Cross-Language Program Analysis and Refactoring

Dennis Strein Hans Kratz
Omnicore Software
Werderstr. 87, 76137 Karlsruhe, Germany
{Strein|Kratz}@omnicore.com
www.omnicore.com

Welf Löwe
Växjö universitet, Software Technology Group, MSI
Vejdes Plats 7, SE-351 95 Växjö, Sweden
Welf.Lowe@vxu.se
www.vxu.se

Abstract

Many software systems are mixed-language systems today, i.e., they bind together components defined in different programming and specification languages. Analyses and refactorings implemented in current software development tools, e.g., integrated development environments (IDEs), cannot process these mixed-language systems as a whole since they are too closely related to particular programming languages and do not process mixed-language systems across language boundaries. In this paper, we discuss the foundations of cross-language analysis and refactoring. We propose a meta-model for capturing relevant information in a language independent way and an architecture for integrating language specific front-ends, and analysis- and refactoring-components. As a proof of concept, we introduce X-DEVELOP*, an IDE implementing the ideas discussed for a number of languages, analyses, and refactorings. Based on our contributions, cross-language software development tools are constructible in a straightforward way delivering the same productivity as today's single language tools.*

1 Introduction

Current integrated development environments (IDEs) like Eclipse and Visual Studio.Net support software development and maintenance by analysis and automated code transformation, i.e. refactoring [8, 3]. For instance, they analyze the correctness of syntax and static semantics while editing a program, or allow for consistent renaming of identifers. This support increases the productivity of software development and maintenance since it automates tedious and error prone tasks.

Many software systems are heterogenous today, i.e., they are composed of components of different programming and specification languages. Even a simple Java program could consist already of Java-source and -bytecode components. A larger system, e.g., a simple Web application, could merge SQL, HTML, and Java codes on the server site and additional languages on the client site. To support these mixed-language systems with automated analysis and refactoring, information from all different sources ought to be retrieved and commonly processed. Only a system global view allows for globally correct analysis and globally consist refactoring.

Today's IDEs fail in mixed-language systems, i.e., in cross-language analysis and refactorings. At best, they can only handle several programming languages individually. The reason is the lack of a common meta-model capturing program information for analysis and refactoring that is (i) common for a set of programming languages abstracting from details of each individual language, and that is (ii) related to the source code level of abstraction in order to allow for source-code analysis and refactoring.

We exemplify this with a small Web application. This Web application integrates `ASP`, `HTML`, `C#` and `VisualBasic` components:

```
// An ASP web page file
<%@ Page language="c#"% ClassName="WebForm">
<html>
  <head>
    <script language="c#" runat="server">
        public Button myButton;
        private void Init(string text) {
            myButton.Text = text;
        }
    </script>
  </head>
  <body>
    <asp:button id="myButton" runat="server">
    </asp:button>
  </body>
</html>

// A VisualBasic file
Public Class VBClient
    Sub SetText(f As WebForm, text As String)
        f.myButton.Text = text
    End Sub
End Class
```

This example contains an ASP Web page file and a VisualBasic file. The ASP Web page that is basically an HTML file with some special ASP elements and program code. When the page is requested on an ASP application server, the code is executed first, which results in a translated HTML code sent to the client. The page contains C# code in a script region. This code declares a field myButton and contains an initialization method Init that accesses the field to set its initial text. The page also contains a special HTML element <asp:button>, which represents a button. This element has an attribute id with the value myButton. The ASP application server uses this id to allow program code to refer to the <asp:button> element and to modify it before it is sent to clients. Thus, this occurrence of myButton is semantically related to the declaration of myButton in the C# code.

The VisualBasic file that accesses the field myButton of the class WebForm representing the ASP page at runtime. The occurrence of myButton in the VisualBasic code refers to the declaration of myButton in the C# code section of the ASP file. At runtime both the VisualBasic and the C# code are compiled to the same platform, i.e. to the .Net platform. This allows the direct interaction between both languages.

Now consider a renaming of myButton in the above example. This would imply that we needed to change code written in different languages, i.e. the C# declaration, the id attribute of the corresponding HTML element and the usage of myButton in the VisualBasic code. Accordingly, a tool automatically performing such a renaming refactoring needs to be able to automatically analyze all involved languages, as well as the involved cross-language relations. Hence, we have requirement (i).

Moreover, a close relation to the source code level of abstraction is needed in order to generate source code in the right language, at the right position, and with little changes to unrelated remaining code in order to avoid confusion of the programmer. The latter is hardly possible when capturing the analysis information on the abstraction level of an intermediate language for compilation, e.g. the .Net Common Intermediate Language. Therefore, we have requirement (ii).

