# 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 universitet, 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 analysis 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 [2]. 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% [4], [17], [16], [7]. 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 principle tasks of these tools are the same: extracting information from a system and building a model thereof, 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 and building a model, analyzing, and, finally, modifying it.

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.

### A. Requirements on Meta-Models

An obvious requirement is *scalable performance*: especially large systems need maintenance tool support and these tools are often part of an edit-compile-cycle.

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 final major requirement is that the meta-model architecture should be efficiently *extensible* with new analysis-, refactoring-, and front-end-components.

### B. Contributions

1. 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*.
2. 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.
3. Finally, as a proof of concept, we implement the introduced design in two maintenance tools: VIZZANALYZER, a software analysis and visualization framework, and X-DEVELOP, a multi-language IDE.

Since, extensibility was a major requirement, the architecture proposed can be understood as an example of a maintainable system design – a side-effect contribution of this paper.

### C. Structure

Section II introduces the language-independent, extensible meta-model and the architecture for constructing and accessing instances. Section III discusses meta-model evolution and the effect of changes to existing components in the architecture. Sections IV and V introduce the proof-of-concept implementations, i.e., two maintenance tools, VIZZANALYZER and X-DEVELOP, respectively. Section VI relates our contributions to existing results. Finally, Section VII concludes the paper and shows directions of future work.

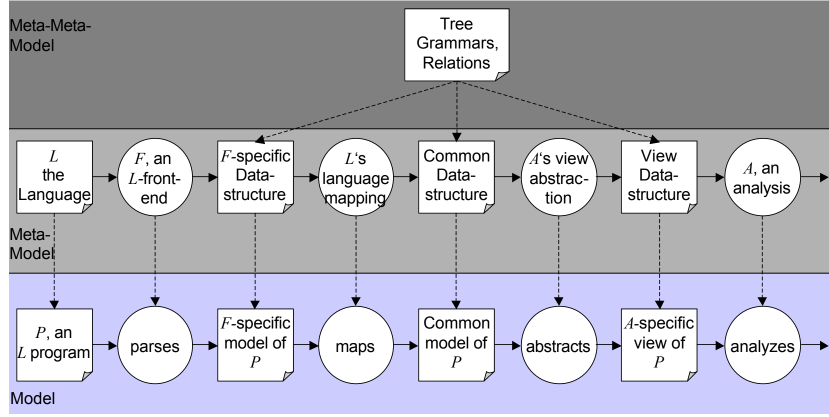


Fig. 1. Information extraction, mapping to common representation, view abstraction, and analysis – displayed on model- and meta-model-level.

## II. Constructing and Capturing Models

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 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*.

In what follows, we clearly distinguish the *model* from the *meta-model* level. The former captures more or less abstract information of a concrete program. In that sense, it contains *model*s of that program. The latter describes all possible *models* of programs. It can be understood as the type of models or a data structure capturing them. We separate front-end specific, common, and view models. When we define mappings between differently abstract models, we do that on the respective meta-model-level.

Finally, there is a common formalism that we will use for defining the front-end specific, the common, and the view meta-models. This common formalism, i.e., the meta-meta-model, will be *tree grammars* and *relational algebra*. See Figure 1 for an overview.

### 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.

We cannot influence the front-end $F$ specific representations, i.e., meta-models $\mathcal{M}^F$, but we may safely 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 BNF-productions defining structural containment tree structures, and $prog^F \in T^F$ the root type of the structural containment trees. BNF-productions $p \in P^F$ have the form:

$$t ::= expr$$

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

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

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

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

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

### 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 III. 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 frontend-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 $P$:

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

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

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* $\alpha^F$. 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 $\alpha^F$ is defined by mapping front-end specific to common model entities:

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

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:

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

In general, we don't require $\alpha^F$ to be:

- *surjective*, i.e., some common meta-model types and relations do not correspond to front-end specific meta-models 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 $\alpha^J$ maps node types and relations of $\mathcal{M}^J$ sketched in Example 1 to the common meta-model $\mathcal{M}$ sketched in Example 2. For the node types:

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

and for the relations:

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

For other types and relations of $\mathcal{M}^J$, 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-search order. We create new common model nodes of types with a mapping defined in $\alpha^F$ – we call those nodes *relevant*. The other, *irrelevant* nodes are ignored.

