2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation

Cross-Language Code Analysis and Refactoring

Philip Mayer, Andreas Schroeder
Chair for Programming & Software Engineering, Institute for Computer Science
Ludwig-Maximilians-Universität München, Germany
{mayer,schroeder}@pst.ifi.lmu.de

Abstract—**Software composed of artifacts written in multiple (programming) languages is pervasive in today's enterprise, desktop, and mobile applications. Since they form one system, artifacts from different languages reference one another, thus creating what we call semantic cross-language links. By their very nature, such links are out of scope of the individual programming language; they are ignored by most language-specific tools and are often only established — and checked for errors — at runtime. This is unfortunate since it requires additional testing, leads to brittle code, and lessens maintainability. In this paper, we advocate a generic approach to understanding, analyzing and refactoring cross-language code by explicitly specifying and exploiting semantic links with the aim of giving developers the same amount of control over and confidence in multi-language programs they have for single-language code today.**

I. Introduction

Most software systems, whether they are small apps for mobile devices or large-scale enterprise applications, are not written in any single language [1]. Instead, a multitude of languages is used in both development and operation of software — this includes standard imperative or object-oriented programming languages such as Java, C, C++, or Ruby, database languages such as SQL, UI description languages (such as HTML or different XML dialects), and a multitude of domain-specific languages for purposes such as system configuration. Additional languages are used for software-related tasks such as automating the build (Ant, Maven) or deployment (mostly, vendor-specific tools). Thus, multi-language programming is also multi-paradigm programming.

Using multiple programming languages in one project (polyglot programming [2]) follows the language-as-a-tool idea: Using each language, which includes DSLs, for the purpose they were built ideally increases the readability of the code and prevents artificial constructs in languages originally built for another purpose [3]. Another reason for multi-language software applications (MLSAs) [4] is integration of legacy systems, as there is typically much knowledge encoded in legacy source code. For these two reasons, we believe that diversity in programming languages will always exist.

Since the individual software artifacts written in different languages still form one software system, they need to interact and/or share information. That is, there is a mechanism encoded in some sort of framework which, based on a certain convention or configuration, binds artifacts together.

We use the term *semantic link* to refer to these underlying semantic connections between artifacts written in different languages since they *link* two or more artifacts together, regardless of how the link is actually defined. Correct semantic links are key to working MLSAs: Even if artifacts in two languages are correct in themselves, they may still not work in combination if the link is broken. Since semantic links are outside the scope of any single language, they are usually established at runtime and are furthermore not checked by the tools written for the language. Although the information on how to resolve such references exists, it is in many cases deeply embedded in framework or (virtual) machine code and only invoked at runtime, not at design-time.

Thus, developers must stay aware of semantic links at all times not to break the system when changing code; program understanding, too, requires another level of awareness. Using multiple languages thus requires additional testing, leads to brittle code, and lessens maintainability. Thinking further, developers may even refrain from using the right language for a particular issue due to these problems, and choose a less-suitable, but better supported language instead.

In this paper, we argue that semantic links are too important to MLSA development to be left implicit and buried in framework or VM code. We thus advocate a *generic approach to explicitly specifying and using semantic links*, which we call XLL (short for *Cross Language Links*). Once captured in an explicit way, we can use these links for supporting three separate use cases:

- *Program Understanding*: Resolved semantic links enable code navigation and viewing links between languages;
- *Code Analysis*: Failure to resolve a link suggests an error or at least a possible problem in the multi-language program to be investigated;
- *Refactoring*: Awareness of links can be used to propagate changes between languages, thus keeping links intact.

In our literature review, we have not found a generic approach to cross-language link specification and exploitation with encompasses all three of these use cases.

Our specific interest in this paper lies in MLSAs which employ languages of different paradigms, in particular a multi-purpose host language and different domain-specific languages such as relational languages and XML-based configurations or declarative UI descriptions. Semantic links in this context are often established by (sometimes transformed) names, and when wrong, fail catastrophically at runtime. We also believe that IDE integration in this context is important to make this approach directly usable for developers.

This paper is structured as follows: We begin by giving two motivational examples for our work in section II. We then

978-0-7695-4783-1/12 $26.00 © 2012 IEEE
DOI 10.1109/SCAM.2012.11

discuss the work on cross-language code analysis and refactoring available in the literature in section III, investigating areas which still need work. Our own approach to specifying and exploiting semantic links is laid out in section IV, and we briefly discuss an implementation in section V. We conclude in section VI.

II. Motivating Examples

We give two examples which we have used as motivation. Both are taken from real-life applications.

A. Ruby on Rails

Firstly, in the area of web-based enterprise applications, we find that *Ruby on Rails* applications [5] tightly integrate with relational databases. Using the "convention over configuration" approach, the Ruby on Rails framework assumes that for each entity class, a table in the linked relational database exists which has the pluralized name of the class name.

