imprecise in terms of settings, because these static footprints include settings for the name of `EDataType` objects, while the dynamic footprints do not include them.

E2GEN further illustrates this problem. Its static metamodel footprint also contains all classes of Figure 10. However, E2GEN only reads the *name* of `EPackage` objects. Thus, many objects in the static model footprints of E2GEN will have a setting for *name*, while only `EPackage` objects will have a setting for it in the dynamic footprint. This explains the low average of σ_s for E2GEN in Table 3.

Other operations read the *name* of each `ENamedElement` object in their static footprints. Thus, their static footprints are not impacted by this issue.

5.6 Efficiency of Static Footprinting

In this subsection, we evaluate the cost of static footprinting and compare it to the cost of dynamic footprinting. Table 4 summarizes the results of this evaluation. We chose 4 models with various size (in terms of objects) out of our sample: the smallest model, the largest one and two models in-between. For dynamic footprinting, we measure the time needed for both executing the operation while keeping a trace of its execution and extracting the dynamic footprint out of this trace. On the opposite, static footprinting has an initial cost for analyzing the operation definition to extract its static metamodel footprint. Once this metamodel footprint has been computed, it can be used to filter any model.

Table 4: Computation time of footprints.

Model	# Objects	Method	E2KV	E2SQL	E2HTML	E2DOT	E2GEN	E2* Suite
DocBook	23	Dynamic	1'955 ms	1'587 ms	1'525 ms	1'615 ms	1'541 ms	3'193 ms
		Static	8 ms	11 ms	10 ms	10 ms	10 ms	12 ms
		SpeedUp	*244x*	*144x*	*153x*	*162x*	*154x*	*266x*
XQuery	100	Dynamic	1'832 ms	1'782 ms	1'792 ms	2'171 ms	1'637 ms	3'863 ms
		Static	16 ms	24 ms	23 ms	24 ms	22 ms	22 ms
		SpeedUp	*115x*	*74x*	*78x*	*90x*	*74x*	*176x*
XHTML	1'035	Dynamic	3'211 ms	6'460 ms	8'762 ms	4'262 ms	2'238 ms	20'495 ms
		Static	101 ms	132 ms	108 ms	90 ms	66 ms	59 ms
		SpeedUp	*32x*	*49x*	*81x*	*47x*	*34x*	*347x*
OCLUML	13'849	Dynamic	6'348 ms	12'237 ms	41'267 ms	24'417 ms	5'426 ms	90'915 ms
		Static	222 ms	344 ms	651 ms	373 ms	111 ms	240 ms
		SpeedUp	*29x*	*36x*	*63x*	*65x*	*49x*	*379x*
Static Metamodel Footprint			6'242 ms	6'733 ms	6'878 ms	6'860 ms	6'910 ms	15'524 ms

Thus, we keep the cost of precomputing the static metamodel footprint separated from the cost of filtering models.

When the static metamodel footprint is provided, static footprint (filtering a model) is always cheaper than computing dynamic footprints, even for our smallest model and our simplest operation. The speed up ranges from 29× to 379×. When the metamodel footprint is not precomputed, static footprinting (precomputing the metamodel footprint and filtering a model) is faster than dynamic footprinting in 6 cases highlighted with shaded cells.

Execution times were measured as follows. Each footprint (whether dynamic, static model or static metamodel footprint) was computed 5 times, each time in a freshly started Java virtual machine. Table 4 displays the medians of these experiments. Kermeta code — model operations and the static analysis of model operations — was interpreted (and not compiled to Java). To minimize the influence of Eclipse internal mechanisms on our measure, we run Eclipse in headless mode and we forced the loading of required plugins by computing a dummy footprint before the one we were interested in. The computer used for this evaluation is an Intel(R) Core(TM) i7 @ 2.8 GHz with 8 Gb RAM running Windows 7 Professional, Java(TM) 1.6.0_20.b02 (64 bits) and Eclipse 3.5.2 with Kermeta 1.3.2.