**Prog. 1** *generateTreeEvents*($n = \langle id, t^F \rangle$)

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

**Prog. 2** *startNode*($n = \langle id, t^F \rangle$)

```
if α^F(t^F) is defined then
    create new node n' :=< newId, α^F(t^F) >
    map(n) := n'   // define n's mapping
    relAncTmp := n   // set n to the current relevant ancestor
    childrenStack.top.append(n')   // top contains current childrenList
    childrenStack.push(create new childrenList)
end if
relAnc(n) := relAncTmp   // define n's relevant ancestor
```

**Prog. 3** *finishNode*($n = \langle id, t^F \rangle$)

```
if α^F(t) is defined then
    if children := childrenStack.pop() is not empty then
        add children to map(n)   // map(n) defined in Program 2
    end if
    relAncTmp := n   // reset the current relevant ancestor
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 Programs 1– 4: A *tree-walker*, cf. Program 1, initially called with the root node of the front-end specific model, traverses the containment tree in depth-first-search 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's key identifer and type, respectively.

The common model data structure is created by the corresponding event-listener, *startNode*, cf. Program 2, and *finishNode*, cf. Program 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 $\alpha^F$; 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 $\alpha^F$, 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 $\alpha^F$ was not defined for a type of a node $n_i$ in $R^F(n_1,\ldots,n_k)$, we would "attach" the relation to $n_i$'s closest relevant ancestor. That is the closest transitive parent of $n_i$, denoted by $relAnc(n_i)$, which is relevant. For relevant nodes $n$, by definition $relAnc(n) = n$.

**Prog. 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
```