Single-value columns are bound to class properties without requiring a declaration. However, multi-valued relationships between classes and thus tables are specified by using three class-level method invocations, namely *belongs_to*, *has_one*, and *has_many* (a fourth, *has_and_belongs_to_many*, is ignored here for simplicity). Column names for relationships are assumed to follow a certain schema, namely, another table (or entity) is referenced by 'entityName_id' columns. Obviously, such columns must exist if declared in a class; the other way round, while not so important, is worth a warning as well.

Thus, we have two types of semantic links: The first one linking entity classes and tables, the second linking entity relationships to columns in linked tables. The second link is only relevant if the first is established. If either of these two links are missing, the system will fail on the first access to an entity class or class association.

The following code shows an example of the Ruby side:

```
class Thesis < ActiveRecord::Base {
   has_many :categories
   has_one :status
}
class Category < ActiveRecord::Base {
   belongs_to :thesis
}
class Status < ActiveRecord::Base {
  belongs_to :thesis
}
```

This code expects the following setup in the database, which must be set up by migrations before they can be used.

```
table theses ( id INT )
table categories ( id INT,
    thesis_id INT )
table statuses ( id INT, status_id INT )
```

B. Android

Secondly, in the mobile world, we find that *Android applications* [6] are built using Java code for writing the logic, while a custom XML dialect is used for describing the user interface. Here, each screen is described in its own file, which is accessed from Java by a constant generated from the actual location of the file. Second, each component which is to be referenced from Java (this mainly applies to buttons, text fields, and other clickable or editable elements) has a unique ID, which is declared in XML and referenced, again, in Java.

In contrast to the Ruby example, layouts and IDs are referenced explicitly in the Android framework by using specific methods; `setContentView` in the first and `findViewById` in the second case. Google provides a tool which generates numerical constants from layout files and XML IDs into a class called *R*; these constants must then be used as parameters in the above methods. However, this mechanism does not check whether an ID used in a Java class is actually declared in the layout file referenced in this class, which is another source of errors.

Thus, we have again two link types: First, a reference to a layout file, and second, a reference to a component ID within a specific layout file. If a referenced layout or ID does not in fact exist in the correct place, the system will fail at runtime.

The following code shows the Java part of these links:

```
WelcomeActivity extends Activity {

  public void onCreate(...) {

     setContentView(R.layout.welcome);

     ... = findViewById(R.id.button);
     ... = findViewById(R.id.textView);
  }
}
```

On the XML side, a layout file called `welcome.xml` is required in the directory `res/layouts`, which must contain the two ID declarations referenced above:

```
<LinearLayout xmlns:android...>

  <Button a:id="@+id/button" .../>
  <TextView a:id="@+id/textView" .../>

</LinearLayout>
```

III. Existing Work

Existing work in the area of multi-language software can be roughly categorized into two categories; the first is program understanding and analysis, the second is refactoring. Program understanding and analysis deals with easing comprehension of a system and gathering metrics, which can be done on source or binary representations, and generally involves creating a new representation of the information retrieved. Refactoring, on the other hand, deals directly with source code and must stay aware of the origin of artifacts at all times. Since we aim at supporting both use cases in our work, we investigate publications from both fields. A third section deals with more general work.

A. Analysis

We begin with the analysis field. To our knowledge, the first work which explicitly deals with multi-language software systems is the description of the PolyCARE tool by Linos in 1998 [7], a tool intended for facilitating program understanding and re-engineering. Although the tool itself handles each language separately and does not link artifacts of multiple languages (as also discussed in [8]), PolyCARE seems to be the first tool with an explicit focus on multi-language systems.

Linos et al. have since created two interesting additional tools. The first one (Multi-Language Tool, MT), described in 2003 in [8], is again intended for program understanding and re-engineering and is targeted at host-to-foreign language dependencies between C/C++ and Java code, i.e., at language-crossing function calls. An interactive, animation-based user interface allows exploration (viewing) of cross-language dependencies. The second tool, described in 2007 in [4], is geared towards metrics calculation. Instead of being based on source code, intermediate-language level code from the .NET platform is used. This approach eases the calculation of metrics across languages since the different syntax of each language does no longer need to be parsed.

Staying in the area of program understanding and analysis, another early work was published in 1998 by Kullbach et al. [9], in which a case study on a multi-language system integrating COBOL and JCL is presented. Here, source code is first translated into an object-based repository; the repository can then be easily queried for information about the combined program. Kullbach et al. identify so-called *conceptual relations* which are quite similar to our idea of semantic links. Analysis of the system is performed using the tool GUPRO MetaCARE, in which the GReQL query language can be used to query the repository.

A similar approach to Kullbach et al. is used by Deruelle et al. in their 2001 work [10]. Here, source and byte code is converted into an XML-based graph structure and queried. A new feature is the ability to track change propagation by graph rewriting. The resulting (transitive) changes are annotated to the graph which allows developers to see affected components; a precursor to automated refactorings as discussed in the next section.

