An Extensible Meta-Model for Program Analysis

Dennis Strein
Omnicore Software
Werderstr. 87, 76137 Karlsruhe, Germany
Strein@omnicore.com
www.omnicore.com

Rüdiger Lincke Jonas Lundberg Welf Löwe
Växjö University, Software Technology Group, MSI
Vejdes Plats 7, SE-351 95 Växjö, Sweden
{Rudiger.Lincke|Jonas.Lundberg|Welf.Lowe}@vxu.se
www.vxu.se

***Abstract*— Software maintenance tools for program analysis and refactoring rely on a meta-model capturing the relevant properties of programs. However, what is considered *relevant* may change when the tools are extended with new analyses and refactorings, and new programming languages. This paper proposes a language independent meta-model and an architecture to construct instances thereof, which is extensible for new analyses, refactorings, and new front-ends of programming languages. Due to the loose coupling between analysis-, refactoring-, and front-end-components, new components can be added independently and reuse existing ones. Two maintenance tools implementing the meta-model and the architecture,** VIZZANALYZER **and** X-DEVELOP**, serve as a proof of concept.**

I. INTRODUCTION

Software maintenance is expensive today – estimations range from 50% to 80% of the total costs of ownership for a software system [1]. In all maintenance tasks, systems need to be comprehended first, and the effort for comprehension even dominates the total maintenance effort. Here, estimations range from 40% up to 90% [2], [3], [4], [5]. Comprehending design specifications or even source code needs to be supported by analysis tools, since, for real systems, these documents tend to become large and complex. The level of abstraction of analysis may vary, but, the principal tasks of these tools are the same: extracting information from a system, analyzing it, and, finally, displaying the results.

In many maintenance tasks, systems need to be changed, as well. These changes imply refactorings and each individual refactoring should be consistent and correct. To guarantee this, tool support is needed again, and the principle tasks that the tools ought to support are similar: extracting information, analyzing, and, finally, modifying the program.

Both analysis and refactoring tools capture and process the information extracted from a program. They abstract from the information of a program, i.e., they build a *model* of that program. This model has a certain type constraining the types of entities and relations captured since relevant for the analyses and refactorings. We refer to this type of the information extracted as the *meta-model*, since it describes all admissible models.

In order to reuse analyses and refactorings, we aim for a *common meta-model* that is independent of specific programming languages. However, some analyses and refactorings are clearly language-specific. Implementing them on a common meta-model is unnecessary since there is no potential of reuse anyway. They are still best implemented on language specific meta-models in language specific tools. The analysis results, however may be of general interest for other analyses and, hence, contribute to the common meta-model.

In summary, we need tool support for maintenance, and all maintenance tools for software analysis and refactoring need a meta-model capturing program information. Moreover, they need information extraction components creating meta-model instances, and analysis and transformation components modifying these instances. This paper describes the design of a meta-model and an architecture to construct and access instances. Furthermore, this paper describes a technique for meta-model evolution.

A. Requirements on Meta-Models

Often analysis and refactoring are defined in a source language dependent way. But, actually, they only assume that the model contains certain entities and relations, not how they are encoded in a particular source language. For example, computing the call sites of a method in an object-oriented language assumes entities like method declarations, classes, interfaces, call expressions, and static call and inheritance relations. Their encoding in any specific language is not important. If the meta-model abstracted from these language specific details, this and other analyses could be reused for different source languages. Hence, in order to increase reuse of maintenance components we require *language transparency* for our meta-model.

Basically, any maintenance tool contains a meta-model that captures the information relevant for its set of analyses, refactorings, and front-ends. This set could change and, as a consequence, the relevant information changes, as well. Hence, our next major requirement is that the meta-model architecture should be efficiently *extensible* with new analysis-, refactoring- and front-end-components.

Our final major requirement is *scalable performance*: especially large systems need maintenance tool support and these tools are often part of an edit-compile-cycle. In a straightforward implementation, a model of the program is created, which captures the whole software system. This model is used in analyses and refactorings. After code modifications, the whole process is re-entered, including the whole model computation. This brute-force implementation is too time consuming for being appropriate in an *Interactive* Development Environment (IDE).

B. Contributions

We contribute to the state of the art of meta-model design in the following way:

a) Language transparency: We define a meta-meta model – consisting of tree grammars and relations over tree node types – for defining meta-model data-structures that, in turn, can capture models of programs. It allows us to extend meta-model data-structure *implementations* by simply extending a tree grammar or relation *specification*.

b) Extensibility: Orthogonally to model- and meta-levels, we separate (meta) models specific for certain analysis-, refactoring- and front-end-components from a common, language-independent (meta) model. Mappings between them are *specified* on meta-model level; the actual mapping implementations are *generated* automatically. This separation leads to a decoupled architecture. As a consequence, change effects are local in many cases or controllable, otherwise.

c) Performance: Orthogonally to the *language transparency* and *extensibility*, we propose a technique for the implementation of meta-models that scales in performance by using *demand-driven partial model construction* and *incremental model updates*. We almost never have to construct a whole model of a software system, but only the necessary parts of the model, demanded by a concrete analysis or refactoring. Furthermore, after source code modifications, we reuse model parts that are not invalidated by the modifications.

Finally, as a proof of concept, we present two maintenance tools that are both based on the architecture and meta-modelling proposed: VIZZANALYZER, a software analysis and visualization framework, and X-DEVELOP, a multi-language IDE.

C. Paper Outline

The remainder of the paper is structured in the following way: Section II introduces the language-independent, extensible meta-model and the architecture for constructing and accessing instances. This architecture follows, in principle a pipe and filter pattern. Section III exemplifies these concepts by proposing an initial common meta-model and a mapping from a `Java` front-end. Section IV discusses meta-model evolution and the effect of changes to existing components in the architecture. Refactorings, i.e. automated source code transformations seem to contradict the pipe and filter pattern. Therefore, Section V discusses explicitly how refactorings can be plugged in into our architecture. Section VI shows how to construct a model incrementally, only changing the parts of the model corresponding to changes in the software system. Section VII introduces the proof-of-concept implementations, i.e., two maintenance tools, VIZZANALYZER and X-DEVELOP, respectively. Section VIII relates our contributions to existing results. Finally, Section IX concludes the paper and shows directions of future work.

II. CONSTRUCTING AND CAPTURING MODELS

In this section, we introduce the architecture for extracting information from software systems, capturing it and accessing it in analysis and refactoring.

We refer to the information extracted from a program as its *model*. We clearly distinguish the *model* of a program from the *meta-model* of models. The former captures more or less abstract information of a concrete program. The latter describes all possible *models* of programs. It can be understood as the type of the models or a data structure capturing them. Finally, there is a common formalism that we will use for defining the meta-models. This common formalism, i.e., the meta-meta-model, will be *tree grammars* for the main structure and *relational algebra* for additional semantic relations.

A model is obviously needed to capture information about a concrete program, e.g., containing a method `m` that invokes a method `n`. A meta-model is required since we need to define a data structure capturing information about all admissible programs, e.g., a class for capturing methods with its name and another class for capturing caller and callee methods. The meta-meta-level is our approach to make the meta-model more flexible. Instead of coding classes for capturing program entities and their relations (meta-model), we use an additional description level for defining and generating those classes (meta-meta-model).

Orthogonally to model, meta-model, and meta-meta-model, our architecture for constructing, capturing, and accessing a model of a software system consists of four major components, cf. circles in Figure 1:

1) Different concrete information-extracting front-ends for programming languages or other program representations. They capture information about a program in a *front-end specific model*.
2) Converters mapping this front-end specific model to a language independent *common model* capturing program information relevant for later analysis and refactoring.
3) Abstractions computing *views* on the common model specific for a subset of analyses and refactorings.
4) Different, concrete *analyses* and *refactorings* accessing their respective views.

A number of *front-end specific model*s relate to one *common model*, which, in turn, can have a number of different *views*. Each view may be accessed by a number of *analyses* and *refactorings*.

We separate front-end specific, common, and view models. Mappings between the different abstract models are executed for each concrete program, i.e., on model level They are implemented on the respective data structures, i.e., on meta-model level. In many cases, these mapping implementations are not programmed directly but generated from mapping specifications on meta-meta level.

All mappings can be understood as program transformations: the source code is transformed into an intermediate representations, the front-end specific model, which, in turn, is transformed into the common model and analysis- and refactoring-specific models. Such program transformations are well known in the field of compiler construction. In fact, our front-ends for different programming languages are identical to the corresponding components in compilers. Program transformations in compilers are often generated from specifications instead of being programmed by hand. In order to exploit generator technology from compiler construction, we inherit the description formalism from this field namely the context free (tree) grammars and semantic relations. Using grammars as meta-models – instead of alternatives like UML

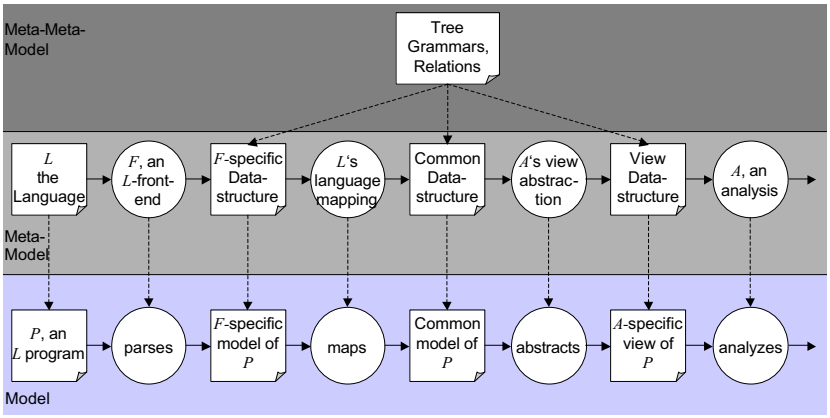

Fig. 1. Information extraction, mapping to common representation, view abstraction, and analysis – displayed on model- and meta-model-level.

diagrams, XML schemata, or database schemata – has also been proposed by Lämmel and Favre [6], [7]. Model mapping techniques are also used by other software tools, for example TXL [19], [20]. TXL is a generalized source-to-source translation system. It automatically parses inputs in the language described by a user-defined grammar and then successively applies user-defined transformation rules to the parsed input, producing as output the transformed source.