In principle, all tools for software analysis, refactoring, and visualization need a meta-model capturing information gained by the analysis of the processed programs[1]. In [9], we described the design of such common meta-models to capture analysis information in a language independent way. Common models could be constructed automatically once we have specified mappings of the language specific meta-models of concrete front-ends to the common meta-model. We extend this approach with two new concepts: we support the construction of cross-language relations (mixed-language systems) and we support multiple languages in a single file (mixed-language files).

Section 2 describes how to design a common meta-model and an architecture around for extracting program information and utilizing them in analysis and refactoring. Section 3 defines a concrete practical common meta-model to capture information about languages found in typical real-world systems mixing object-oriented programming languages and markup languages. Section 4 describes the implementation of this meta-model in the X-DEVELOP IDE, and summarizes some experiences with a real-world applications. Section 5 discusses related work and Section 6 concludes the paper and shows directions of future work.

2 Design of a Common Meta-Model Architecture

Our architecture for analysis and refactoring in mixed-language software systems consists of three major classes of components: information extracting front-ends, a common meta-model (model data-structure), and analysis and refactoring components.

The *common meta-model* captures program information in a language independent representation.

Different language specific *front-end*s extract information from program (fragments) written in the respective languages. They use language specific analysis and capture information about the program in a language specific meta-model first. Information that is relevant for global analysis and refactoring is also stored in the common model. We are not restricted to *programming* languages here. The front-ends and models can capture other kinds information, as well, e.g., markup information.

The front-ends retrieve the information represented in the common model to implement low-level analyses (e.g., to look-up declarations). Retrieved information needs to be provided first by another front-end, potentially a front-end for a another language. Hence, this architecture allows the front-ends to access each others information and, hence, to construct cross-language relation between them.

Different high-level *analyses* and *refactorings* access the common model. The common model represents information gained from analysis of a complete mixed-language program. Thus, concrete analyses based on this information are language agnostic and can handle cross-language relations.

In short, a front-end is responsible for parsing and analyzing specific languages, whereas the common meta-model stores the relevant analysis information abstracting from language specific details. The common meta-model

[1] We need to distinguish *model* and *meta-model*: a model abstracts – more or less – from a concrete program, and, in that sense, it models that program. A meta-model describes all possible models of programs and can be understood as the type of models or a data structure capturing them.

is accessed by language independent analyses and refactorings.

A common meta-model will always only be able to support a limited set of languages and language constructs. Therefore, the it is not fixed, but may evolve as the need to support new language concepts arises. We have given more emphasis on such model evolution in [9].

Furthermore, the common meta-model does not need to be a union of all language concepts of all languages to support. Instead, it is sufficient to model only those language concepts, that are relevant to higher level analyses or to other languages. A union model would be unnecessarily complex and, most important, it requires much more changes to the model whenever support for a new language is desired.

What follows is a description of the general design of common meta-models. In the first place, this design is not limited to a specific class of language. Instead, following this general design, any particular meta-model can be derived for a particular class of languages. We outline such a particular meta-model then in Section 3.

2.1 Language-Specific and Common Models

The information about a mixed language system is captured in the different language-specific models. Parts that are relevant for cross-language analysis and refactorings are abstracted to a common model.

In general, a specification of a software systems is separated in different files f each having a specific file type F, denoted by $f \in F$. Each file can be parsed as a whole by a specific front-end; each front-end is dedicated to a specific file type F. Each file $f \in F$ may contain code several parts $f = \{p_1, \ldots, p_n\}$, and each part p_i may be of a different languages L_i. A front-end for F extracts the parts p_i and brings together all parts of the same language to common blocks $b_i \in L_i$. A block is the concatenation of same-language fragments in a file. As a result, we get as many blocks b_i as we have different languages in a file. Now we can say that a file type F is determined by the languages of the different blocks potentially contained in files of that type $F = \{L_1, \ldots, L_n\}$.

Example 1 `ASP` *files in Web applications are* `HTML` *files with embedded code scripts of several different programming languages. Thus,* $\texttt{ASP} = \{\texttt{C\#}, \texttt{HTML}, \texttt{JavaScript}, \texttt{ASP} - \texttt{directives}\}$. *Even simple* `C#` *files contain program code, pre-processor directives, and code comments. Thus,* $\texttt{C\#} = \{\texttt{C\#} - \texttt{code}, \texttt{C\#} - \texttt{comments}, \texttt{C\#} - \texttt{pre} - \texttt{processor} - \texttt{code}\}$.

The information for each block b_i can be parsed to an abstract syntax tree $AST_{b_i}^{L_i}$. For each file f, the result is a set of syntax trees representing the different blocks:

$$ASTS_f^F = \{AST_{b_1}^{L_1}, \ldots, AST_{b_n}^{L_n}\}, b_i \in L_i.$$

We denote the function ϕ^F that maps files $f \in F$ to its set of syntax trees $ASTS_f^F$ the *parsing function* of F:

$$\phi^F : F \rightarrow ASTS^F,$$

where $ASTS^F = \{AST^{L_1}, \ldots, AST^{L_n}\}$ is the set of syntax trees of block languages that are legal in files of type F and AST^L denotes the set of syntax trees of a particular language L.