Another method for analyzing dependencies between Java and C/C++ is discussed by Moise and Wong in 2005 [11]. Their approach is built on top of the Source Navigator [12] tool; source code is first extracted into facts which conform to the GXL (Graphical eXchange Language) format which can be used by several reverse-engineering tools. The algorithm for matching Java and C/C++ artifacts is coded by hand. Source Navigator uses so-called pluggable extractors for producing facts about each language, a prerequisite to a flexible architecture.

A very interesting approach to multi-language analysis has been published in 2010 by Cossette et al. [13]. Their tool, DSketch, is integrated into Eclipse [14] and uses pattern matching to find dependencies between artifacts, which are again similar to our semantic links. Developers write pattern specifications directly on the source code (which includes Java, XML, etc.) and a heuristic approach is used for finding and displaying matching elements right in the IDE. The use case here is manually finding related code through queries which greatly improves code understanding; since the tool uses approximations it is not directly usable for refactorings.

Finally, Pfeiffer and Wasowski discuss dependencies between artifacts in different languages and the components defining those artifacts in [15]. The motivation and idea behind their work is similar to ours in that they regard inter-language dependencies and cite a lack of knowledge about these as a major maintenance problem. Furthermore, they use a pattern language for specifying dependencies between definitions and references. The approach is performed offline (on EMF-XML files generated from actual source [16]) and focuses on visualization of established links, i.e. broken links are not considered. In their approach, links are defined through key/reference pairs which are directional. Therefore, annotation of different levels of severity is not possible.

A common theme to the previously mentioned program understanding and analysis approaches — with the exception of [13] — is the use of a (more or less abstract) model for representing facts about the system. Some tools are entirely independent of source code, such as the metrics calculation tool by Linos et al. [4], while others are closer; however, in general, fine-grained links back into the source code are not required since manipulation is not intended.

B. Refactoring

As far as we know, the first work to explicitly deal with refactoring of multi-language software is Strein et al., 2006 [17], [18]. A tree-grammar based common meta-model is used to capture information across languages and is filled by individual language adapters called *front-ends*, which are similar to the *extractors* of Moise and Wong [11]. However, Strein et al. add enough information to the common model to allow direct textual modifications in the source code.

Analyses and refactorings, finally, only access the information in the common model and can thus be implemented in a language-agnostic way. As an example, the rename refactoring for a method can be implemented by looking for method references, which are, on this level, the same elements across languages.

Strein et al. point out that the common meta-model is not a union model of all languages, but only contains the language concepts necessary for analysis and refactoring. Still, adding new languages with new concepts means that the meta-model needs to change; thus, meta-model evolution is explicitly discussed in [18].

The work of Strein et al. has been particularly inspiring for our own work although we use a different approach to knowledge representation and refactoring implementation: As we shall point out later, we do not employ a common meta-model and thus, there is no language-agnostic layer. Instead, our semantic links are established directly between artifacts of different languages. Instead of re-writing refactoring logic on a common layer, refactorings for individual languages are thus extended and combined.

In 2008, Chen and Johnson [19] rightly point out that more tool support for common use cases in cross-language linking is required to encourage developers to employ refactorings for increasing understandability and maintainability of their programs; but also that doing so is hard. As in our case, reliance on domain specific languages is a trigger for their work: They present a case study for extending the Java rename refactoring in Eclipse to consider references in XML documents used by the Struts, Hibernate, and Spring frameworks. The *refactoring participant* mechanism of the Java Development Tools (JDT, [20]) is used to implement the Java-based extensions.

Although their work does not present a generic approach, it similar in spirit since it explicitly addresses dealing with the semantic links between Java and XML fragments which are embedded within the frameworks mentioned above. Also mentioned is the difference between unidirectional and bidirectional implementations of refactorings, i.e. from where a refactoring can be triggered, which we address below as well.

In the same vein as Chen and Johnson, Kempf et al. [21] present an implementation of another JDT refactoring participant, this time extending the reach of the Java rename refactoring to the Groovy language.

Finally, in a work from 2011, Schink et al. [22] describe their work on implementing multi-language versions of the Rename Method and Push Down Method refactorings for Java, Hibernate, and SQL. Also addressed is the refactoring *Introduce Default Value* which is, refreshingly, a non-OO refactoring from the relational world. However, it is concluded that a general approach to multi-language refactorings — across all possible MLSAs — is not feasible while keeping to the established notions of semantic preservation.

A common theme of the refactoring approaches presented is that multi-language refactoring is desirable, but hard to achieve in general. The approaches presented are quite diverse: Strein et al. [17] use a full-blown cross-language meta-model and implement the complete refactoring in a language-agnostic way; by contrast, the implementations of Chen and Johnson [19] and Kempf et al. are implemented as JDT refactoring participants and work directly on the individual language level [21]. The work of Schink et al. additionally addresses relational databases [22].

As far as we know, the explicit specification of semantic links used in some of the analysis-related works has not yet been employed for refactorings. We will discuss a generic approach for exploiting such links in section IV.