**Prog. 5** *newRelationTuple*($R^F(< id_1, t_1^F >, \ldots, < id_k, t_k^F >)$)

```
if α^F(R^F) is defined then
    for each n_i =< id_i, t_i^F > 1 ≤ i ≤ k do
        r_i := map(relAnc(n_i))  // map and relAnc defined in Program 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. Program 4, and the corresponding event listener *newRelationTuple*, cf. Program 5.

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 Programs 1 and 4, is front-end specific and must obey the front-end specific meta-model APIs.

### C. View Meta-Models and Analysis

The common model is the data repository for program analysis. 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 $\alpha^A$. In contrast to the mapping from front-end specific to common models, $\alpha^A$ 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 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 [12] instead.

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

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

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 $\alpha^{CBC}$ 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.

### D. Refactoring

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

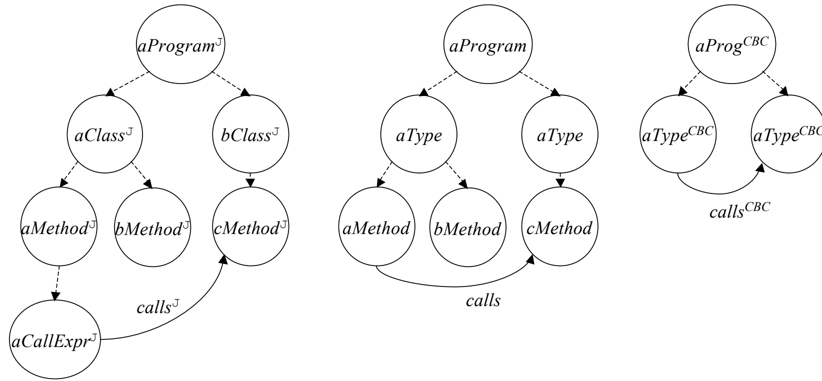


Fig. 2. A front-end specific, a common, and a view model corresponding to Examples 1- 4.

model this information is preserved. We use this position information to direct textual modifications of the source in refactorings. When a refactoring has been applied, we might completely rebuild the front-end specific and the common models again. In practice, our architecture implementation 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. Note, that we do not apply the refactorings on the model and then serialize this to source representations. This is not possible due to the abstractions in the models.

Refactoring requires to keep track of original source code entities since they change them. Consequently, if a source code entity is to modify in a refactoring $R$, we postulate some restrictions on the common and view meta-models $\mathcal{M}$ and $\mathcal{V}$, respectively, and the front-end and view mappings $\alpha^F$ and $\alpha^R$, respectively. Trivially, those source code entities must not be filtered in neither $\alpha^F$ nor $\alpha^R$. 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. Finally, we require *conservative* static analyses in the front-ends. We cannot guarantee the refactoring to be consistent, otherwise. If our call relation, e.g., just contained *some* call sites (that we renamed properly) while others remain unchanged, we would not be able to guarantee semantic correctness.

Other than that, refactoring and analysis can be treated identically. They are both defined on views of the common meta-model.

## III. 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 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 $\alpha^A$ 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 $\alpha^F$. 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 $\alpha^F$, 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.

*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:

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

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 $\alpha^{CBC}$ form Example 4 to this new meta-model, $\mathcal{V}^{CBC}$ remains unchanged, i.e., the view grammar is still

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

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 provided that no other type relevant for $A$ can transitively be derived from $t$.
- Introducing a new production $t ::= \ldots$ provided that 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. Inversely, if a meta-model change is not safe for an analysis $A$, we should check and potentially adapt $A$.

## IV. VizzAnalyzer

The VizzAnalyzer tool[1] is an instantiation of the VizzAnalyzer reverse engineering framework [12], 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.

### A. Adding a New Analysis

The initial meta-model was suitable for a number of object-oriented metrics. Adding McCabe's Cyclomatic Complexity (CC) metric [20] 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.

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:

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

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

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

A new view mapping $\alpha^{CC}$ trivially defines:

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

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.

### B. Adding a New Front-End

The initial, Recoder[2]-based front-end 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

[1] `www.arisa.se`

[2] `recoder.sourceforge.net`

some sort of data-flow analysis, which, in turn, requires a whole program representation [18], [5]. To add these kind of analyses, we integrate another front-end based on the byte code reader of the SOOT Framework[3]. It constructs a program representation 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 the SOOT front-end never generates.

## V. X-DEVELOP

X-DEVELOP[4] is a commercial Integrated Development Environment (IDE) supporting multiple programming languages. Its kernel implements a common meta-model as described in this paper. It is used to implement code-analysis-driven refactorings and visualization. Support for concrete languages is implemented in X-DEVELOP using language front-end plug-ins, currently for: `C#`, `Java`, `VisualBasic`, `J#`, `HTML`, `XML`, `ASP`, `JSP`, `JavaScript`.

As an example, we sketch how the *Rename Method* refactoring is implemented on X-DEVELOP's 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 could implement the refactoring in a language-independent way using the common model. As a result the refactoring 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. In fact, the *Rename Method* refactoring is only one example of a whole series of refactorings implemented in X-DEVELOP using the same approach.