A. Front-end Specific Meta-Model

Each *front-end* understands a specific program representation, e.g., a programming or specification language. It provides a front-end specific model of that program representation. In general, such a model consists of entities representing concrete program entities and relations between them representing syntactic and semantic program relations.

Often we cannot influence the front-end F specific representations, i.e., meta-models $\mathcal{M}^F$. For a first try, we assume that they can be described in the following way: a front-end F specific meta-model is denoted by

$$\mathcal{M}^F = (\mathcal{G}^F, \mathcal{R}^F).$$

$\mathcal{G}^F$ is a tree grammar specifying the set of model entities and their structural containment. $\mathcal{R}^F$ is a set of semantic relations over model entities. Formally,

$$\mathcal{G}^F = (T^F, P^F, prog^F)$$

with T^F the set of model entities (node types), P^F a set of EBNF-productions defining containment tree structures, and $prog^F \in T^F$ the root type of the structural containment trees. EBNF-productions $p \in P^F$ have the form:

$$t ::= expr$$

where $t \in T^F$, and $expr$ is a regular expression over $T \subseteq T^F$. Expression are either sequences $(t_1 \dots t_k)$, iterations (t^*), or alternatives $(t_1 | \dots | t_k)$ with the obvious semantics.

$\mathcal{R}^F$ denotes a set of semantic relations over model entities:

$$\mathcal{R}^F = \{R_1^F, \dots, R_n^F\}$$

and each $R_i^F, 1 \leq i \leq n$ is defined over subsets of entities T^F, i.e., $T_1 \times \dots \times T_k$, $T_j \subseteq T^F, 1 \leq j \leq k$.

Example 1: Let $\mathcal{M}^{\mathtt{J}} = (\mathcal{G}^{\mathtt{J}}, \mathcal{R}^{\mathtt{J}})$ be the meta-model of a `Java` front-end `J`, which we will use as a running example. $\mathcal{G}^{\mathtt{J}} = (T^{\mathtt{J}}, P^{\mathtt{J}}, Program^{\mathtt{J}})$ defines `Java`'s Abstract Syntax Trees (ASTs). It contains, among others, entities for class $ClassD^{\mathtt{J}}$, interface $IntD^{\mathtt{J}}$, method declarations $MethodD^{\mathtt{J}}$, and call expressions $CallExpr^{\mathtt{J}}$. Productions $P^{\mathtt{J}}$ define the structural containment in ASTs. $\mathcal{R}^{\mathtt{J}}$ contains "extends" relations on classes and interfaces: $ext_c : ClassD^{\mathtt{J}} \times ClassD^{\mathtt{J}}$ and $ext_i : IntD^{\mathtt{J}} \times IntD^{\mathtt{J}}$. Also, $\mathcal{R}^{\mathtt{J}}$ contains "implements" relations $imp : ClassD^{\mathtt{J}} \times IntD^{\mathtt{J}}$ and "call" relations $call^{\mathtt{J}} : CallExp^{\mathtt{J}} \times MethodD^{\mathtt{J}}$.

Remark: In practice, we assume to reuse existing compiler front-ends, since they are existing, complete, and tested. We assume that the existing compiler front-ends do capture *all* the details that one might need downstream in our processing.

As mentioned before, the meta-model *implementation* in the front-ends may be not under our control. However, since compiler architectures and data-structures are well-established the above *modelling* is valid. Under the assumption that designers of front-ends follows standard compiler architectures and data-structures, deviations of an existing and the described meta-model could be adapted with minor implementation effort. $\diamond$

B. Common Meta-Model

The *common meta-model* $\mathcal{M}$ abstracts from front-end specific details. As argued before, it is not static but evolves on introduction of new analyses, refactorings, and front-ends, cf. Section IV. However, at any point in time, it can be defined as a pair of a tree grammar for entities and their structural containment and a set of semantic relations. Hence, we may use the same describing formalism, i.e., meta-meta-model, for defining the common meta-model at any point in its evolution. We denote the common meta-model by

$$\mathcal{M} = (\mathcal{G}, \mathcal{R}),$$

i.e., we skip the index for the specific front-end. Apart from that, a common meta-model and a front-end-specific meta-model look exactly the same.

Example 2: Let $\mathcal{M} = (\mathcal{G}, \mathcal{R})$ be our common meta-model at a certain point in evolution and let $\mathcal{G} =$

$(T, P, Program)$. Assume that T only defined node types for the whole program, and type and method declarations, i.e., $Program, TypeD, MethodD \in T$. The containment structure is defined by P:

$$\begin{aligned} Program &::= TypeD^* \\ TypeD &::= MethodD^* \end{aligned}$$

$\mathcal{R}$ defines an inheritance relation $inh : TypeD \times TypeD$ and a call relation $call : MethodD \times MethodD$.

C. Mappings

As mentioned before, the common meta-model is an abstraction of several front-end specific meta-models. For each front-end F, this abstraction is called the *front-end mapping* α^F. It is defined by mapping front-end specific grammars $\mathcal{G}^F$ to the common meta-model grammar $\mathcal{G}$ and front-end specific relations $\mathcal{R}^F$ to the common meta-model relations $\mathcal{R}$.

For the grammars, the front-end mapping α^F is defined by mapping front-end specific to common model entities:

$$\alpha^F : T^F \rightarrow T.$$

The front-end specific program node types $prog^F$ are always mapped to the common program node type $prog$, i.e., $\alpha^F(prog^F) = prog$. For the front-end mapping of relations, we define the mapping of individual relations:

$$\alpha^F : \mathcal{R}^F \rightarrow \mathcal{R}.$$

In general, we don't require α^F to be:

- *surjective*, i.e., some common meta-model types and relations do not correspond to front-end specific meta-model types and relations, nor
- *complete*, i.e., some language specific meta-models types and relations may be ignored.

Surjectiveness would imply that every front-end must at least provide the information, which the common meta-model is able to capture. This is unnecessarily restrictive. If an analysis needs information that a particular front-end cannot provide (but others can), this front-end is not applicable (with some others the analysis works fine).

Completeness would imply that there is basically no abstraction from the front-end specific to the common meta-model (just renaming of types and relations). This would lead to unnecessary efforts in plugging in very rich and detailed front-ends, even if this detailed information is never used in analysis.

Example 3: Our front-end mapping α^{J} maps node types and relations of $\mathcal{M}^{\text{J}}$ sketched in Example 1 to the common meta-model $\mathcal{M}$ sketched in Example 2. For the node types:

$$\begin{aligned} \alpha^{\text{J}}(Program^{\text{J}}) &= Program \\ \alpha^{\text{J}}(ClassD^{\text{J}}) &= TypeD \\ \alpha^{\text{J}}(IntD^{\text{J}}) &= TypeD \\ \alpha^{\text{J}}(MethodD^{\text{J}}) &= MethodD \end{aligned}$$

and for the relations:

$$\begin{aligned} \alpha^{\text{J}}(ext_c^{\text{J}}) &= inh \\ \alpha^{\text{J}}(ext_i^{\text{J}}) &= inh \\ \alpha^{\text{J}}(imp^{\text{J}}) &= inh \\ \alpha^{\text{J}}(call^{\text{J}}) &= call \end{aligned}$$

For other types and relations of $\mathcal{M}^{\text{J}}$, a front-end mapping is not defined.

The front-end mapping is specified on meta-model level and implies a mapping for concrete model instances in the following way: first, the front-end specific structural containment tree is mapped to the corresponding common structure. Then the mapped semantic relations are attached to the common structure.

The mapping of the containment trees is defined recursively: starting with the root, we traverse the front-end specific containment tree in depth-first order. We create new common model nodes of types with a mapping defined in α^F – we call those nodes *relevant*. The other, *irrelevant* nodes are ignored.

Proc. 1 *generateTreeEvents*($n =< id, t^F >$)

```
call startNode(n)
for each c ∈ childrenOf(n) do
  call generateTreeEvents(c)
end for
call finishNode(n)
```

Proc. 2 *startNode*($n =< id, t^F >$)

```
if α^F(t^F) is defined then
  create new node n' :=< newId, α^F(t^F) >
  append n' to children of Stack.top
  Stack.push(n')
end if
map(n) := Stack.top   // target of n's closest relevant ancestor
```

Proc. 3 *finishNode*($n =< id, t^F >$)

```
if α^F(t^F) is defined then
  Stack.pop()
end if
```

A generic event-based interface between front-end specific and common meta-models and an abstract algorithm for mapping the actual model instances is given in Procedures 1– 4: A *tree-walker*, cf. Procedure 1, initially called with the root node of the front-end specific model, traverses the containment tree in depth-first order and generates *startNode*-events on traversal downwards and *finishNode*-events on traversal upwards, respectively. Nodes of the structural containment tree are pairs $< id, t >$ with id and $t \in T$ the nodes' key identifer and type, respectively.

The common model data structure is created by the corresponding event-listener, *startNode*, cf. Procedure 2, and *finishNode*, cf. Procedure 3. They preserve the tree structure, but filter out irrelevant nodes.

A front-end specific relation is a set of tuples $R^F(n_1, \ldots, n_k)$ over containment tree nodes. For constructing the common model, we ignore those relations that are not mapped by α^F; we just consider the relations for which such a mapping is defined. Let $R^F : T_1^F \times \ldots \times T_k^F \in \mathcal{R}^F$ be such a relation with front-end mapping $\alpha^F(R^F) = R$. Assume each type in each T_i^F was mapped by α^F, as well. Then each node in $R^F(n_1, \ldots, n_k)$ would have a correspondence in the common model; R could simply be defined over those nodes. However, if α^F was not defined for a type of a node n_i in $R^F(n_1, \ldots, n_k)$, we would "lift" the relation to n_i's closest relevant ancestor. That is the node in the common model corresponding to the closest transitive parent of n_i, which is relevant. It is captured by $map(n_i)$ defined in Procedures 2 and used in Procedures 5.