5.7 Threats to Validity

We have only evaluated our approach with operations written by us. Thus, the evaluation of our approach may not be reliable and results may differ with a different set of operations. Our experiments require both models and operations. If we had used published model operations (such as those presented in [4], [14] or [27]), we would have had to create UML models. Creating models rather than model operations would have been an equally strong threat to the validity of our experiments. Still, to mitigate this threat, we either based our operations on existing tools (E2DOT, E2GEN and E2HTML) or benchmarks from the MDE community [2] (E2SQL) or used their results for implementing our approach (E2KV). Furthermore, the source code of these operations can be found in [11].

Finally, we only considered Ecore metamodels and operations written in Kermeta, because of the availability of many metamodels written in Ecore. There is no apparent reason to believe that static footprinting would be less precise or less efficient in other settings (e.g. models of software systems expressed in UML), but the limited scope of our experiment remains nevertheless a threat to its external validity.

6. DISCUSSION

Measuring model usage and footpring are meaningful, since many operations do not use all the information contained in the input models. Still, we did not investigate empirically to which extent model usage or footprinting improve modeling processes or increase the quality of models. This is future work.

Because a footprint contains all elements touched during the execution of an operation, the operation produces the same output when executed on a footprint as if it were executed on the complete model. There are nevertheless some exceptions. When an operation modifies its input models in place, such as refactorings, "untouched" model elements are implicitely copied to the output model and therefore impact the outcome of the operation. A similar issue exists with operations creating decorator models like E2GEN, that is, when the output model contains references to the input model. In these cases, footprints cannot be used in place of complete models as input to operations.

Furthermore, if an operation is not deterministic, e.g., it uses collections such as sets or hashtables, the outcome of the operation may differ when it is executed on a footprint or on the complete model. Still, the differences are often meaningless, e.g., the ordering of classes in a package.

Static footprinting estimates a dynamic footprint by considering types only. Therefore, static footprints become imprecise when an operation definition involves classes that have many subclasses. In addition, static footprinting ignores conditions expressed in terms of objects states. For example, the footprint of an operation working only on a `EPackage` called "persistence" will produce a static footprint containing all packages. Improving the precision of static footprints by using a more sophisticated static analysis and filters than presented in this paper is left for future work.

In a similar direction, if a model operation relies on a reflection mechanism (e.g. an interpreter which, given an OCL constraint and an UML model, evaluates the constraint on the UML model), the static footprint will be the complete model, as it is impossible to infer, from the model operation definition only, which objects or settings will be touched during the execution of the operation. In this case, one

must resort to dynamic footprinting.

So far, we have implemented our approach only for operations written in Kermeta, because its imperative and object-oriented nature makes it more accessible to industrial practices. We could implement footprinting for other transformation languages such as QVT [21] or VIATRA [28]. Actually, computing the static metamodel footprint of an operation written as a graph transformation is almost trivial: it suffices to collect metamodel elements present in the left-hand side argument of each transformation rule. Furthermore, some transformation languages have dedicated support for tracing a transformation's execution [5], making the dynamic footprinting approach easy to implement.

7. RELATED WORK

Many frameworks have been proposed to define and evaluate the quality of models. Among others, a good model is at the right level of detail [6] for its purpose. For Lindland [15], a model is semantically correct if it contains all statements that are correct and relevant for the problem at hand (completeness), but nothing more (validity). In [23], Schuette and Rotthowe proposes the minimalism criteria to operationalize their principle of construct adequacy. A model is *minimal* if none of its elements can be removed without a loss of information for the potential model users. These criteria rely on the evaluation of the relevance of the content of the model to the problem at hand, but, to the best of our knowledge, no objective measures has yet been proposed to quantify this attribute. As Davis et al. point out [6], this attribute is difficult to measure because it is highly scenario-dependent. Since more and more of activities involving models become automated (especially in a MDE environment [16]), we propose to use model operations for characterizing the purpose of a model. Thus, footprints can be used to define and measure the relevance of model elements based on their usage by model operations.