Example 2 *A front-end for* `ASP` *files must be able to construct syntax trees representing the different file parts of the different languages that may be part of* `ASP` *files.* $ASTS^{\texttt{ASP}} = \{AST^{\texttt{C\#}}, AST^{\texttt{HTML}}, AST^{\texttt{JavaScript}}, AST^{\texttt{ASP-directives}}\}$. *The front-end have a high degree of freedom how to implement this parsing function. One possibility is to split the file initially in textual language-same fragments and then use traditional lexers and parsers.*

The abstract syntax trees AST^L of L can be described by a language specific tree grammar $\mathcal{G}^L$. Formally,

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

with T^L the set of AST node types, P^L a set of BNF-productions defining the abstract syntax tree structures, and $prog^L \in T^L$ the root type of the abstract syntax trees. BNF-productions $p \in P^L$ have the form $t ::= expr$, where $t \in T^L$, and $expr$ is an expression over $T \subseteq T^L$. Expression are either sequences $(t_1 \ldots t_k)$, iterations (t^*), or alternatives $(t_1 | \ldots | t_k)$ with the well-known semantics.

Both AST^L and $\mathcal{G}^L$ define the set of legal ASTs of (blocks of a) language L; it holds for the parsing function of F:

$$\phi^F : F \rightarrow \{AST^{L_1}, \ldots, AST^{L_n}\},$$

with the L_i being languages of F.

In order to make each b_i processable by further language L_i-specific analysis, the parsing of the language-same fragments may go along with some extra code generation. For instance, separating `HTML` from `C#` code in the example from the Introduction also requires to generate class declaration code around the declaration and initialization of `myButton`. This extra code ensures that the syntax trees conform to their respective tree grammars.

The abstract syntax trees are the basis for static semantic analysis. When looking only at a single language, the results are semantic relations over AST node types of T^L of a single language L. However, in mixed-language systems there are, in general, semantic relations over nodes types $T^{L_1}, \ldots, T^{L_n}$ of all involved languages $L_1, \ldots, L_n$ contained in different blocks of files of that system.

Example 3 *The tree grammar $\mathcal{G}^{\texttt{C\#}}$ of* `C#` *may contain method declaration $MethodD^{\texttt{C\#}}$ nodes and call expression $CallExp^{\texttt{C\#}}$ nodes. An important semantic relation is the call relation:*

$$call^{\texttt{C\#}} : CallExp^{\texttt{C\#}} \times MethodD^{\texttt{C\#}}.$$

The tree grammar $\mathcal{G}^{\texttt{VB}}$ of `VisualBasic` *may also contain method declaration $MethodD^{\texttt{VB}}$ nodes and call expression $CallExp^{\texttt{VB}}$ nodes. In a* `.Net` *environment, e.g., it is possible to call* `C#` *methods from* `VisualBasic` *sites. Thus, there is a semantic cross-language relation:*

$$call^{\texttt{VB,C\#}} : CallExp^{\texttt{VB}} \times MethodD^{\texttt{C\#}}.$$

The basic idea is to capture both intra-language and cross-language relations in a common meta-model. Therefore, such a common meta-model abstracts from the language specific abstract syntax trees and captures the semantic relations in a language independent manner. First, we describe how such a common meta-model is defined; then we discuss how common models (instances) are constructed.

Let

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

denote a common meta-model, where $\mathcal{G} = (T, P, prog)$ is a common AST grammar and $\mathcal{R} = \{R_1, \ldots, R_m\}$ is the set of common semantic relations over syntactic constructs $t \in T$. $\mathcal{G}$ defines an abstracted and unified view on the different abstract syntax trees of the specific languages involved; $\mathcal{R}$ defines a unified view on the different semantic relations of these languages. $\mathcal{G}$ contains only those syntactic constructs that are necessary for either constructing or representing the relations in $\mathcal{R}$. We denote those constructs as *relevant*. For instance, relevant syntactic constructs for representing $call$ relations are call expression nodes $CallExp$ and method declaration nodes $MethodD$. Independent of the language they are actually defined in, they will be represented in the common model. For constructing the $call$ relation, we also need scopes, e.g., name-spaces, packages, classes, inner classes, method declarations, blocks etc., in order to relate the call expression to the proper method declaration. These scope defining constructs are represented in the common model, as well.