Proc. 4 *generateRelationEvents*($\mathcal{R}^F$)

```
for each R_i^F ∈ R^F do
  for each (n_1, ..., n_k) ∈ R_i^F do
    call newRelationTuple(R^F(n_1, ..., n_k))
  end for
end for
```

Proc. 5 *newRelationTuple*($R^F(n_1, \ldots, n_k)$)

```
if α^F(R^F) is defined then
  for each n_i ∈ (n_1, ..., n_k) do
    r_i := map(n_i)   // map defined in Proc. 2
  end for
  add tuple (r_1, ..., r_k) to relation α^F(R^F)
end if
```

The mapping of a front-end specific to a common relation is done in a second phase using the event generator *generateRelationEvents*, cf. Procedure 4, and the corresponding event listener *newRelationTuple*, cf. Procedure 5.

Remark: Note, that the *abstract event generation* (algorithm schema) and the event handlers work independently of different concrete front-ends, languages, the front-end mappings, and a current common meta-model. The abstract event generation and the event handling do not change when any of these components changes. However, a *concrete implementation* of the abstract event generation side, i.e., the implementation of Procedures 1 and 4, is front-end specific and must obey the front-end specific meta-model APIs, cf. discussion in Remark II-A.

Note, that we need not to map *all* constructs and relations that a front-end provides also to the common model. We do that lazily on demand of analyses, cf. Sections IV and VI. $\diamond$

D. View Meta-Models and Analysis

The common model is the data repository for program analysis. Obviously, an analysis can only performed correctly, if the common model provides enough information. Only then can the analysis be implemented in a language-independent way.

To avoid that the common meta-model (and the front-ends) have to provide too many language specific concepts, we could, alternatively, decide to leave a certain analysis language specific and just deliver its result as a common relation. For instance, the method call target resolution is based on language specific scoping rules. Instead of representing the (intermediate) concept of a scope in the common model and implement the resolution on the common model, we may decide to resolve call targets in the language specific front-ends. The common model only needs to represent the call relation as a results of this analysis. In general, we can always fall back to a language dependent model as a source of information and basis for specific analyses.

Some analyses are dependent on language properties not provided by all languages. That is not a problem at all; it simply means that not all analyses are applicable to all programs.

Analyses might directly traverse that model, extract required information, and perform computations. However, we introduce analysis specific *views* on the common model further abstracting from the common model and providing exactly the information required by that analysis. Several analyses could share a view, and, hence, a view factoring out this information is useful to have.

Views are further abstractions of the common model. Formally, a view meta-model specific for an analysis A is described as

$$\mathcal{V}^A = (\mathcal{G}^A, \mathcal{R}^A).$$

$\mathcal{G}^A$ is a tree grammar specifying the set of view entities and their structural containment required by A. $\mathcal{R}^A$ is a set of semantic relations over view entities required by A. Again, we use the same description framework for defining a view meta-model as before: the front-end specific, the common, and the view meta-model are all defined with the same meta-meta-models: tree grammar and relational algebra.

View model construction follows the same principles as the abstractions from front-end specific to common models: we ignore some entity types, which leads to filtering of the corresponding nodes. We propagate relevant descendants of filtered nodes to their relevant ancestors by adding them as direct children. Moreover, we ignore some relation types and attach remaining relations defined over filtered nodes to the relevant ancestors of those nodes.

Like in our mapping from front-end specific to common models, construction of a view is defined using a mapping specification α^A. In contrast to the mapping from front-end specific to common models, α^A is not front-end specific, but specific for a set of analyses A. However, the same definitions for the actual model to view mapping apply.

Even for computing a specific view on a common model instance, we reuse the event-based architecture (and implementation). Handlers for tree- and relation-events are responsible for constructing the view's tree-structure and for adding relevant relations, respectively. The event-generator consists of the same phases as before: tree walker and relation generator. Since the event source, i.e., the common model data-structure, is part of our system (as opposed to the front-end specific model data-structure), we can even define the event-generator side *implementation* (as opposed to the only abstract algorithm

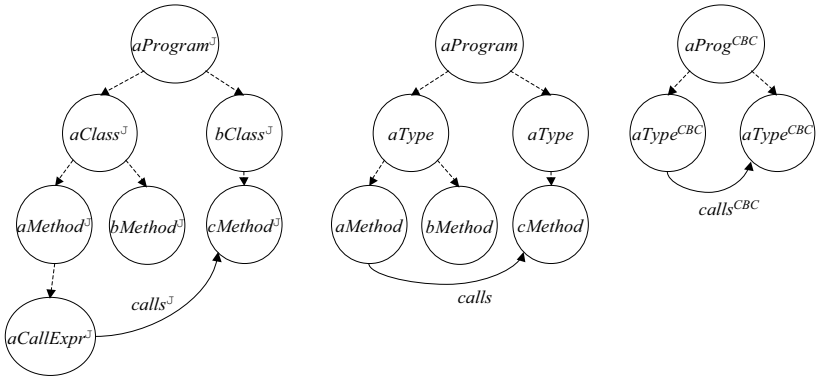

Fig. 2. A front-end specific, a common, and a view model corresponding to Examples 1- 4.

schema generating events for the front-end specific models).

Finally, analysis accesses the corresponding view and performs computations. We deliberately skip a discussion on how to store and display analysis results and refer to [8] instead.

Example 4: Let CBC be a metric analysis computing the relative coupling of a class via method calls:

$$CBC(c) :=| \; calls(c, c') \mid / \mid calls(c, \_) \mid \;\; where \; c \neq c'.$$

An appropriate view meta-model $\mathcal{V}^{CBC}$ would define the entity types for type declarations $TypeD^{CBC}$ and a call relation $call^{CBC}$. As opposed to the call relation of the common meta-model, cf. Example 2, this $call^{CBC}$ is defined over class declaration nodes $call^{CBC} : TypeD^{CBC} \times TypeD^{CBC}$.

Our mapping α^{CBC} defines a view meta-model for the common meta-model $\mathcal{M}$ sketched in Example 2, i.e., for the node types

$$\begin{aligned} \alpha^{CBC}(Program) &= Prog^{CBC} \\ \alpha^{CBC}(TypeD) &= TypeD^{CBC} \end{aligned}$$

and for the relations

$$\alpha^{CBC}(calls) = calls^{CBC}.$$

Hence, nodes of the method declaration type and inheritance relations are filtered. Figure 2 shows an example mapping of a front-end specific to a common model, and further to a CBC view model.

III. Example Meta-Model and Front-end

In this section we, present a Common Meta-Model (CMM) and a mapping from a `Java` front-end as an extended example of the ideas discussed so far. We base CMM on the Dagstuhl Middle Meta-Model 0.007 (DMM) [9], a commonly accepted meta-model for program analysis. Originally, DMM was defined in a UML-like notation. Here, we use our model description formalism instead.

A. Dagstuhl Middle Meta-Model

DMM provides more elements and relations than needed for the `Java` front-end. These (so far) unnecessary elements and relations are marked with wavy underline text (text) in Tables I, II, and III. Additionally, some relevant elements and relations, not contained in the original DMM, were captured in CMM. They are marked with underlined text (text).

Model	::=	ASourceObject* AModelObject*
ASourceObject	≻	ASourcePart | ASourceUnit
ASourcePart	≻	SourcePart | MacroExpansion | MacroArgument | ADefinition | AResolvable | Comment
ADefinition	≻	MacroDefinition | Definition
AResolvable	≻	Declaration | Reference | Resolvable
ASourceUnit	≻	SourceFile | SourceUnit
AModelObject	≻	ModelObject | AModelElement
AModelElement	≻	ModelElement | Package | AStructuralElement | ABehaviouralElement | CompilationUnit
AStructuralElement	≻	AType | AValue
AType	≻	Type | AStructuredType | CollectionType | EnumeratedType
AStructuredType	≻	StructuredType | Class
Class	::=	Field* Method* Initializer*
Class	≻	AbstractClass | AnonymousClass
AValue	≻	Value | EnumerationLiteral | AVariable
AVariable	≻	Variable | Field | FormalParameter
EnumerationType	::=	EnumerationLiteral*
ABehaviouralElement	≻	ExecutableValue | Method | Routine | Constructor
Method	::=	FormalParameter*
ExecutableValue	::=	FormalParameter*
Routine	::=	FormalParameter*

TABLE I
DAGSTUHL MIDDLE METAMODEL, PRODUCTIONS P AND SPECIALIZATION HIERARCHY

The root node type of G is *Model*. The productions P of G describe a structural containment relation, denoted by ::= in the productions, and a specialization hierarchy on the meta-model entities T, denoted by ≻. DMM is shown in Table I, II, and III. The structural containment relation is added to the original DMM. In contrast to DMM, we separate abstract and concrete meta-model elements: abstract model elements, denoted by "A<*entity name*>", are never instantiated in a concrete model.