Along this line, footprints are best used with editors designed to visualize a single underlying model from various viewpoints, such as the ADORA editor [10] or the orthographic modeling environment [1]. Their visualization techniques can hide model elements irrelevant for a model operation, producing a view specifically tailored for a given model operation.

Formally, footprinting is a form of model slicing. However, unlike other model slices (such as [12]), our slicing criterion is not defined in terms of the behavior depicted by the model being sliced, but is related to the behavior of a model operation executed on the model.

Furthermore, model footprints can be used for impact analysis, i.e., deciding whether a change in a model impacts the outcome of the operation. In [7] for example, Egyed uses dynamic footprints of consistency rule instances (these footprints are called *scope* in [7]) to decide whether a given rule instance (that is, a rule evaluated with respect to a given model element) must be reevaluated after a change in the model. Later, he extended his technique to generate fixes for inconsistencies [8]: Since the scope of a rule instance contains all elements that affects its truth-value, at least one of these elements must change to fix the inconsistency. With these papers, Egyed demonstrated the advantages of instance-based incremental consistency checking over type-based incremental consistency checking. In our work, we are interested in operations applied to the model in its entirety, which typically requires more time to execute than verifying some conditions on some set of elements. Thus, we are interested in estimating footprints statically, rather than tracing the execution of the operation.

A metamodel footprint documents the usage of an operation. Many modeling methods prescribe which kind of details is to be modeled for a given perspective (e.g., [13] or [22]). Sometimes, the creators of operations document explicitely which elements their operation uses (e.g., Section 3 of [4]). Static footprinting not only generates automatically this documentation from the definition of operations, it can also be used to identify model elements that are excessive according to this documentation.

Metamodel footprints can also be used during the validation of model operations by checking that the metamodel footprint covers all relevant modeling constructs. [14] lists metamodel coverage as a possible fault in model transformations. This motivated Wang et al. to propose a tool analyzing the metamodel coverage of model transformations [29]. In contrast to their analysis, static metamodel footprints can be computed from model operations written in imperative languages.

Finally, metamodel pruning [26] is a generic algorithm which, given a metamodel and a set of its elements, extracts a view from the metamodel containing all these elements. In our work, we extract footprints from models. We use a variation of this algorithm to extract a view of static metamodel footprint from the complete metamodel (for an example of such a view in this paper, see Figure 6).

8. CONCLUSION AND FUTURE WORK

During its execution, a model operation typically uses only a fraction of the content of its input models. The part of a model actually touched by a model operation forms its footprint. Measuring the model usage of operations (comparing the size of a footprint with respect to the size of a model) helps project managers to detect problems within the models and the operations involved in their project. Then, establishing the footprint of some operations with respect to a model supports modelers in reducing the scope and decreasing the level of details in the model so that it only contains the information that impacts the outcome of these operations. Moreover, footprinting gives hint for finding defects in operations and may suggest improvements for them.

In this paper, we have presented a method to reveal the footprint left by a model operation (dynamic footprinting) and a method to estimate this footprint without executing the operation (static footprinting). We have implemented both approaches and compared them on 75 models and 5 model operations (+1 combining these operations). This experiment suggests that static footprinting can estimate dynamic (actual) footprints with a high precision (in average, 100% in terms of objects and 94% in terms of settings for E2*). Furthermore, static footprinting is between 29 and 379 times faster than dynamic footprinting when the static metamodel footprint of an operation is precomputed.

We believe that footprinting will reveal helpful information to build models at the right level of detail for their purposes. This suggests two complementary directions for future work. So far, we only considered the part of a model that was actually used and ignored the missing elements in a model that could have been used by an operation if they were modeled. Using static metamodel footprints, it may

be possible to provide modelers with some hints on missing elements in their models. Furthermore, automated model operations are only a fraction of uses of models. We plan to investigate means to assess objectively the adequacy of a model's level of detail with respect to other modeling purposes.