C. Other Work

Another interesting work on multi-language programming deals with the business perspective. Published by Fjeldberg in 2008 [2], this work lists advantages and disadvantages of polyglot programming. To quote: *Perceived advantages of polyglot programming are productivity and maintainability, and the perceived disadvantages are knowledge, maintainability, and tool support.*

In the same vein, Kontogiannis et al. [23] list open issues for discussion in the area of comprehension and maintenance of multi-language applications (2006). Again, tool support is mentioned. On the data gathering side, questions of formalization and modeling of multi-language systems, extraction, discovery and storage of extracted information, and how to support exploration, queries, and knowledge management are identified. On the presentation side, there is the question of how to present information in a usable way. Finally, multi-language software, and the methods and tools created to support it is expected to have an impact on the software maintenance process which is not yet known.

IV. The XLL Approach

As indicated in the introduction and the previous chapter, our aim is to support developers in multi-language development in three use cases: Program understanding, code analysis, and refactoring. We support all three use cases by allowing developers to *explicitly declare the semantic links between language artifacts*.

Of course, what needs to be specified are actually *link types between artifact types*, for example the fact that each Ruby entity class must have a corresponding table with the pluralized class name; the actual resulting semantic links between concrete artifacts should be computed.

There are several requirements for this approach:

- *Artifact specification and access*. To be able to specify link types between artifact types, we must have a way of describing which artifact types are available, which attributes they have, and how they are related (for example, a Java class has methods; both class and method being artifact types). Furthermore, we must be able to access both the types and the actual instances in a concrete system. If we want to support developers in an IDE, additional functionality for navigation, error markings, and refactorings must be supported.
- *Link type specification*. Specification of links is core to our approach. Links must be expressive enough to support linking artifacts in complex environments such as the Java language where the interesting artifacts might be buried in several layers of classes or methods. Furthermore, there should be support for change propagation — to support refactoring of an artifact, we require a way of suggesting appropriate changes in related artifacts. On the other hand, the link language should be readable and usable by developers.
- *Resolving semantic links*. Link resolution, i.e. finding actual links for link types within a concrete software system, must be addressed and supported by the approach. The resolution must support both finding successful links and unsuccessful links, since both carry relevant information for developers.
- *Exploiting link information*. Program understanding is concerned with successful links — these enable code

navigation or visualizing dependencies. Code analysis is concerned with unsuccessful links — these indicate errors in the system. Finally, refactoring is concerned with keeping successful links intact, whether by preventing developers in certain actions or propagating change.

In the following subsections, we discuss the XLL approach by addressing these four requirements. For a graphical view of XLL, please refer to Fig. 3 at the end of this section.

A. Artifact Specification and Access

Since we are interested in linking artifact types together, we must first know about available artifact types and their relationships. In many languages, artifacts are linked together in a tree- or graph-like structure; for example, a common structure in OO languages is having classes which contain methods which in turn contain statements; we must be able to navigate such a structure to find the artifacts we want to link. Furthermore, we must be able to retrieve individual instances for actual link resolution, and have a link back into the source code for navigation, annotation, and refactoring.

In general, multi-language relationships can be handled on three levels: On the source code level, using language-specific meta-models, and using language-spanning meta-models.

Using source code requires parsing (mostly) text, for example using regular expressions. Supporting more complicated code relationships such as method calls are difficult to implement using this approach; furthermore, the link descriptions have to deal with all intricacies of a language which can get very technical.

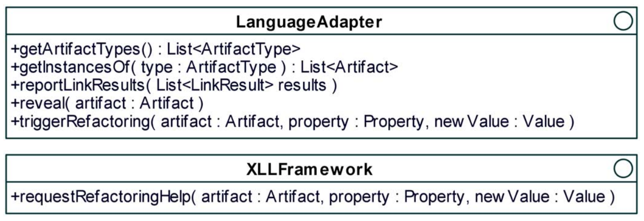
The second option is using a more abstract but still language-dependent representation of artifacts. This approach requires one adapter per language which manages the artifact types and instances. This approach allows re-using existing tools (such as the Eclipse JDT) and is able to hide complicated operations from link specifications in the adapter.

Finally, using a common meta-model allows writing analysis and refactoring rules in a language-agnostic way. However, this approach disregards the different concepts (such as classes, tables, etc.) available in each language. We believe that multi-language systems are created on purpose and that the difference between language artifacts is significant. Furthermore, a common model must be changed for each new language added.

We have thus adopted the second option, i.e. language-specific meta-models, for the XLL approach. This requires adapters similar to those of Moise and Wong [11]: the generic cross-language semantic link evaluation framework we propose must be able to access, based on user-provided link specifications, the relevant artifacts. The following information and functionality should be available:

- The set of artifact types available in the language, their attributes, and the associations between them. A convenient way for specifying this is using a meta-model, for example on the basis of EMF.
- Access to the actual instances of each artifact type, their attributes and associated artifacts. In other words, access to a concrete model instance for the meta-models defined above.