Attributes of DMM entities and binary relations are defined in R as unary and binary relations, respectively. Unary relations include: isSubclassable(Class), size(CollectionType), name(FormalParameter | MacroArgument | MacroDefinition | ModelObject | AResolvable | ASourceUnit), visibility(Field | Method | ModelElement), path(SourceFile), position(FormalParameter), isConstructor(Method), isDestructor(Method), isAbstract(Method), isDy-

Accesses	:	ABehaviouralElement × AStructuralElement
Contains	:	SourceObject × SourcePart, Package × ModelElement
Declares	:	SourceObject × ModelObject
Defines	:	SourceObject × ModelObject
Describes	:	SourceObject × Comment
HasValue	:	Variable × Value
Imports	:	Class × Package
Includes	:	SourceFile × SourceFile
InheritsFrom	:	Class × Class
Invokes	:	ABehaviouralElement × ABehaviouralElement
IsActualParameterOf	:	ModelElement × Invokes
IsConstructorOf	:	Class × Constructor
IsDefinedInTermsOf	:	Type × Type
IsEnumerationLiteralOf	:	EnumerationLiteral × EnumeratedType
IsExpansionOf	:	MacroDefinition × MacroExpansion
IsFieldOf	:	Field × StructuredType
IsMethodOf	:	Method × Class
IsOfType	:	AValue × Type
IsParameterOf	:	FormalParameter × BehaviouralElement
IsReturnTypeOf	:	Type × BehavioralElement
IsSubpackageOf	:	Package × Package

TABLE II

DAGSTUHL MIDDLE METAMODEL, BINARY SEMANTIC RELATIONS IN R

$\text{program}^{\text{R}}$	::=	$\text{compilation\_unit}^{\text{R}*}$
$\text{compilation\_unit}^{\text{R}}$	::=	$\text{type}^{\text{R}+}$
type^{R}	$\succ$	$\text{class}^{\text{R}*}$ | $\text{interface}^{\text{R}*}$
$\text{interface}^{\text{R}}$	::=	$\text{method}^{\text{R}*}$
class^{R}	::=	$\text{constructor}^{\text{R}*}$ $\text{method}^{\text{R}*}$ $\text{field}^{\text{R}*}$ $\text{initialization\_block}^{\text{R}}$?
method^{R}	::=	$\text{formal\_param}^{\text{R}*}$ $\text{statement}^{\text{R}*}$
$\text{statement}^{\text{R}}$	$\succ$	assign^{R} | $\text{call\_expr}^{\text{R}}$ | $\text{create\_expr}^{\text{R}}$ | do^{R} | for^{R} | if^{R} | switch^{R} | while^{R}
$\text{initialization\_block}^{\text{R}}$	::=	$\text{statement}^{\text{R}*}$
$\text{constructor}^{\text{R}}$	::=	$\text{statement}^{\text{R}*}$
do^{R}	::=	$\text{expression}^{\text{R}}$ $\text{statement}^{\text{R}*}$
for^{R}	::=	$\text{expression}^{\text{R}}$ $\text{expression}^{\text{R}}$ $\text{expression}^{\text{R}}$ $\text{statement}^{\text{R}*}$
if^{R}	::=	$\text{expression}^{\text{R}}$ $\text{statement}^{\text{R}*}$ $\text{statement}^{\text{R}*}$
switch^{R}	::=	$\text{expression}^{\text{R}}$ ($\text{expression}^{\text{R}}$ $\text{statement}^{\text{R}*}$)*
while^{R}	::=	$\text{expression}^{\text{R}}$ $\text{statement}^{\text{R}*}$
assign^{R}	::=	$\text{expression}^{\text{R}}$ $\text{expression}^{\text{R}}$
$\text{expression}^{\text{R}}$	$\succ$	$\text{call\_expr}^{\text{R}}$ | $\text{create\_expr}^{\text{R}}$ | $\text{read\_expr}^{\text{R}}$ | $\text{write\_expr}^{\text{R}}$ | ...
$\text{call\_expr}^{\text{R}}$	::=	$\text{expression}^{\text{R}*}$ -- *designator, actual parameters*
$\text{create\_expr}^{\text{R}}$	::=	$\text{expression}^{\text{R}*}$ -- *actual parameters*

TABLE IV

RECODER FRONT-END, PRODUCTIONS P^{R}

contains	$\succ$	containsField, containsMethod, containsConstructor
Relationship	$\succ$	ModelRelationship, SourceRelationship, SourceModelRelationship
ModelRelationship	$\succ$	InheritsFrom, IsPartOf, Invokes, IsOfType, Accessess, IsDefinedInTermsOf, IsPartOfSignatureOf, IsActualParameterOf, HasValue, IsSubpackageOf
InheritsFrom	$\succ$	Extends, Implements
Invokes	$\succ$	InvokesConstructor, InvokesSuperConstructor, InvokesSuper
IsPartOf	$\succ$	IsEnumerationLiteralOf, IsMethodOf, IsFieldOf, IsConstructorOf
IsPartOfSignatureOf	$\succ$	IsParameterOf, IsReturnTypeOf

TABLE III

DAGSTUHL MIDDLE METAMODEL, SPECIALIZATION HIERARCHY $\prec$ OF SEMANTIC RELATIONS

call^{R}	:	$\text{call\_expr}^{\text{R}} \times \text{method}^{\text{R}}$
create^{R}	:	$\text{create\_expr}^{\text{R}} \times \text{method}^{\text{R}}$
$\text{extends}^{\text{R}}$	:	$\text{interface}^{\text{R}} \times \text{interface}^{\text{R}}$
$\text{extends}^{\text{R}}$	:	$\text{class}^{\text{R}} \times \text{class}^{\text{R}}$
$\text{implements}^{\text{R}}$	:	$\text{class}^{\text{R}} \times \text{interface}^{\text{R}}$
$\text{field\_access}^{\text{R}}$	:	$\text{read\_expr}^{\text{R}} \times \text{field}^{\text{R}}$
$\text{field\_access}^{\text{R}}$	:	$\text{write\_expr}^{\text{R}} \times \text{field}^{\text{R}}$
$\text{type\_ref}^{\text{R}}$	:	$\text{expression}^{\text{R}} \times \text{type}^{\text{R}}$

TABLE V

RECODER FRONT-END, SEMANTIC RELATIONS R^{R}

namicallyBound(Method), isOverrideable(Method), visibility(Field | Method | ModelElement), path(SourceFile), startLine(SourcePart), start(SourcePart), endLine(SourcePart), and endChar(SourcePart). Binary relations and their specialization $prec$ are listed in Tables II and III, respectively.

B. Java Specific Model for RECODER Front-end

This section describes a language-specific representation of the programs that the metrics are applied to. Therefore, we use a meta-model of the compiler front-end RECODER [10] denoted by $M^{\text{R}} = (G^{\text{R}}, R^{\text{R}})$.

The model entities T^{R} of the grammar G^{R} are implicitly defined by the productions P^{R}, which in turn define the containment structure of the model entities and their specialization hierarchy, cf. Table IV. The root node type of G^{R} is *program*. Unary semantic relations (intrinsic node attributes like names, types, positions, and visibility) are similar to DMM and therefore omitted. Binary semantic relations R^{R} are shown in Table V. In the RECODER meta-model, there is no particular type hierarchy $\prec^{\text{R}}$ for relations defined.

The mapping functions for mapping the language-specific types T^{R} and relations R^{R} to the common meta-model are given in Table VI. The mapping of actual models filters nodes of types t^{R} with $\alpha(t^{\text{R}})$ not defined. Otherwise it keeps the containment structure of the language-specific model in the common model, but some transitive children in the language-specific model become direct children in the common model. Remember, tuples of the language-specific $rel^{\text{R}} : t_1^{\text{R}} \times t_2^{\text{R}}$ are mapped to the corresponding tuple of $\alpha^{\text{R}}(rel^{\text{R}}) : t_1 \times t_2$. Furthermore, if nodes of a source tuple are filtered, the target tuple nodes become the next suiting parent in the target containment structure.

For example, $\alpha^{\text{R}}(call^{\text{R}} : call\_expr^{\text{R}} \times method^{\text{R}}) \mapsto Invokes : ABehaviouralElement \times ABehaviouralElement$, and $call^{\text{R}}$ nodes are filtered. Hence, a source tuple $\text{call}^{\text{R}}(aCallExpr, aMethod)$ with $aCallExpr$ contained in a method $anotherMethod$ of the language-specific model is mapped to a tuple $Invokes(anotherMethod', aMethod')$ with $anotherMethod'$ and $aMethod'$ the caller and callee, respectively, in the common model.

An extended description of this example can be fond in [11].

IV. META-MODEL EVOLUTION

The initial common and view meta-models are usually designed to be suitable for a set of information sources, analyses and refactorings. However, when a new information source or a new analysis or refactoring is added, the meta-models could change, as well. Now we discuss, which kind of changes remain local, and, which changes may have global effects.

Trivially, new analyses and refactorings relying only on information already provided by an existing view do not

α^{R}(program^{R})	$\mapsto$	Model
α^{R}(compilation_unit^{R})	$\mapsto$	SourceUnit
α^{R}(class^{R})	$\mapsto$	Class
α^{R}(interface^{R})	$\mapsto$	Class
α^{R}(constructor^{R})	$\mapsto$	Method, Method.isConstructor = true
α^{R}(method^{R})	$\mapsto$	Method, Method.isConstructor = false
α^{R}(field ^{R})	$\mapsto$	Field
α^{R}(initialization_block ^{R})	$\mapsto$	Method, Method.isConstructor = true
α^{R}(call^{R})	$\mapsto$	Invokes
α^{R}(create^{R})	$\mapsto$	Invokes
α^{R}(extends^{R})	$\mapsto$	InheritsFrom, InhertisFrom.inheritanceType = Extends
α^{R}(implements^{R})	$\mapsto$	InheritsFrom, InhertisFrom.inheritanceType = Implements
α^{R}(field_access^{R})	$\mapsto$	Accesses
α^{R}(type_ref^{R})	$\mapsto$	IsOfType

TABLE VI
MAPPING FUNCTIONS α^{R}

trigger changes in the meta-models.

Assume analysis A (w.l.o.g., we do not explicitly discuss refactoring in the following) cannot be applied on any of the existing view meta-models, but, the required information is encoded in the common meta-model already. Then a new view meta-model $\mathcal{V}^{A}$ and a new view mapping α^{A} from the common to the view meta-model are to specify. There is no additional implementation effort since event-generator and -handlers for creating the actual views are generated automatically, cf. Programs 1– 5.

In general, a new analysis also requires an extension of the common meta-model, which, in turn, implies that common model creation is affected, as well. Either the frond-end(s) are able to provide this new piece of information, just that it has not been considered relevant so far, or, a new front-end needs to be integrated. In both cases, we need to extend the common meta-model $\mathcal{M}$ and the front-end F specific mapping α^{F}. Given the tree- and relation event generators work according to our schemata in Programs 1 and 4, no additional programming is needed when reusing an existing front-end. Then we just specify the missing entities and relations as relevant in α^{F}, and common model creation is generated automatically. New front-ends obviously require specific implementations of the Program 1 and 4 schemata.