9. REFERENCES

[1] C. Atkinson and D. Stoll. Orthographic modeling environment. In *Fundamental Approaches to Software Engineering*, pages 93–96, 2008.

[2] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In *4th International Conference on Graph Transformations*, pages 396–410, 2008.

[3] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[4] V. Cortellessa and R. Mirandola. Deriving a queueing network based performance model from uml diagrams. In *Iinternational Workshop on Software and Performance (WOSP 00)*, pages 58–70, New York, NY, USA, 2000. ACM.

[5] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[6] A. Davis, S. Overmyer, K. Jordan, J. Caruso, F. Dandashi, A. Dinh, G. Kincaid, G. Ledeboer, P. Reynolds, P. Sitaram, A. Ta, and M. Theofanos. Identifying and measuring quality in a software requirements specification. In *First International Software Metrics Symposium*, pages 141–152, 1993.

[7] A. Egyed. Instant consistency checking for the uml. *International Conference on Software Engineering (ICSE 2006)*, 0:381–390, 2006.

[8] A. Egyed. Fixing inconsistencies in uml design models. In *International Conference on Software Engineering (ICSE 2007)*, pages 292 –301, 20-26 2007.

[9] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[10] M. Glinz, S. Berner, and S. Joos. Object-oriented modeling with ADORA. *Information Systems*, 27(6):425–444, 2002.

[11] C. Jeanneret, M. Glinz, and B. Baudry. Footprinting operations written in kermeta. `http://www.ifi.uzh.ch/rerg/people/jeanneret/footprints`, 2010.

[12] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of state-based models. In *International Conference on Software Maintenance*, pages 34–43, 2003.

[13] P. Kruchten. The 4+1 view model of architecture. *Software, IEEE*, 12(6):42 –50, nov. 1995.

[14] J. M. Küster and M. Abd-El-Razik. Validation of model transformations: first experiences using a white box approach. In T. Kühne, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2006*, volume 4364 of *LNCS*, pages 193–204, Berlin, Heidelberg, 2007. Springer.

[15] O. I. Lindland, G. Sindre, and A. Sølvberg. Understanding quality in conceptual modeling. *IEEE Software*, 11(2):42–49, 1994.

[16] P. Mohagheghi and J. Aagedal. Evaluating quality in model-driven engineering. In *International Workshop on Modeling in Software Engineering (MISE 07)*, page 6, Washington, DC, USA, 2007. IEEE Computer Society.

[17] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *8th International Conference on Model Driven Engineering Languages and Systems*, pages 264–278, 2005.

[18] OMG. Meta object facility version 2.0, formal/2006-01-01, 2006.

[19] OMG. Object constraint language specification version 2.0, formal/2006-05-01, 2006.

[20] OMG. Uml superstructure specification version 2.1.2, formal/2007-11-02, 2007.

[21] OMG. Qvt specification version 1.0, formal/08-04-03, 2008.

[22] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, April 2005.

[23] R. Schuette and T. Rotthowe. The guidelines of modeling: An approach to enhance the quality in information models. In *17th International Conference on Conceptual Modeling*, pages 240–254, 1998.

[24] A. Schürr. Specification of graph translators with triple graph grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, 1995.

[25] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[26] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model pruning. In *12th International Conference on Model Driven Engineering Languages and Systems*, pages 32–46, 2009.

[27] M. Shousha, L. Briand, and Y. Labiche. A uml/marte model analysis method for detection of data races in concurrent systems. In *12th International Conference on Model Driven Engineering Languages and Systems*, pages 47–61. Springer, 2009.

[28] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, 2002.

[29] J. Wang, S.-K. Kim, and D. Carrington. Verifying metamodel coverage of model transformations. In *17th Australian Software Engineering Conference (ASWEC 2006)*, page 10 pp., 18-21 2006.