A language adapter must be implemented by hand for each language to be supported by the framework, though it can re-use existing meta-models (for example from the Atlant Ecore zoo [24]) and, for providing instances, existing AST parsers (for example the JDT, or XML DOM implementations). It is not necessary for the approach in general that meta-models are EMF-based; existing AST nodes or custom artifact types can be used as well. The functionality required from the language adapter is shown in Fig. 1; only the first three methods of `LanguageAdapter` are relevant here, the others and the `XLLFramework` interface will be discussed later.



```
LanguageAdapter
+getArtifactTypes() : List<ArtifactType>
+getInstancesOf( type : ArtifactType ) : List<Artifact>
+reportLinkResults( List<LinkResult> results )
+reveal( artifact : Artifact )
+triggerRefactoring( artifact : Artifact, property : Property, newValue : Value )

XLLFramework
+requestRefactoringHelp( artifact : Artifact, property : Property, newValue : Value )
```

Fig. 1. Language Adapter and Framework Interface

There is great freedom in the design of the individual language meta-models and adapters. On one end of the scale, we can provide a complete model of a language, enabling access to all elements but complicating the meta-models and models. On the other end, we can only provide the elements necessarily for link specification, which leads to smaller meta-models but less expressive power in the link specifications.

B. Link Type Specification

We now come to specifying the actual cross-language semantic links, i.e. the links between artifacts in different languages. To enable our use cases of program understanding, code analysis and refactoring, we require the following information to be present in these specifications:

- The artifact types which are to be linked and the path to the individual artifacts linked in each language. As mentioned above, an artifact may only be relevant if embedded in certain other artifacts; for example, the call to `has_a` in Ruby is only relevant for us if it occurs in a class extending `ActiveRecord`. The path to each artifact is not only relevant for catching appropriate artifacts, but also for checking edits: If the path changes, we might break a semantic link.
- Constraints on the artifact attributes necessary for establishing a correct link. For example, the name of a class in Ruby is the singularized version of the name of a table; other relation types may include substring extraction, concatenation, or simply equality. Besides ensuring semantic linking, this information can also be used for suggesting refactorings. It should be possible to specify relations in a bidirectional way.
- Hierarchical relationships between links. For example, looking for component references in an Android activity

```
transformation RubyOnRails ( ruby: Ruby, rdb: RDB ) {

    top relation ClassToTable {
        classname: String;
        error domain ruby c:Class { superClass = sc:Class { name='ActiveRecord' }, name= classname}
        error domain domain rdb t:Table { name=pluralize(classname) }
    }

    top relation BelongsToToTable {
        entityname: String;
        error domain ruby m:MethodInvocation { name='belongs_to', parent=c:Class {},
                parameter= p:Parameter { value= entityname }}
        error domain rdb col:Column { name= entityname + '_id', table= t:Table{}}
        when { ClassToTable(c, t) }
    }

    top relation HasOneManyToTable {
        entityname, classname: String;
        error domain ruby m:MethodInvocation { parent= c:Class { name=underline2camel(classname)},
                ((name = 'has_one', parameter= p:Parameter { value= singularize(entityname) }) |
                 (name = 'has_many', parameter= p:Parameter { value= entityname }) }
        error domain rdb col:Column { name= classname + '_id', table= t2:Table{ name= entityname }}
        when { ClassToTable(c, t) }
    }
}

transformation Android2XML ( djava: DJava, xml: XML ) {

    top relation ActivityToLayout {
        layoutName : String;
        error domain djava a:Activity { referencedLayout=layoutName }
        warn domain xml f:XMLFile { parent = d:Directory { name='layout', parent= dd:Directory { name= 'res' }},
                name = layoutName + '.xml' }
    }

    top relation IDReferenceDeclaration {
        reference: String;
        error domain djava lr:LayoutReference { activity= a, referencedID=reference }
        nocheck domain xml attr:Attribute { name='android:id', value= '@+id/' + reference },
                parent= e:Element { file= f }}
        when { ActivityToLayout(a, f) }
    }
}
```

Fig. 2. Example Cross Links: (1) Ruby on Rails, (2) Android

is only relevant if a layout has been identified before. Furthermore, link specifications should support recursive calls to enable more complex use cases.

Given these requirements, a relational, constraint-based specification seems to be well suited for describing, and later evaluating, link types. One architecture and language which fits this description and supports many of the above requirements is the MOF Query/View/Transformation (QVT) Relations specification, or QVT/R in short [25]. Although the aim of QVT/R is describing *model transformations* and thus has a different aim than what we intend to do here, we can re-use many of the ideas, syntax, and semantics. In the following, we will thus use QVT/R for the specification of link types and as the basis for link resolution.

The root element of a QVT/R specification is a *transformation* which consists of *relations*, each of which takes artifacts from at least two model *domains* and relates them based on their associations and attributes. QVT/R uses so-called *object* and *collection templates* to describe the elements to be bound to variables; *property template items* describe which element or elements are expected to be bound to a property of a matched object. A transformation is always executed in a certain direction, i.e. with a target domain. A domain can be annotated with either *checked* or *enforced*; depending on this flag, missing link partners are either only reported or automatically created.