However, by changing the common meta-model, and thereby relevant types and relations, we could even run into a reuse problem: if formerly irrelevant nodes get relevant, mappings may create relations that are not well-typed any longer. Practically, this would mean that a relation that used to be attached to one node type is now attached to a descendant of that type. This, in turn, could lead to situations where analyses cannot work as before, i.e., cannot be reused without adaptation. Fortunately, the effect of changes in the common meta-model is often not visible in the existing view-model, and, hence, many analyses can be applied without changes. Criteria for safe extensions, i.e. extensions that are guaranteed not to have effects on existing view-models, are discussed below.

Example 5: Given the common meta-model of Example 2 containing type and method declarations, inheritance relations on type declarations, and call relations on method declarations. Additionally, we assume the view meta-model of Example 4 for computing the coupling between classes CBC. Now we are to add a new (complexity) analysis that counts *statements*. This leads to the following changes: block and statement nodes are introduced in the common meta-model; its grammar productions change accordingly:

$$\begin{aligned} Program &::= TypeD^{*} \\ TypeD &::= MethodD^{*} \\ MethodD &::= Block \\ Block &::= Statement^{*} \\ Statement &::= Assign|If|Loop \\ If &::= Block_{then}\ Block_{else} \\ Loop &::= Block_{body} \end{aligned}$$

The original call relation $call : MethodD \times MethodD$ changes to $call : Statement \times MethodD$ since statements are the relevant ancestors of call expressions in the meta-model now. However, when applying α^{CBC} from Example 4 to this new meta-model, $\mathcal{V}^{CBC}$ remains unchanged, i.e., the view grammar is still

$$Prog^{CBC} ::= TypeD^{CBC\ *}$$

and the call relation is $call^{CBC} : TypeD^{CBC} \times TypeD^{CBC}$, as before. This is because type declarations are the relevant ancestors of statements in the view mapping. Hence, the old CBC analysis is applicable without any change.

What we learned from the above example is that many effects of changing the common meta-model are removed from subsequent view abstractions. This comes actually at no surprise, since the view mapping is defined by explicitly declaring relevant types; newly introduced types would not be declared relevant in existing view mapping specifications. As long as changes just extend the common model trees, view mappings would compensate and produce the original views for the existing analyses.

In general, a changed common meta-model could change a view meta-model, which would affect existing analyses. However, there are *safe* changes to the common meta-model guaranteed not to affect an analysis A:

- Adding a *new* type to a sequence expression on the right-hand side of a production.
- Adding an *existing* type t to a sequence if no other type relevant for A can transitively be derived from t.
- Introducing a new production $t ::= \ldots$ if no type relevant for A can transitively be derived from t.
- Adding a new relation.

In all these cases, the nodes newly introduced to a common model will be filtered by existing view mappings and the relations will be attached to the original node types. Conversely, if a meta-model change is not safe for an analysis A, we should check and potentially adapt A.

Obviously, we have to produce a first common meta-model in the series, computing the essential structural and relational information. This initial meta-model, e.g., the Common Meta-Model as defined in Section III, should be carefully designed

with a first set of analyses and refactorings in mind. This meta-model evolves on demand of new analyses and refactorings.

As one builds more dialects into the front-ends, and has to carry more and more nodes through the intermediate representations, there is a threat that they might finally end up as a kitchen sink with everything included. This again cannot be avoided automatically, Instead it requires careful design when extending the architecture. New front-ends should reuse existing program element and relation types if possible in order not to end up with many constructs implementing the same or similar concepts.

V. Refactorings

This section outlines how our architecture and the information represented in the view models can be used to implement even refactorings. This seems to be difficult since the architecture presented so far follows the pipe and filter pattern. Each transformation step from program to view abstracts from information and is, hence, not invertible. Refactorings on the other hand require, by definition, to modify the original program.

Regarding the information required from a program, refactoring and analysis can be treated identically as described in Section II-C. They are both defined on views of the common meta-model. Consequently, if a source code entity is to modify in a refactoring R, we postulate the same restrictions on the common and view meta-models $\mathcal{M}$ and $\mathcal{V}$, respectively, and the front-end and view mappings α^F and α^R, respectively, as for an analysis. Trivially, those source code entities must not be filtered in neither α^F nor α^R. We cannot, e.g., rename a method if method declaration entities are removed (and their positions with them). Moreover, we need relations used in refactoring not to be abstracted either, i.e., they must not be attached to transitively relevant descendants. We cannot, e.g., find and rename the call sites of the renamed method declaration if the call relation is abstracted to class level.

If the information is not sufficient, the view and potentially the common meta-model need to be extended (including an extension of the corresponding front-end). If that's not possible, e.g., because the required information is not available in a front-end-specific meta-model, the corresponding analysis or refactoring is not applicable.

There may be highly language specific refactorings – e.g., change `int x[]` to `int[] x` in all places in a `Java` program – where it may not be worth the effort to support the required information in the common model. It is quite unlikely that we want to support such inherently language specific refactorings for other languages, as well.

The supported language-independent refactorings depend only on the complexity of the common model. In principle, all refactorings can be implemented given a detailed enough meta-model. For example we could use a common meta-model that is a union of all language-specific models and contains all details. In practice, however, many refactorings can be implemented using a rather simple common model. For example, the Rename-Method-Refactoring discussed below just requires method calls and definitions to be supported by the common model.

In contrast to the requirements of analysis, a refactoring should lead to a *correct* transformation in the source language. For a first try, our notion of correctness here reduces to compilability of the transformed code. However, this leads to an additional requirement on front-ends: the static analysis in the front-ends needs to be *conservative* w.r.t. compliability. If our call relation, e.g., just contain *some* call sites (that we rename properly) while others remain unchanged, we would not be able to guarantee correctness.

The major difference to analysis is that refactoring requires to keep track of original source code entities to change. Obviously, we cannot apply the refactorings on the model and then serialize this to source representations due to the abstractions in the models. Instead, every node in the front-end specific model is annotated with information describing its textual origin, i.e., the source file and the exact position in that file, e.g., defined by byte-offsets for start and end positions. When nodes are mapped to the common model this information is preserved. We use this position information to directly modify the text of the source in refactorings. Hence, the refactorings compute actual textual transformations, in contrast to many other tools, which first compute model transformations and then "'unparse"' the transformed model back to source code. In case of a Rename-Refactoring, for example, the textual modifications comprise the identifiers in the source code that need to be changed.

After a refactoring has been applied, we might completely rebuild the front-end specific, the common, and the view models again, i.e. following the pipe and filter achitecture.

Example 6: As an example, we sketch how the *Rename Method* refactoring can be implemented using the common model, i.e., in a reusable, language independent way. This refactoring is a code transformation that changes the name of a method and consistently updates all call sites. It is more than a simple text replacement: we need to find all possible calls to the renamed method and the name may appear in completely different meanings, e.g., representing a variable name. Additionally, we need to rename other methods if the changed calls could invoke these other methods, too. Finally, there could be more methods with the same name that are actually unrelated and, hence, should not be changed.

Our common model provides information for implementing this refactoring. We make use of the common node types $Identifier$ representing identifiers in the source code, $CallExpr$ representing method call expressions and $MethodD$ representing method declarations, as well as the following common relations:

$$\begin{aligned} calls &: CallExpr \times MethodD \\ methodname &: MethodD \times Identifier \\ callname &: CallExpr \times Identifier \end{aligned}$$

Relation $methodname$ represents the identifiers of methods, $callname$ the called method name. Relation $calls$ is a conservative approximation of the dynamic call relation, i.e., $(c, m) \in calls$ iff c may call m, and $(c, m) \notin call$ iff c is definitely not a call to m. Because $calls$ includes all possible method invocations we can compute the set of calls and methods to

be changed: starting from the call sites having a method as a declared target, we compute the transitive-reflexive closure $calls^*$ of the $calls$ relation.

Then we look up the name of the method declarations and call sites in the $methodname$ and $callname$ relations, resulting in nodes of type $Identifier$. For each such identifier node n in the common model, we lookup its textual position in the source and rename it.

Following this approach, we can implement refactorings in a language-independent way using the common model. As a result the refactorings will work with any language that is supported by a front-end. Additionally, whenever a new language is added the refactoring will work with this new language, as well, without having to implement language-specific code for this particular refactoring.

For more details on refactoring, please refer to [12].

For achieving the required performance, our architecture supports incremental model updates that reuse parts of the existing model to speed up the process of rebuilding the model after refactoring. This is necessary, since maintenance tools integrate the model construction in the edit-compile cycle where program changes occur quite frequently. We describe this mechanism in Section VI below.

VI. Partial and Incremental Model Construction

Modern IDEs provide many features that rely on analysis and refactoring of the programs processed, i.e., constructing models of the programs. Examples include features like refactoring, coding assistance or code visualization. If these features are to be integrated seamlessly in the work cycle of the environment the code analysis must deliver results almost instantaneously, even when confronted with large program sizes and frequent program modifications by the user or by refactorings. We have to make sure that implementations of the meta-model meet this requirement.

We show that a brute-force implementations for model construction – computing a model of the whole system whenever the code changes – is not required. Instead, we can use partial model construction and incrementally reuse model parts after source code changes. However, the captured and recycled information highly depends on the concrete analyses and refactorings.

A. Demand-driven Partial Model Construction

When looking at how to implement concrete high-level analyses or refactorings, we found that it is rarely necessary to compute a whole model of a software system. Instead, it is sufficient to capture only a partial model, which also means that source code analysis is only required for the parts reflected in this partial model. This leads us to the questions which model parts are to be constructed in order to implement a concrete feature. The answer depends on the feature and the concrete request to the feature.

Example 7: We take the Rename refactoring as an example. As we have seen in Example 6, our common model provides information for implementing this refactoring. Basically, we look for nodes of type $Identifier$ that are part of calls which are in turn specified by the $calls^*$ relation as being calls to the renamed method. In order to achieve this, we do not have to construct the whole model, i.e. the whole $calls^*$ relation. Theoretically, we can build a subset $calls^*_S \subset calls^*$ that at least includes all relevant calls (i.e. calls to be updated). A possible criterion could be to look only at those calls whose identifier $Identifier$ equals the method name. For example when renaming a method foo we will only look for calls to methods called foo and perform a semantic analysis to determine whether they are calls to the desired foo method. To determine where to look for appropriate identifiers it is necessary to build some sort of pre-selection data structure of the source code to be analyzed. For the Rename Refactoring this can be an identifier index. This index is a map stating which identifiers occur in which compilation unit. When looking for a particular identifier, we only have to look in compilation units given by the map. Such pre-selection data structures have to be built once in an initial scan of the analyzed system. Afterwards, they can be updated incrementally.