For each language involved, we define relevant syntactic constructs by mapping specific syntactic meta-model entities to corresponding common meta-model entities. For each language L, the front-end defines such a *syntax mapping*

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

inducing a filtering function for computing common ASTs from relevant constructs of language L-specific ASTs, cf. also [9]. We assume α^L to be surjective for common node types $T_{relevant} \subseteq T$ and complete for L-specific node types $T^L_{relevant} \subseteq T^L$, which are relevant for semantic analysis. Also, we keep track of the induced mapping between the *relevant* L-specific nodes to the common AST nodes. This one-to-one mapping on node instances is denoted by $[\![\cdot]\!]^L$ for each language L.

Now we are ready to discuss how the common semantic relations are constructed. The basic ideas is that semantic analysis is based on the language specific and the common models. Analyzed semantic relations are only stored in the common model, but semantic analysis is triggered and performed (by front-ends) in a language-specific manner. From the front-end perspective, the common meta-model can be understood as a common compiler definition table that the language specific analyses read from and contribute to.

Initially, a common model just contains the relevant syntactic constructs from the different blocks of the files of the mixed-language system. Then, for each block, used occurrence of names are related to the corresponding defining occurrence, according to the semantic rules of that block's language. Since all relevant syntactic constructs therefore are already stored in the common model, this analysis can be performed on the common model. Regardless, whether the proper definition is in the same block, file, language, or not, it will be found in the common model and the semantic relation can be added to the common model as well.

Inversely, this means that, by applying the inverse function of $[\![\cdot]\!]^L$, we can also get the L-specific relations. The inverse is always defined since $[\![\cdot]\!]^L$ is a one-to-one mapping.

Note, that some semantic analyses are based on others. For instance, types of formal and actual parameters need to be resolved (type analysis) in order to resolve static call targets (name analysis), which, in turn, is required for type analysis on the caller's site. These complicated interleaving between type, inheritance, name, overloading, and operator analysis are neither simplified nor complicated due to the generalization to the cross-language case.

Since the used occurrences of names may be distributed in blocks of different languages, a common relation $R \in \mathcal{R}$ is constructed by different language specific *semantic analysis function* σ^L:

$$\sigma^L : \mathcal{G}^L \times \mathcal{M} \to \mathcal{M}.$$

As an input, σ^L takes a language L-specific $AST^L_b \in \mathcal{G}^L$ of a block b and an instance of the common meta-model $\mathcal{M} = (\mathcal{G}, \mathcal{R})$ and adds relation tuples to some relations $R \in \mathcal{R}$. Note, that σ^L using instances of $\mathcal{M}$ means that it can interact with arbitrary other languages' relevant constructs and can rely on other analysis functions' results generated into the common model.

Example 4 *For the* `Java` *programming language the calculation of σ^{Java} includes the resolution of method calls (as*

found in `Java` *compilers). The rules for these resolution are defined in detail in the* `Java` *language specification. These rules are encoded in* σ^{Java}*. On the other hand,* σ^{Java} *can use the common model* $\mathcal{M}$ *to look for method declarations by searching the common syntax tress for nodes of the common type* $MethodD$*. This way the* `Java` *front-end can construct method call relations to other programming languages.*

The common meta-model allows to store analysis results of software systems incorporating arbitrary language combinations an integrative way. Concrete high-level analysis and refactoring are based on the common model and can be implemented independently from the actually supported languages.

2.2 Front-ends

As mentioned before, the support for a specific file type F is implemented in *front-end* $\mathcal{F}^F$ of F. In the previous section, we have already seen the functions that need to be implemented by the front-ends in order to construct their respective contributions to the common meta-model. They are summarized here.

Each *front-end* supports a specific file type F that incorporates a set of supported specific languages: $F = \{L_1, \ldots, L_n\}$. A filetype F-specific front-end $\mathcal{F}^F$ is defined by a triple:

$$\mathcal{F}^F = (\phi^F, \{\alpha^{L_1}, \ldots, \alpha^{L_n}\}, \{\sigma^{L_1}, \ldots, \sigma^{L_n}\})$$

The front-end provides the *parsing function* ϕ^F that sorts file parts according to their languages into blocks and constructs syntax trees representing the different file blocks.

For each language L of such a block, the front-end defines the *syntax mapping* α^L, that maps language specific syntax trees AST^L to common meta-model trees AST.

For each language L the *semantic analysis function* σ^L constructs common semantic relations between nodes that are defined by the *syntax mapping*. σ^L is based on the common meta-model $\mathcal{M}$ as well as specific syntactic meta-model $\mathcal{G}^L$ to handle language specificities. Through the common meta-model $\mathcal{M}$ it can indirectly access information created by front-ends for other languages. This way, we can construct cross-language relations of arbitrary language combinations.

Note, that semantical relations are not limited to the typical relations for statically typed programming languages. They may include dynamic relations that would actually only be computed at runtime, e.g., dynamic types in weakly or dynamically typed languages or dynamic call targets in object-oriented languages. However, the computation of (non-trivial) dynamic program properties using static analysis is generally an undecidable problem. Thus, we use *conservative approximations* for constructing these dynamic relations.