QVT/R thus already provides us with the idea of specifying relations between model elements (in XLL: artifacts), and with the separation of domains (in XLL: languages); furthermore, constraints are used to bind artifacts together based on the values of associations and attributes. Other aspects of QVT/R match less well with our approach. We have thus adapted QVT/R and parts of OCL [26] with regard to both specification (syntax and intended semantics) and link resolution (next subsection). The changes to the specification are as follows:

- The QVT/R specification does not support disjunctions. This makes sense from a transformation perspective, as it is not clear with a disjunction which element to create without asking the user. In our case, we only deal with user-triggered refactorings on successful links, and can thus allow disjunctions in template specifications (using the pipe (|) sign).
- We are interested in both successful and unsuccessful links: Successfully matched artifacts enable program understanding and refactorings, unsuccessfully matched artifacts are problems. However, sometimes checking into one direction does not make sense, or the missing links are only to be regarded as warnings, not as errors. We thus allow annotation of domains with this effect (using **nocheck**, **warning**, and **error**).
- Finally, we are interested in bidirectional checking, i.e. checking each referenced domain as a target. This requires that all functions, even complicated ones, must be bidirectional as well. Consider our Ruby example: Here, tables carry the pluralized name of classes. We assume that such use cases occur often and thus allow the use of externally implemented bidirectional functions.

Fig. 2 shows the links present in our two case studies in the extended QVT/R textual syntax. The top specification shows the Ruby on Rails relations; the bottom deals with Android.

We first consider the Ruby on Rails transformation. The specification first lists which languages (and thus language adapters) are to be used; in this case the `Ruby` language (termed `ruby` in the example) and the `RDB` language (termed `rdb` in the example). The language adapter provides both the meta-model of this language and thus the artifact, attribute, and association names used in the relations (for example the `Class` artifact and the `parent` association) and, for later evaluation, concrete instances from source code.

The Ruby on Rails transformation consists of three relations. The first one deals with matching classes to tables: For each class based on `ActiveRecord`, a table must exist with the pluralized name of the class, and the other way around. The other two relations refer to the `ClassToTable` relation using a *when* clause; i.e. they are only evaluated if a link between class and table exists. An error is reported in both directions.

In the relation `BelongsToTable`, we check that for each `belongs_to` method invocation in a class, a column in the matching table with the tableized name of the referenced entity (and the string `_id`) exists. Again, this is an error in both directions, i.e. if a table has an `_id` column, there must be a `belongs_to` statement in the corresponding class.

Finally, we have the `HasOneManyToTable` relation in which we handle both `has_one` and `has_many` statements. In both cases, we require a column in the table of the *referenced* entity. In Ruby, `has_many` uses the original table name of the entity, while `has_one` uses the singularized version. This relation uses a disjunction: Moving from Ruby to RDB requires that we have an `_id` column in either case; moving from RDB to Ruby requires either a `has_many` or `has_a` statement.

Secondly, we consider the Android link. Here, we have the two domains (and thus languages and language adapters) DJava and XML. The names reflect the fact that XML is indeed a generic language adapter while we have created a specific adapter for the Java-based Android framework. We have two relations: Inside the first relation, we link activities with XML files: The layout referenced in the activity must exist as an XML file. Inside the second relation, we match layout references (extracted from `[this.]findViewByID` methods by the language adapter) with attribute specifications in XML starting with the string `@+id/`, which is the Android way of defining IDs.

In the first relation, the dJava pattern is annotated with error, meaning that if we do not find a matching XMLFile for an activity, this is indicated as an error. The other direction, i.e. an XMLFile exists but not an activity, is only a warning. In the second relation, a missing ID declaration in XML for a reference is again an error; if, however, a declaration exists which is not used we do not report anything.

The following section discusses how these link specifications are actually resolved.

C. Resolving Semantic Links

Link resolution is concerned with processing concrete element instances according to the link specifications from the previous subsection. In contrast to QVT/R, the model instances in XLL are actually language artifacts which stem from code written in the underlying programming language. They are created by the language adapters referenced in the link specifications, which are invoked by the XLL framework for a) checking that the referenced meta-model elements actually exist (thus validating the linking specification) and b) for retrieving instances of these elements.

For the use cases of program understanding and program analysis, the result of evaluating a link specification is thus a list of successful and unsuccessful semantic links. These links are extracted from the set of all artifacts provided by the language adapters as follows: In a first step, the patterns present in each relation domain are evaluated independently, generating a set of matching language-specific artifacts. However, the individual domain patterns not only contain constraints on the associations and attributes of source artifacts, but also constraints for variables used across multiple domain patterns (such as `layoutName` in the Android relation `ActivityToLayout`). These variables carry the actual cross-language link information.

Thus, the second step consists of solving the conjunction of the constraints from the individual domain patterns. This evaluation yields the set of artifacts for which the combined constraints hold true — these are the successful semantic links. Any artifact matching an individual domain pattern which is *not* present in the set of successful semantic links is in violation of the link, and is reported as a warning or an error (as specified in the relation).