This partial model construction can even be enhanced: as we have seen the implementation of a concrete feature mostly requires only a certain part of the system to be represented by the model. Additionally, we found, that even this part does not have to represented completely in memory at once. Instead, this model part can be further divided into sub-parts, which can be processed sequentially. This technique significantly reduces memory consumption of implementations.

The model, i.e. its entities and semantic relations, has to be constructed only for compilation units. However, this has not be to be done at the same time for all compilation units, which may contain relevant entities and relations. Instead, it is possible to analyze the compilation units sequentially and only keep the information from one compilation unit in memory at a time.

Example 8: Again, we take the Rename refactoring as an example. As we have seen in Example 7 it is possible to implement the Rename refactoring by only looking at compilation units, which may contain relevant calls.

B. Incremental Model Update

Incremental model update means that the model does not have be reconstructed completely after a source code modification. In fact, source code modifications most of the time have only a limited scope. Thus, parts of the model stay valid and can be recycled. This incremental update is also possible if the model is constructed only partially. Additionally, pre-selection data structures used for partial model construction (e.g. indexes) can be updated incrementally also.

The key problem is to determine which information to retain. To do this, we annotate the information in the model with versions and digests from the source code.

Example 9: One part of the model are syntax trees. Each compilation unit has an accompanying syntax tree. When the compilation unit changes textually, the syntax tree has potentially to be rebuild, depending of the modification. As

a criterion we compute a digest of the compilation unit, such that changes of the digest indicate changes of the syntax tree. The simplest digest is the file version of the compilation unit (assuming compilation units are technically stored on a file-system with file versions).

The same principle can be used to annotate semantic relations with source digest also.

C. Implementation Details

Even with partial model construction and incremental model reuse, there will be a short analysis time left. During this time, changes to the code can still occur and should be allowed. To achieve this, program analysis needs to run in the background. However, this means we have to ensure that analysis results stay consistent with the source code. This is achieved using source code snapshots: First, the analysis operates on a snapshot of the program, which is captured when the analysis is started. Consequently, the analysis results will only be valid for the program version as found in the snapshot. Thus, if the source code has changed the analysis has probably be rerun with the new code version, depending of the change and the concrete analysis.

VII. Proof of Concept

As a proof of concept, we discuss two software maintenance tools: VizzAnalyzer and X-develop. Both tools are implemented on the architecture discussed before. VizzAnalyzer is a program analysis tool, designed with extension towards new languages and new external software analyses and visualizations in mind. Therefore, we exemplify the extensibility of our architecture using VizzAnalyzer. X-develop is a multi-language IDE, implementing analysis and refactorings. As an interactive tool, a major design goal was scalable performance. Using X-develop, we exemplify the ability of our architecture to plug in refactorings while maintaining performance.

A. VizzAnalyzer

The VizzAnalyzer tool[1] is an instantiation of the VizzAnalyzer reverse engineering framework [8], which has been developed at Växjö university. Its flexible architecture allows to integrate tools for information extraction, analysis, and visualization. Its core architecture is an implementation of the design discussed before. It can be seen in action in Figure 3.

This section exemplifies the process of adding new analyses and front-ends to the VizzAnalyzer. It also presents some VizzAnalyzer benchmarks.

1) Adding a New Analysis: The initial meta-model was suitable for a number of object-oriented metrics. However, adding McCabe's Cyclomatic Complexity (CC) metric [13] was not possible on the initial meta-model. CC is a measure of the control-flow complexity of a method, defined as the number of linearly independent paths.

[1] `www.arisa.se`

The control structure of a method in object-oriented languages is encoded with `if`, `for`, `do`, `while`, and `switch` statements. For simplicity, exceptions, tertiary operators, and boolean expressions in the control statements are neglected. CC of a method can be computed by counting control statements contained in a method (adding one since methods without branches implement a single path).

The initial meta-model did not contain control statement entities. A method declaration was a sequence of call expressions:

$$MethodD \quad ::= \quad CallExpr^*$$

In order to calculate CC, we extend the meta-model with a new type *CStmt* and the following productions:

$$\begin{aligned} MethodD &::= (CallExpr|CStmt)^* \\ CStmt &::= (CallExpr|CStmt)^* \end{aligned}$$

The front-end mapping – initially there was only one `Java` front-end – was extended to map all `Java` control statements to *CStmt*:

$$\begin{aligned} \alpha^{\mathrm{J}}(If^{\mathrm{J}}) &= CStmt \\ \alpha^{\mathrm{J}}(For^{\mathrm{J}}) &= CStmt \\ &\ldots \end{aligned}$$

A new view mapping α^{CC} trivially defines:

$$\begin{aligned} \alpha^{CC}(Program) &= Program^{CC} \\ \alpha^{CC}(MethodD) &= MethodD^{CC} \\ \alpha^{CC}(CStmt) &= CStmt^{CC} \end{aligned}$$

On these views, Cyclomatic Complexity of a method m could simply be computed by counting the control statement descendants under m.

Altogether, we needed to add less than 100 lines of specification and `Java` code. More precisely: The walker traversing the AST in the front-end was extended to generate events for `if`, `for`, `do`, `while`, and `switch` statements. The common meta-model specification was extended by the *CStmt* type and the corresponding grammar productions. The extensions of the walker implementation and the meta-model specification generated the extended mapping of front-end specific to common models. The CC view and the mapping specification from the common meta-model to the view was defined as described above. The view specific meta-model and the mapping specification together generated the new view abstraction. Finally, the CC metric was implemented.

2) Adding a New Front-End: The initial, Recoder[2]-based front-end, cf. Section III could only handle `Java` source code. This means that the influence of external libraries provided in byte code was neglected. It is well-known that all more precise call graph construction algorithms require some sort of data-flow analysis, which, in turn, requires a whole program representation [14], [15]. To add these kind of analyses, we integrated another front-end based on the byte code reader of the Soot Framework[3]. It constructs a program representation

[2] `recoder.sourceforge.net`
[3] `www.sable.mcgill.ca/soot`

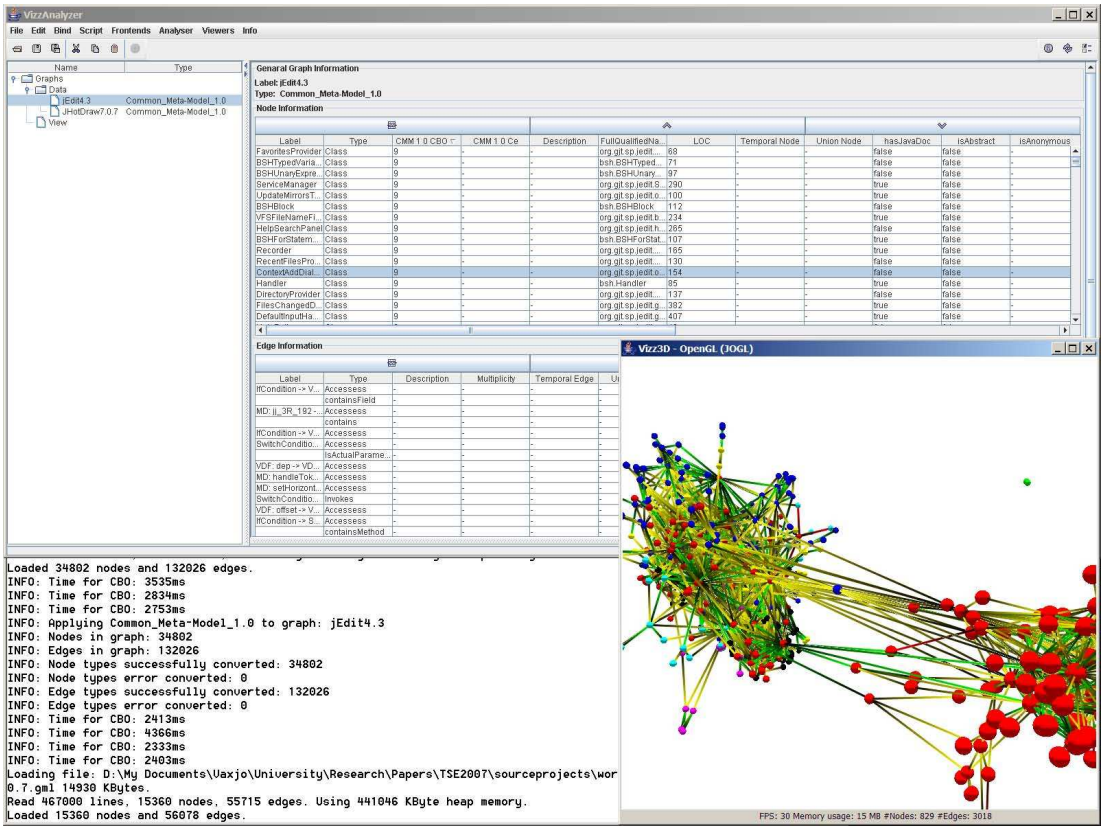

Fig. 3. Example: Metrics analysis and visualization of JEDIT and JHOTDRAW. The VizzAnalyzer (top) shows nodes and edges of JEDIT including some basic information and metrics values (CBC and LOC). Vizz3D (bottom, right) shows a 3D visualization of the call and access structure between classes of JEDIT using a force-based clustering algorithm. Coloring of nodes corresponds to packages.

	Lines of code	Model build time	Analysis time	Model size
JEDIT	145508	91s	3s	100mb
JHOTDRAW	57020	42s	<1s	54mb

TABLE VII
VIZZANALYZER'S CPU AND MEMORY REQUIREMENTS OF THE MODEL

where each method is represented by a basic block graph and each basic block contains a sequence of statements. The edges in the basic block graph represents control-flow.