Example 5 *The* `Java` *programming language, e.g., is an object-oriented, polymorphic language that supports virtual method calls. Thus, for a given call site the actually called method will be determined at runtime. Our conservative approximation of the* $call^{Java}$ *relation for* `Java` *and the corresponding common* $call$ *relation conservatively contains calls to all method declarations that are possible call targets.*

At least these three front-end components – parsing function, syntax mapping(s), semantic analysis function(s) – need to be implemented, whenever a new filetype with new languages is to integrate. Further adaptation might be necessary too, e.g., when the set of relevant common model constructs needs to be extended in order to capture properties of a new language. For a deeper analysis of the evolution of our architecture, we refer to [9].

2.3 High-Level Analysis and Refactorings

The common model is an integrative representation of a whole mixed-language program including cross-language relations. We use it as a source of information for high-level analysis (*high-level* as opposed to the syntactic and semantical analyses that we consider *low-level*) and refactoring, i.e. code transformation. To perform analysis and refactoring correctly, we rely on the correctness of information supplied by the common model, i.e., indirectly, by the specific front-ends. As already stated, we require conservative static low-level analyses in the front-ends. We cannot guarantee the refactoring to be consistent, otherwise.

Example 6 *A rename refactoring can automatically rename a method (declaration) and all call sites. While the textual modifications required for this refactorings are simple - they just have to update the names - finding the correct places to change is not. These places might be spread throughout the software system and different languages, which do not necessarily be programming languages, e.g. ASP directives allow specifying which method to call on certain events. The refactoring needs to know a conservative approximation of all possible call sites. Each of them could possibly target to other method declarations also. These other targets ought to be renamed, as well, to guarantee semantic correctness. Actually, we need to compute the transitive closure of the "possible method declarations" relation, since new call targets may introduce new call sites may introduce new call targets etc. For object-oriented languages, the common* $call$ *relation must therefore contain calls to all possibly called methods, cf. Example 5.*

With analysis information captured in the common meta-model at hand, we are now ready to perform the actual code transformations. Every node in the language specific syntax trees 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 relevant nodes are mapped to the common model this information is preserved, and this mapping is invertible. In refactorings, we use this position information for *direct* textual modifications of the source. The alternative, a modification of the model and a serialization to the respective file, is a non-option: parsing the different parts of a file and regrouping them in blocks are not invertible – even non-modified parts would be changed and moved leading to confusion for the user. A benefit of our approach is that transformations only change parts of the source code that actually needs to be modified.

After a refactoring has been applied, we might completely rebuild the front-end specific and the common models again. In practice, our implementation supports incremental model updates that reuse parts of the existing model to speed up the process of rebuilding the ASTs and the common model after refactoring. This is necessary, since IDEs integrate refactorings in the edit-compile cycle where program changes occur quite frequently.

3 A Practical Common Meta-Model Architecture

In the previous section, we described the general design of common meta-models to capture analysis information from mixed-language systems. This design is not limited to particular kinds of languages. In this section, we outline a concrete practical meta-model to capture information from languages found in typical Web applications: object-oriented programming and markup languages.

3.1 A Practical Common Meta-Model

Our common meta-model is defined by common tree grammar $\mathcal{G} = (T, P, prog)$ and common relations $\mathcal{R}$ to handle object-oriented programming languages like `Java`, `C#`, `VisualBasic` and `J#`, as well as markup languages like `XML` and `HTML`. We can not describe the complete meta-model. Instead we outline some of the most important node types and relations and how they are used.

Node types T include:

- $Prog$, represents a compilation unit
- $ClassD$, represents class declarations
- $InterfaceD$, represents method declarations
- $TypeD$, represents type declarations
- $MethodD$, represents method declarations
- $ParameterD$, represents method parameters
- $Identifier$, represents identifiers in the code
- $CallExp$, represents method call expressions
- Exp, represents general expressions
- $Stmt$, represents statements
- $Element$, represents markup elements
- $Attribute$, represents markup element attributes

Productions P defining the structural relations between nodes of these types include:

- $ClassD ::= Identifier(MethodD|ClassD|InterfaceD)^*$ represents the common syntax of classes.
- $MethodD ::= Identifier(ParameterD)^* Stmt$ represents the common syntax of methods. The $Identifier$ is the name of the method, the $ParameterD$ nodes are the parameters and the $Stmt$ node is the method body.
- $CallExp ::= ExpIdentifier(Exp)^*$ represents the common syntax of method calls. The first Exp is the expression the method is invoked on, the $Identifier$ is the name of the called method, the trailing Exp nodes are the arguments.
- $Element ::= (Attribute)^*(Element)^*$ represents elements and their attributes.