A relation may also contain *where* and *when* clauses which affect the set of successful links and the set of non-matched artifacts. A *where* clause specifies additional constraints that semantic links must satisfy, reducing the set of successful links and enlarging the unsuccessful ones. Thus, it can be characterized as a postcondition. By contrast, the *when* clause is a precondition: It reduces the set of artifacts to be considered for linking before the patterns of the current relation are evaluated. Since the relation is only considered when the precondition is satisfied, the when clause restricts the set of unsuccessful links.

Although each pattern may refer to any number of variables, links are only established between the root variables of each domain pattern in a relation. Thus, evaluating a relation leads to a list of successful matches between root variables, and a list of root variables without counterpart(s).

We now discuss changes in the underlying source code. Firstly, any edit within a language on a published artifact (i.e., an artifact provided by the language adapter) may change links, leading to new links being created or existing links being broken. This applies to both direct edits and refactorings. In general, we react to such changes similar to other IDEs: by continually running analysis in the background and annotating code. For example, if a link breaks, we add an error annotation; if a link is newly established, we add a link annotation.

There is one use case where we can offer more than the "after-the-fact" annotations, namely certain refactorings on successfully linked artifacts. Since most cross-language links are established by names, rename refactorings on these names must be propagated to artifacts in other languages. If a rename refactoring is invoked in any one language on an attribute of an artifact which takes part in a link (not necessarily as a root variable), we can again use the link specification to compute the change propagation.

Change propagation is a subset of the QVT/R enforcement mechanism; however, in XLL, we start from successfully established links. Based on a change (i.e., an artifact, an attribute, and a new value) we first determine the set of affected links — if none is affected, the refactoring may proceed without cross-language impact. If there are affected links, we evaluate the source domain of each artifact link relation and compute the impact of the change on variables shared by the relation domains. If there is an impact, we propagate the change through the shared variables to the linked attributes of artifacts in the target domain. In contrast to computing artifact links, this process considers attribute values as modifiable if they are in relation to changed shared variables.

Changes in target artifact attributes must again be considered as input to the process since they may, in turn, affect other artifact links. Thus, change propagation must be repeated until the set of changes is stable.

If the changes lead to a consistent state, the process reports a set of new values for attributes of artifacts in the target domain. These can be used to trigger language-specific refactorings (more on this below in link exploitation). If a consistent state cannot be reached, the user is notified and may abort the current refactoring or continue with performing the required changes manually.

D. Exploiting Link Information

The successful and unsuccessful links created by the link resolution can be used in our three use cases as follows:

- *Program Understanding*. The key to understanding multi-language software is the information about where semantic links occur. Having successfully evaluated the links as discussed above, we can now use successful links to visualize the connections between languages in textual or graphical form, or help developers navigate the code. For example, we can offer navigation functionality between an Android activity and the corresponding layout file.
- *Program Analysis*. As we have seen before, non-resolvable semantic links mostly lead to errors at runtime — which is the worst possible time. A key aim of this approach thus was moving the detection of such errors to design-time. With the evaluation of the cross-links, we immediately get information about missing links including a severity. This information can be annotated in an IDE; it can also be used as part of a build process (Ant, Maven) to produce build errors or failures which can be listed in a continuous integration tool.
- *Refactoring*. Finally, a common method for improving code quality and readability is refactoring, i.e. changing source code while keeping its semantics. As mentioned in the last section, changes proposed by a refactoring of artifacts in a successful link can be evaluated in the link specification. Through change propagation, a property of a target artifact in another language might need to be changed, which means a second, derived refactoring in that language. Thus, that refactoring does not need to be explicitly defined but is a result of keeping links intact. For example, if a class is renamed in Ruby, we can issue a refactoring with the new pluralized name for the corresponding table to keep the link intact.

The three use cases discussed above work best when integrated into an IDE, and thus existing views and refactorings can be re-used. To enable such an integration, we require additional functionality from the XLL framework and from the language adapters (compare again Fig. 1):

- Successfully bound artifacts of one language should be reported to the language infrastructure, such that errors can be reported and successful links annotated with navigation information.
- To enable code navigation, a language adapter must provide the ability to open a certain artifact in an editor or view for the user in the underlying IDE.
- Refactorings triggered by the user on a specific artifact must be reported to the XLL framework along with the artifact, affected attribute, and new values for the attribute, such that linked artifacts can be identified and changes to them suggested as well. We call this a *request for refactoring help*.