## VI. RELATED WORK

All maintenance tools for software analysis, refactoring, and visualization need a meta-model capturing information on the processed programs. Most tools are language specific using a meta-model particular for one language. For example, RECODER [13] and IDEs like CODEGUIDE[4] or the ECLIPSE JDT[5] define a `Java`-specific meta-model.

A language-specific design of the meta-model also 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.

Common intermediate representations (IR) of programs are used in compilers and virtual machines. These IRs preserve the execution semantics of a system and serve as a base for program analysis and optimization. Examples range from the Java virtual machine [11] or the .Net common language runtime [14] to abstract state machine based IRs [10], [6]. 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. IRs preserve the execution semantic of programs but not the source structure and its relations. However, this information is required in source code processing tools. Another problem 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.

There are other approaches supporting the integration of arbitrary languages in reverse engineering tools. The DMS system [1], e.g., simplifies the implementation of parsers, generalized symbol tables and code analyzers for specific languages. Our approach goes beyond DMS in offering a common semantic model to represent relations that can be used for the language-independent implementation of higher-level analyses

[3] `www.sable.mcgill.ca/soot`

[4] `www.omnicore.com`

[5] `www.eclipse.org`

and refactorings.

Common models of software systems are also used in software architecture and design methods and tools. The Unified Modelling Language (UML) [3], [19] 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) [15] is an extensible meta-meta-model for defining, manipulating, and integrating meta-models like UML in a language-transparent manner. XML Metadata Interchange (XMI) [21] 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.

The work most closely related to ours is that on the Graph Exchange Language (GXL) [9], [8]. It defines an XML-based, language independent standard format for the information exchange between maintenance tools. GXL also distinguish model level (graphs) from meta- and meta-meta-level (schema and meta-schema, resp.) guaranteeing extensibility and different levels of abstraction. Our approach goes beyond GXL as we automatically generate mappings between those abstractions – in contrast to manually program them. We achieve this by assuming a designated containment structure, which other relations are attached to. In fact, we could implement these ideas on top of GXL as an implementation basis, which is future work.

## VII. 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 *scalable* since it filters unnecessary details and captures only relevant information. It is *language-transparent* since it abstracts from language specific details of programming language concepts that are not necessary for analysis and refactoring. Finally, 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 for *efficient* meta-model extensions. In practice, we tested and fine-tuned our meta-model architecture in two maintenance tools: VizzAnalyzer and X-develop.

Next on our agenda is an evolution of the meta-model architecture towards cross-language systems, i.e., systems with components defined in different programming and specification languages. This is interesting since many applications, e.g., Web applications, are cross-language applications. 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] I. Baxter, P. Pidgeon, and M. Mehlich. Dms: Program transformations for practical scalable software evolution. In *International Conference on Software Engineering*, 2004.

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

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley Longman, ISBN 0-201-57168-4, 1998.

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

[5] 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)*, pages 108–124, 1997.

[6] A. Heberle and W. Löwe. On ASM-based specification of programming language semantics and reusable correct compilations. In *5th International Workshop on Abstract State Machines*, 1998.

[7] S. Henninger. Case-based knowledge management tools for software development. *Journal of Automated Software Engineering*, 4(3), July 1997.

[8] R. Holt and A. Winter. GXL: Representing graph schemas. In *WCRE 2000 - 7th Working Conference on Reverse Engineering*, 2000.

[9] 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.

[10] P.W. Kutter and A. Pierantonio. Montages: Specifcations of realistic programming languages. *J.UCS*, 3(5):416 – 442, 1997.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition*. Addison Wesley, 1998.

[12] W. Löwe and Th. Panas. Rapid construction of software comprehension tools. *International Journal of Software Engineering and Knowledge Engineering*, 15(6):905–1023, December 2005.

[13] A. Ludwig and D. Heuzeroth. Metaprogramming in the large. In *GCSE'2000*, number 2177 in LNCS. Springer, 2000.

[14] James S. Miller and Susann Ragsdale. *The Common Language Infrastructure Annotated Standard*. Addison Wesley, 2004.

[15] Meta-Object Facility (MOF), version 2.0. ⟨URL:http://www.omg.org/technology/documents/formal/MOF_Core.htm⟩, 2006.

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

[17] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, September 1984.

[18] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, 2000.

[19] Unified Modeling Language (UML), version 2.0. ⟨URL:http://www.omg.org/technology/documents/formal/uml.htm⟩, 2006.

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

[21] XML Metadata Interchange (XMI), version 2.1. ⟨URL:http://www.omg.org/technology/documents/formal/xmi.htm⟩, 2005.