The transition from the RECODER- to the SOOT-based front-end only involved adding two new node types (*BasicBlock* and *Alloc*), and two new relations (*controlFlow* and *allocates*) to the common meta-model.

All coupling and cohesion metrics previously developed for the common-model could immediately be reused. The Cyclomatic Complexity metric could however not be reused since it requires the control statement node type – a node type that our SOOT-based front-end does not generate.

Another front-end extension for analysing UML class and sequence diagrams did not even require changes to the meta-model. Only an XMI reader (XML front-end) and a XMI-front-end mapping needed to be implemented. Since the common meta-model did not change, all metrics could remain unchanged, as well.

3) Benchmarks: To show the real-world fitness of VIZZANALYZER we show how it performs on some real-world projects. The results are shown in Table VII. System configuration was: Windows XP SP2, Pentium M 1.7GHz, 1 GB RAM. All measured times are overall times, measured between invocation of the commands and retrieval of the results, including parsing, building of the representation and writing of the result into a file. The values measured are:

- Model build time, that is the time needed to build the full model from the source code. This includes reading from file, parsing and model creation. Most of the time is spent in reading and parsing in the RECODER-based front-end.
- Analysis time, that is the time needed to perform a particular analysis on the model. In this case it was the CBC metric described in Section II.
- Model size, that is the amount of memory in RAM used by the model, after it has been completely created.

B. X-develop

X-DEVELOP[4] is a commercial Integrated Development Environment (IDE) supporting multiple programming languages. Its kernel implements a common meta-model used to implement refactorings and code-analysis based tools as described in this paper.

1) Kernel: X-DEVELOP's kernel is an implementation of a common meta-model. This common meta-model is implemented as an API framework, which is used both by clients and by front-ends. Thus, the front-ends share this API not only as a common-model to store their analysis results, but they can use this API as an utility for program analysis also. This design boosts reuse of analysis functionality and simplifies the implementation of front-ends. A characteristic of X-DEVELOP's common meta-model is, that it can be used to capture whole-system models of heterogeneous software systems, which incorporate several programming languages. For more information please refer to [12].

2) Front-ends: Support for concrete languages is implemented in X-DEVELOP using language front-ends, currently for: `C#`, `Java`, `VisualBasic`, `J#`, `HTML`, `XML`, `ASP`, `JSP`, `JavaScript`.

It is the responsibility of these front-ends to capture sufficiently detailed information and store it in the common model. Our front-ends perform a complete semantic analysis - similar to the analysis done by compilers - including complete cross-reference relations. Although all details of the rather complex supported languages are implemented, the front-ends are still reasonably small, e.g., the `C#` front-end has 35932 lines of code, the Java front-end has 50542 lines of code.

3) Refactorings and Tools: The common model is used to implement concrete refactorings - e.g. rename method/class/variable, change method signature, move classes to other namespace/package, extract method, inline method - and high level tools - e.g. usage search, code completion - in a language independent way. Figure 4 shows the results of a precise search for usages of a method in X-DEVELOP.

4) Benchmarks: To show the real-world fitness of X-DEVELOP we show how it performs on some real-world projects. System configuration was: Windows XP SP2, Pentium 4 2GHz, 1 GB RAM. The results are shown in Table VIII. All measured times are overall times, measured between invocation of the commands and retrieval of the results, including parsing and building of the representation. As outlined in Section VI the work-cycle of our model is governed by incremental model updates and partial model construction. We have to take this into account when doing benchmarks. The times measured are:

- The time for the initial analysis, starting with an empty model, i.e. without incremental model update. The task of the initial analysis is to build indexes and to do a complete semantic analysis in order to find potential errors in the code. Note, that after the initial analysis a complete build is never really required in practice, because all of our features are implemented using partial model construction for relevant program parts only. However, this time is interesting for comparing the speed of the data flow analysis algorithm.
- The time required to perform a Rename Refactoring [5], i.e. the time to build a partial model, starting with an empty model, i.e. without incremental model update. The model only needs to be build as far as required by the concrete refactoring request. In theory, this could involve the whole model. In practice, we found that usually only a small part of the model needs to be built. Depending of the size of this part the time required for the computation is only a few seconds.
- The time required to build a partial model for the Rename Refactoring not starting with an empty model. These times have been measured by subsequently performing the refactoring analysis. This is the normal use case that occurs in practice, i.e. partial model construction combined with incremental reuse of parts of the model after program changes.
- The memory consumed by the cold "empty" model, i.e., the memory required for when the model holds only the minimum amount of required information, which are mainly indexes and dependence graphs to support partial model construction.
- The memory consumed by the model during the analysis performed by the Rename Refactoring.

VIII. Related Work

Our architecture is based on two ideas: the former is the (vertical) separation of model, meta-model and meta-meta-model. The latter is the (horizontal) separation of language-specific, common and analysis-specific views. To give this related work section structure, we discuss different domains that require meta-modeling and their contributions.

A. Language Specific Meta-Models

A meta-model for static program analysis is a high-level abstraction of a program where the information contained is specialized for a given set of analyses and refactorings. All static analysis tools contain such a model. Most tools are specialized to handle a specific language and use meta-models especially designed for this language. These meta-models are not language-transparent and can (in general) not easily be extended to handle other languages. Examples of single-language meta-models can be found in several tools like Datrix [16], RECODER [17], and IDEs like CODEGUIDE[4] or the ECLIPSE JDT[6].

B. Common Intermediate Representations

Common intermediate representations (IR) of programs are also a type of meta-models that are used in compilers and

[4] `www.omnicore.com`

[5] In JEDIT we renamed the method org.gjt.sp.jedit.EditPane.getView() and its 4 calls to a more meaningful name - there are overall 9 different `getView()` methods with different meanings. In JHOTDRAW we renamed the method `org.jhotdraw.gui.JSheet.init()` and its 2 calls to a more meaningful name – there are overall 18 different `init()` methods with different meanings.

[6] `www.eclipse.org`

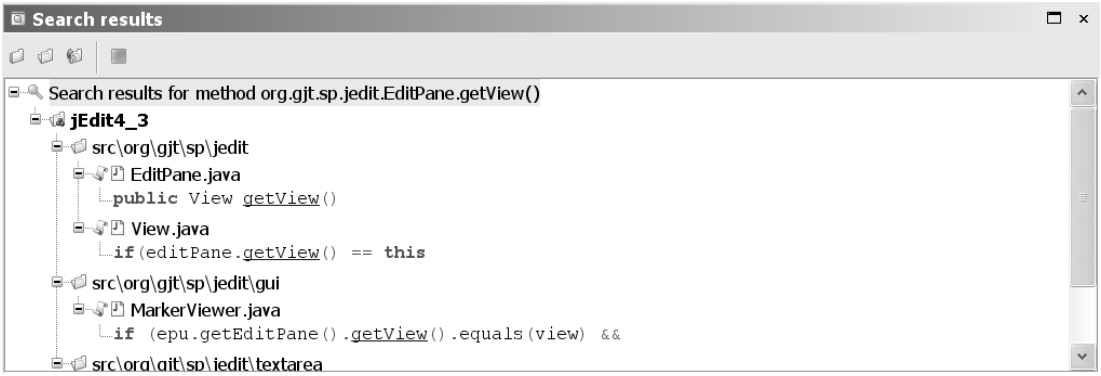

Fig. 4. Example: Precise search for usages of methods.

	Lines of code	Initial model build time	Partial model build time		Model size	
		cold	cold	warm	cold	warm
JEDIT	145508	32s	5s	2s	30mb	50mb
JHOTDRAW	57020	18s	4s	2s	20mb	40mb

TABLE VIII
X-DEVELOP'S CPU AND MEMORY REQUIREMENTS OF THE MODEL

virtual machines. These IRs preserve the execution semantics of a system and serve as a base for program analysis and optimization. Most are tight to a specific compiler and hence to a specific language. An example of an IR that can handle multiple languages include the .Net Common Language Runtime [18].

For several reasons, IRs are insufficient as a basis for source code transformations: one key issue is the lack of information and the missing link to the source code. Another problem with IRs is the specialization to compilable programming languages. The representations are not general enough to support other types of specifications that can usually be found as sources in software systems, e.g., UML specifications, scripting-, and markup languages.

C. Transformation Systems

Several transformation systems are also able to process multiple languages. These systems are designed for converting a model conforming to one meta-model into a model conforming to another meta-model. Examples of such systems are TXL [19], [20], QVT (Queries/Views/Transformations) [21],ASF+SDF [22], and DMS [23].

However, these tools construct language specific models to capture program information. This language-specific design of the meta-model makes analysis and refactoring inherently language-specific. Adapting them to support other languages requires changes to all parts of the system: the basic analyses for parsing source code, the design of the language-specific meta-model as well as higher-level analyses using this information. In practice, such an adaption is very expensive. Our solution to this problem is the use of a common meta-model decoupling language-specific and language independent parts.

D. Exchange Formats

There have also been efforts to define a standard exchange format for tools to exchange information, e.g., the Graph Exchange Language (GXL) [24], [25] and Rigi [26]. An overview about various exchange patterns implemented by different tools can be found in [27]. The observations in this paper strengthen the case for GXL as an exchange format. It defines an XML-based, language independent standard format for the information exchange between maintenance tools. GXL distinguish model level (graphs) from meta- and meta-meta-level (schema and meta-schema, resp.) guaranteeing extensibility and different levels of abstraction.

However, GXL is only an exchange format for tools, and thus in concrete implementations inherits all their specific limitations. In fact, we could implement our ideas on top of GXL as an implementation basis as well, which is future work.

E. UML-Related Approaches

Common models of software systems are also used in software architecture, design methods and tools. The Unified Modeling Language (UML) [28], [29] is defined to specify, visualize, and document software system design. UML as a meta-model is language independent, and, to a certain degree, extensible by means of new stereotypes. However, it describes software systems on an architectural or design level, which is not sufficiently detailed for refactorings of source code.