The semantics of a program is represented by a set of common relations between nodes of the common model: $\mathcal{R} = \{R_1, \ldots, R_m\}$. Similar to the common node types and productions, we define common relations required for object-oriented and markup-languages. The most important relations include:

$$\mathcal{R} = \{invokes, calls, inh, parametertype, methodtype, override\}$$

More specifically:

- $invokes : CallExpr \times MethodD \in \mathcal{R}$, represents method calls defined by the static program semantics.
- $calls : CallExpr \times MethodD \in \mathcal{R}$, represents all potential method calls.
- $inh : TypeD \times TypeD \in \mathcal{R}$, represents inheritance of types.

- $parametertype : ParameterD \times TypeD \in \mathcal{R}$, represents types of method parameters.
- $methodtype : MethodD \times TypeD \in \mathcal{R}$, represents types of methods.
- $override : MethodD \times MethodD \in \mathcal{R}$, represents a method that override another method.

The common model is constructed by concrete language front-ends, which in turn use the model as a source for analysis to access information from other languages. We discuss this in the next section.

3.2 Practical Front-Ends

As mentioned before, front-ends implement a *parsing function* ϕ^F, the *language mapping* functions α^L, and the language specific *sematic analysis* functions σ^L. Front-ends for different file types may share parts of the *parsing function*, *syntax mapping*, and *analysis function* in case they can contain the same languages.

Example 7 *The* `C#` *language appears in our* `ASP` *file type as well as pure* `C#` *files. Thus, we have two* front-ends *for the two file types* $F_1 = \texttt{ASP} = \{C\#, HTML, JavaScript\}$ *and* $F_2 = \texttt{C\#} = \{C\#\}$ *and* `C#` *is contained in both. Thus, both* front-ends *can share (parts of) the parsing function, the syntax mapping, and semantic analysis functions for* `C#`.

For our object-oriented programming languages, the sematic analysis functions include name and type resolution. Rules for these resolutions are highly language-specific and are defined in detail in respective language specifications. Despite the fact that they operate on a common model instead of language-specific models, their implementation does not differ from the one found in ordinary compiler front-ends for these languages.

As described, language interaction is achieved, because front-ends use the common model as a source for semantic analysis. For example, a front-end for a language L_1 searches for nodes of the common type $MethodD$ to provide the common method $calls : CallExp \times MethodD$ relation. To lookup the type of a method it looks in the common relation $methodtype$. This $methodtype$ relation, in turn, is constructed by the semantic analysis of a language L_2 that contained method declaration, and, hence, also constructed the $MethodD$ node. Actually, the languages don't matter; L_1 and L_2 may as well be the same language.

Semantic relations are not restricted to compiler-like relations. Generally, the semantic relations can express dynamic program properties and are constructed by conservative analysis. Our meta-model distinguishes, e.g., the $calls : CallExp \times MethodD$ and the $invokes : CallExp \times MethodD$ relations. The $calls$ relation is a conservative method call relation. We say $(c, m) \in calls$ if method call c is possibly a call to method m, and $(c, m) \notin calls$ if method call c is definitely not a call to method m. In statically typed object-oriented languages, it is always possible to find a static call target method m for any given call c following the rules of the language specification (unless the program is erroneous), i.e. the static $invokes$ relation. However, object-oriented languages support method overriding and virtual method invocation. Thus, the dynamic call target may also be one of the different overriding methods of the static target method.

For our object-oriented languages, we can construct the $call$ relation as follows: Provided the relation $override : MethodD \times MethodD$ expresses that a method overrides another:

$$calls(c, m) \quad \Leftrightarrow \quad \exists m' : invokes(c, m') \wedge overrides(m', m)$$

This is a very rough approximation that takes all overriding methods as possible candidates. Better results could be generated by a conservative flow analysis.

3.3 Practical Refactorings

We exemplify the use of the common model for high-level analysis and refactoring with the *Rename Method* refactoring. 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. The situation gets even more complex if multiple languages are involved. The method defined in language L_1 can be called from a call in language L_2 and we need to change code in both languages.

Our common model already provides syntactic and semantic information for implementing this refactoring. We make use of the common model node types $Identifier$ representing identifiers in the source code, $CallExpr$ representing callers and $MethodD$ representing callees, as well as the common $call$ relation defined above. Because $calls$ includes all possible method invocations, we can compute the set of calls and methods to be changed: starting from the call sites their declared targets, we compute the transitive closure of the union of the $calls$ relation and its inverse $calls^{-1} : MethodD \times CallExp$.