Meta-Object Facility (MOF) [30] is an extensible meta-meta-model for defining, manipulating, and integrating meta-models like UML in a language-transparent manner. XML Metadata Interchange (XMI) [31] provides rules by which a meta-model (XML schema) can be generated for MOF-based meta-meta-models. Like UML, both technologies are insufficient for general maintenance tools since they cannot capture detailed information on implementation level. When

used, e.g., in the Model-Driven Architecture (MDA) [32] and Engineering (MDE) [33] approaches, source code is explicitly added by the generators transforming a model into compilable code.

F. Metric-Related Meta-Models

Software metrics are used as indicators to identify problematic parts of a software system that might need maintenance or refactorings. A large number of meta-models have been used in this context with the purpose of presenting precise and language independent metric definitions. These papers present a meta-model by identifying a set of relevant source code entities (e.g. classes, methods, and fields) and a set of relations among these entities (e.g. inheritance, calls) that can be found in all programs of a given type (e.g. statically typed object-oriented programs). Then they provide a clear and precise metrics definition using these entities and relations. The basic formalism used changes from one paper to another.

Certain papers present their meta-models as a relational database schema and their metrics as SQL queries [34], [35]. Other models are based on the UML meta-model ([36], [37]) and use OCL to define their metrics [38], [39], [40], [41], [42]. Other meta-models relevant in the software metric community include the object-oriented FAMIX meta-model developed in the European Esprit Project FAMOOS [43], and the Dagstuhl Middle Metamodel (DMM) [9].

The meta-models used in these papers are language transparent (within their target programming paradigm) but are not designed with extensibility in mind. They only include entities and relations that are required to give a precise definition of a given set of software metrics.

G. Summary

In summary, the related work discussed above is not sufficient for supporting language independent program analysis. In contrast to our approach, they don't abstract reusable program information in a language independent manner, e.g. the language specific meta-model and transformation systems, or they miss a formal link between program information for specific language and its language independent abstraction - either the link from the common to the program specific model as, e.g., common intermediate representations, exchange formats - or vice versa as, e.g., exchange formats again, UML- and metric related approaches.

IX. Conclusions

We presented a meta-model for capturing program information as used in maintenance tools for analysis and refactoring and an architecture for creating instances thereof. The meta-model (and the architecture around) is *language-transparent* since it abstracts from language specific details of programming language concepts that are not necessary for analysis and refactoring. It is *extensible* due to the decoupling of front-ends extracting the information and analyses and refactorings accessing and modifying it. Moreover, meta-model extensions can simply be specified; the corresponding constructor and access operations are then generated automatically. This allows *efficient* meta-model extensions. Finally, it is *scalable* since it filters unnecessary details and captures only relevant information. Additionally, the architecture proposed allows for partial and incremental updates of the models avoiding a compleate recomputation whenever the source code changes. In practice, we tested and fine-tuned our meta-model architecture in two maintenance tools: VizzAnalyzer and X-develop.

Although both tools are widely used in industry projects, their real power should be demonstrated with running time, memory consumption data on benchmark programs. Moreover, although both tools went through quite a number of versions with extensions of both frontend and analysis/refactoring, the real flexibility and maintainability of our approach should be assessed in experiments, too. The latter appears quite difficult since the number of core developers of VizzAnalyzer and X-develop is quite small and controlled experiments promising statistically relevant results are hard to design and execute.

Another issue is the extension of the meta-model architecture towards dynamic analysis, e.g., debuggers and profilers, usually supporting static analysis in maintenance tasks. Finally, a (de-)serialization of our meta-model (from) to GXL would open up for the integration of many other maintenance tools and is therefore interesting from a practical perspective.

References

[1] B. W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[2] R. K. Fjeldstad and W. T. Hamlen, "Application program maintenance study: Report to our respondents," in *Proceedings GUIDE 48*, 1979.

[3] T. A. Standish, "An essay on software reuse," *IEEE Transactions on Software Engineering*, vol. 10, no. 5, pp. 494–497, September 1984.

[4] P. Selfridge, "Integrating code knowledge with a software information system," in *Proceesings of the 1190 Knowledge-Based Software Engineering Conference*, 1990.

[5] S. Henninger, "Case-based knowledge management tools for software development," *Journal of Automated Software Engineering*, vol. 4, no. 3, July 1997.

[6] Paul Klint, Ralf Lämmel, and Chris Verhoef, "Toward an engineering discipline for grammarware.," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 3, pp. 331–380, 2005.

[7] J. M. Favre, "Understanding-in-the-large," in *WPC '97: Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, Washington, DC, USA, 1997, p. 29, IEEE Computer Society.

[8] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow, "Txl: a rapid prototyping system for programming language dialects.," in *ICCL*, 1988, pp. 280–285.

[9] Adrian D. Thurston and James R. Cordy, "Evolving txl.," in *SCAM*, 2006, pp. 117–126.

[10] W. Löwe and Th. Panas, "Rapid construction of software comprehension tools," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 6, pp. 905–1023, December 2005.

[11] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder, "The dagstuhl middle metamodel," in *Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003)*, may 2004, vol. 94, pp. 7–18.

[12] A. Ludwig, "Recoder," http://recoder.sourceforge.net, 2002.

[13] R. Lincke and W. Löwe, "Foundations for defining software metrics," in *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006), Genoa, Italy*, Oct. 2006.

[14] D. Strein, H. Kratz, and W. Löwe, "Cross-language program analysis and refactoring," in *SCAM 2006 - Sixth IEEE International Workshop on Source Code Analysis and Manipulation – co-located with ICSM'06*, 2006.

[15] Arthur H. Watson and Thomas J. McCabe, "Structured testing: A testing methodology using the cyclomatic complexity metric," *NIST Special Publication 500-235*, 1996.

[16] Frank Tip and Jens Palsberg, "Scalable propagation-based call graph construction algorithms," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 281–293, 2000.
[17] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers, "Call graph construction in object-oriented languages," in *Procedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, 1997, pp. 108–124.
[18] Sébastien Lapierre, Bruno Laguë, and Charles Leduc, "Datrix(tm) source code model and its interchange format: lessons learned and considerations for future work," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 1, pp. 53–56, 2001.
[19] A. Ludwig and D. Heuzeroth, "Metaprogramming in the large," in *GCSE'2000*. 2000, number 2177 in LNCS, Springer.
[20] James S. Miller and Susann Ragsdale, *The Common Language Infrastructure Annotated Standard*, Addison Wesley, 2004.
[21] "MOF QVT final adopted specification," http://www.omg.org/docs/ptc/05-11-01.pdf, 2007.
[22] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier, "Compiling language definitions: the ASF+SDF compiler," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 4, pp. 334–368, 2002.
[23] I. Baxter, P. Pidgeon, and M. Mehlich, "Dms: Program transformations for practical scalable software evolution," in *International Conference on Software Engineering*, 2004.
[24] R. Holt, A. Winter, A. Schürr, and S. Sim, "GXL: Towards a standard exchange format," in *WCRE 2000 - 7th Working Conference on Reverse Engineering*, 2000.
[25] R. Holt and A. Winter, "GXL: Representing graph schemas," in *WCRE 2000 - 7th Working Conference on Reverse Engineering*, 2000.
[26] H. A. Müller and K. Klashinsky, "Rigi-a system for programming-in-the-large," in *ICSE '88: Proceedings of the 10th international conference on Software engineering*, Los Alamitos, CA, USA, 1988, pp. 80–86, IEEE Computer Society Press.
[27] Dean Jin, James R. Cordy, and Thomas R. Dean, "Where's the schema? A taxonomy of patterns for software exchange.," in *IWPC*, 2002, pp. 65–74.
[28] Grady Booch, James Rumbaugh, and Ivar Jacobson, *Unified Modeling Language User Guide*, Addison-Wesley Longman, ISBN 0-201-57168-4, 1998.
[29] "Unified Modeling Language (UML), version 2.0," ⟨`URL:http://www.omg.org/technology/documents/formal/uml.htm`⟩, 2006.
[30] "Meta-Object Facility (MOF), version 2.0," ⟨`URL:http://www.omg.org/technology/documents/formal/MOF_Core.htm`⟩, 2006.
[31] "XML Metadata Interchange (XMI), version 2.1," ⟨`URL:http://www.omg.org/technology/documents/formal/xmi.htm`⟩, 2005.
[32] David S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, OMG Press, 2003.
[33] Douglas C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.
[34] F.G. Wilkie and T.J. Harmer, "Tool support for measuring complexity in heterogeneous object-oriented software," in *18th IEEE International Conference on Software Maintenance (ICSM'02)*, 2002, p. 152.
[35] M. El-Wakil, A. El-Bastawisi, M. Riad, and A. Fahmy, "A novel approach to formalize object-oriented design metrics," in *Proceedings of Evaluation and Assessment in Software Engineering (EASE)*, Keele, UK, 2005.
[36] OMG, "Object management group. OMG unified modeling language specification version 1.3," 1999.
[37] OMG, "Object management group. OMG unified modeling language specification version 2.0," 2006.
[38] R. Reißing, "Towards a model for object-oriented design measurement," in *Proceedings 5th Int. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2001, pp. 71–84.
[39] A.L. Baroni, "Formal definition of object-oriented design metrics," M.S. thesis, Vrije Universiteit Brussel, Faculty of Sciences (Belgium) together with Ecole des Mines de Nantes (France) and Universidade Nova de Lisboa (Portugal), 2002.
[40] A.L. Baroni, C. Calero, M. Piattini, and F. Brito e Abreu, "A formal definition for object-relational database metrics," in *Proceedings of 7th International Conference on Enterprise Information Systems (ICEIS 2005)*, Miami, USA, May 2005.
[41] M. Goulão and F. Brito e Abre, "Formalizing metrics for COTS," in *Proceedings of the International Workshop on Models and Processes for the Evaluation of COTS Components (MPEC'04) at the ICSE'2004*, Edinburgh, Scotland, May 2004.
[42] J.A. McQuillan and J.F. Power, "Towards re-usable metric definitions at the meta-level," in *PhD Workshop of the 20th European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, July 2006.
[43] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod, "The famoos object-oriented reengineering handbook," http://www.iam.unibe.ch/˜famoos/handbook/, Oct. 1999.