Then we look up the name of both calls and method declarations, i.e., we find the $Identifier$ nodes under

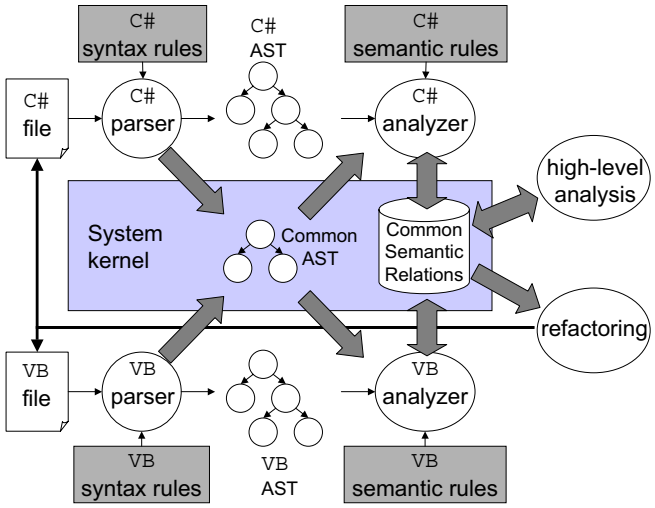

Figure 1. Kernel Architecture.

$CallExp$ and $MethodD$, respectively:

$$\begin{array}{rcl} CallExp & ::= & Exp\ \textbf{Identifier}\ (Exp)^* \\ MethodD & ::= & \textbf{Identifier}\ (ParameterD)^*\ Stmt \end{array}$$

For each such identifier node in the common model, we look up its textual position in the source and rename it textually and update our models accordingly.

Similar refactorings that are actually implemented in a similar way include:

- Change method signature, i.e., adding, removing, or reordering parameters. This refactoring is also based on the transitive closure of the $calls$ relation. Then it looks for actual and formal parameters under the $CallExp$ and $MethodD$ nodes involved.

- Move classes, i.e., changing name-space (`C#`) and package (`Java`) of classes. This refactoring requires name-space and package declarations, resp., using and import statements, resp., and fully-qualified names in the common model.

Following this approach, we can implement a refactoring in a language-independent way. Also, since our model reflects cross-language relations, our implementation can handle mixed-languages systems. The refactoring will work with any language that is supported by a front-end, and, 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.

4 The X-DEVELOP Implementation of a Practical Common Meta-Model Architecture

X-DEVELOP[2] is a product quality IDE supporting multiple programming languages and cross-language analysis and refactoring. Its kernel implements a common meta-model as described above. This common meta-model is then used in source-code-analysis-driven refactorings. Additionally, the kernel provides a common infrastructure, which is a set of data-structures and helper algorithms that can be used by front-ends to store their analysis results and to simplify their implementation. 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`. Figure 1 shows the architecture of X-develop's kernel and its interaction with language front-ends. In the figure, we only illustrated `C#` and `Visual Basic` as example.

4.1 Experiences with a Real-World System

In this section, we summarize some experiences processing a real-world software system with X-DEVELOP. We choose the open source project `ITextDotNet`, a PDF generation tool for the `.Net` platform, as an example. The source code is freely available. The overall size of the project is 251.225 lines of code. `ITextDotNet` is a good example of a mixed-language system. It makes use of three widespread `.Net` programming languages - `C#`, `J#` and

[2] `www.omnicore.com`

	Method Declarations		Call Expressions		Unrelated Textual	Total
	relevant	irrelevant	relevant	irrelevant	Occurrences (irrelevant)	
J#	7	80	420	795	479	1781
C#	0	0	374	249	136	759
VB	0	0	85	66	48	199
Whole system (sum)	7	80	879	1110	663	2739

Table 1. Relevant and irrelevant `add` Method Declarations and Call Sites in `ITextDotNet`.

Visual Basic. The project consists of core libraries written in J# and three clients using this core library. One client is written in J#, the second one in C# and the third one in Visual Basic. A complete semantic model of the source code also requires the analysis of the binary libraries (DLLs) referenced by the source code. Thus, there is a fourth language involved: .Net binary. X-DEVELOP has front-ends for each of those 4 languages.

One can realize the usefulness of our architecture already with our example code transformation: assume, we were to rename the method add(Element element) of the class com.lowagie.text.Document to some more meaningful name, say addTextElement. A textual search for add results in 2739 hits, 1781 of them in J# code, 199 in Visual Basic Code and 759 in C# code. Among those text places, there are the relevant method calls and declarations that have to be changed, as well as other unrelated method declarations and calls with the same name. Additionally, there are completely unrelated textual occurrences of add, for example, the declaration of a variable named add.

A semantic search for methods named add yields that there are actually 87 methods with the name add and the right signature defined in this project, of which 7 are directly or indirectly part of the inheritance hierarchy of com.lowagie.text.Document.add. Those methods have to be renamed together with all their call sites. In order to do that correctly, these occurrences have to be distinguished from syntactic occurrences of add. This requires semantic analysis.

Performing this code transformation manually would obviously be very time consuming and error prone. It would involve finding and modifying the correct of the 2739 possible places across different programming languages and sorting out the proper 7 method declarations and their 879 proper call sites. The Rename Refactoring implemented on top of our common model does this automatically. Given a method declaration, it asks for the new name, determines all methods in the hierarchy, finds the correct calls to those methods and changes them. In our case, the analysis finds all 879 relevant call sites, 420 of them in J# code, 85 in Visual Basic Code and 374 in C# code. Table 1 shows the full statistics.

The time required for the analysis is only about 5 seconds on a Mobile Pentium 4M machine with 2 GHz and 768 MB of RAM and another 5 seconds to update the common model after the modification.

5 Related Work

All tools for software analysis and refactoring need a meta-model capturing information on the processed programs. Although some of today's tools can handle several programming languages individually, they fail in mixed-language systems, i.e., in cross-language analysis and refactoring. They use language specific meta-models designed particularly for one language. For example, RECODER [6] and IDEs like CODEGUIDE[4] or the ECLIPSE JDT[3] define a Java-specific meta-model. These models can neither capture multi-language systems, i.e., capture source code written in different programming languages in a common model, nor mixed-language systems, i.e., capture relations between code fragments written in different programming languages.

Several transformation systems, e.g. ASF+SDF [12] and DMS [1], have been used for processing mixed language systems. Our approach goes beyond those tools by integrating semantic analysis across languages. Our common semantic model can represent whole mixed-language systems and can be used for the language-independent implementation of high level analyses and refactorings that rely on semantic information.

A re-engineering tool for multi-language software systems has been discussed in [4]. Besides many technical differences, their approach differs from our approach by using a union model of several programming languages. As stated above, the disadvantages of a union model is that it is unnecessary complex and that much more changes to the model are required whenever support for a new language is desired.

The MOOSE re-engineering environment [10] uses a common model to implement support for different languages. Refactorings can then implemented language independently using this common model. However, their model can not capture mixed-language systems and represent cross-language relations. Additionally, their model dif-

[3] www.eclipse.org

fers from ours by being mainly focused on traditional programming languages, i.e. Smalltalk and C++.

Common intermediate representations (IR) of programs are used in compilers and virtual machines as well. These IRs preserve the execution semantics of a system and serve as a base for program analysis and optimization. Examples are the `Java` virtual machine [5] or the `.Net` common language runtime [7]. 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.

Common models of software systems are also used in software architecture and design methods and tools. The Unified Modelling Language (UML) [2, 11] is defined to specify, visualize, and document software system design. UML as a meta-model is language independent, and, in principle, can also describe mixed-language systems. However, it describes software systems on an architectural or design level, which is not sufficiently detailed for implementing refactorings or similar tools.

6 Conclusions

In this paper, we introduced the design of a meta-model for cross-language analysis and refactoring and an architecture for constructing instances of that meta-model from mixed-language systems. Moreover, we proposed a practical meta-model for object-oriented and Web applications; it allows capturing systems with markup-, scripting-, and application-components using different markup scripting and object-oriented programming languages. Finally, we described our X-DEVELOP IDE implementing this practical meta-model and the architecture around including a number of front-ends, analyses, and refactorings.

The set of programming languages manageable depends on the capabilities of our common meta-model. In [9], we showed how such a common meta-model may evolve on demand of new programming languages. However, we currently lack clear constraints defining descriptively the class of languages captured today. These constraints are currently developed. They should in the future complement the description of the capabilities of our approach in this paper, which just lists the languages that X-DEVELOP actually supports.

Future work also includes the handling of those dynamically typed programming languages, e.g., interpreted languages like `SmallTalk`, where sufficiently precise models can only by constructed using data-flow analysis.

Another issue for future work is the awareness of generated code linking client and server components, e.g., in Web Services and EJB applications. For these extensions, we expect that our meta-model and architecture generalize. They require new front-ends, e.g., for WSDL- or deployment descriptor formats, and new conservative analyses resolving calls from clients to servers via the middleware.

Rather ambitious is extend the approach towards the `C/C++`-language. Here, the pre-processor with its text-rewriting semantics that is even cross-cutting the language constructs is expected to cause problems not solvable with our meta-model and architecture in a straight-forward way.

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] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley Longman, ISBN 0-201-57168-4, 1998.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program comprehension in multi-language systems. In *WCRE '98: Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, page 135, Washington, DC, USA, 1998. IEEE Computer Society.

[5] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification Second Edition*. Addison Wesley, 1998.

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

[7] J. S. Miller and S. Ragsdale. *The Common Language Infrastructure Annotated Standard*. Addison Wesley, 2004.

[8] W. F. Opdyke. Refactoring object-oriented frameworks, 1992.

[9] D. Strein, R. Lincke, J. Lundberg, and W. Löwe. An extensible meta-model for program analysis. In *ICSM 2006 - 22nd IEEE Int. Conference on Software Maintenance*, 2006.

[10] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, Dec. 2001.

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

[12] M. 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.*, 24(4):334–368, 2002.