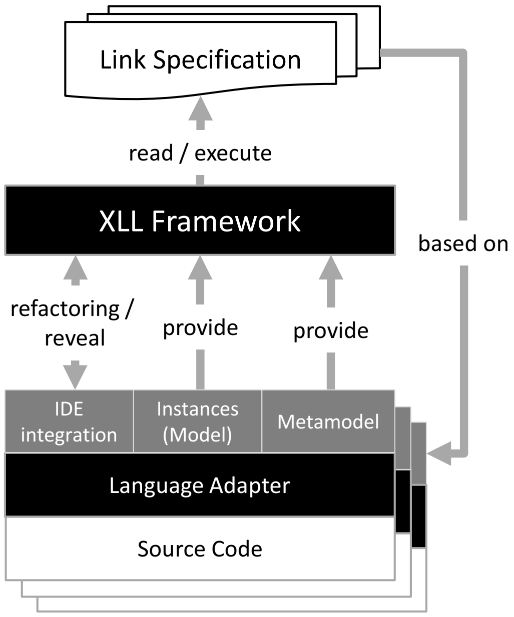


Fig. 3. The XLL Approach

- Finally, the XLL framework must be able to trigger refactorings in a certain language. For example, if a class is renamed in Ruby, link resolution will suggest a change in a table as well. This change should then be performed by a Rename Table refactoring in the RDB language infrastructure.

With this discussion of link exploitation, we have completed our description of the XLL approach to cross-language linking. An overview of the complete approach in graphical form is shown in Fig. 3. The figure shows the XLL framework (middle) which evaluates the user-defined link specification (top). On the bottom, the language adapters provide access to language-specific details: The link specification is based on the meta-model; the framework furthermore requires instances for evaluating the link specification and an IDE integration for code navigation and refactoring.

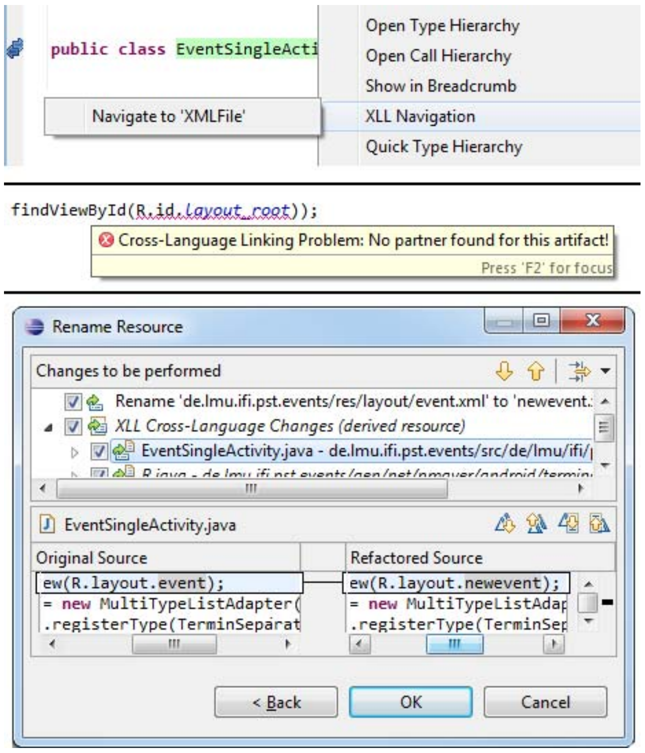
V. Implementation

We have implemented a version of our XLL approach into the integrated development environment Eclipse, and have checked the two real-life applications mentioned in section II to ensure that the implementation is working as expected.

To support the framework, we have implemented four language adapters as plug-ins for Eclipse: Two generic ones (Ruby and XML), one semi-generic one (Ruby-based RDB), and one specialized one ("Android Java"). All four provide meta-models and instances based on the underlying source as well as refactoring and navigation support.

Support for the three use cases is as follows:

- Program understanding is supported by annotating linked artifacts and offering code navigation via pop-up menu. As seen in Fig. 4 (1) for the Android example, the XLL navigation allows developers to open a layout XMLFile (the counterpart of an Activity) directly from the code.



```
public class EventSingleActi
```

```
findViewById(R.id.layout_root));
```

Cross-Language Linking Problem: No partner found for this artifact!

Rename 'de.lmu.ifi.pst.events/res/layout/event.xml' to 'newevent.

XLL Cross-Language Changes (derived resource)

EventSingleActivity.java - de.lmu.ifi.pst.events/src/de/lmu/ifi/

```
ew(R.layout.event);              ew(R.layout.newevent);
= new MultiTypeListAdapter(      = new MultiTypeListAdap
.registerType(TerminSeparat      .registerType(TerminSep
```

Fig. 4. Example: (1) Navigation (2) Error Marking (3) Refactoring

- Code analysis is supported via warning and error annotations in the source code; for example, an ID reference without declaration is flagged as an error and listed in the problems view. Fig. 4 (2) shows an ID reference in an Android activity which is not declared in the corresponding layout file.
- Finally, refactorings are supported using custom language-specific refactoring participants. These participant check whether the refactoring affects an artifact which takes part in a successful link. If so, the XLL framework is triggered to determine whether refactorings in other languages are required, which can then be invoked. Fig. 4 (3) shows a rename of the Android XML layout file `event.xml` with language-spanning changes in the Java source code (in particular, the reference to `event`).

While the first two use cases are relatively easy to implement, the third requires the availability of language-specific refactorings. Unfortunately, good refactoring support in Eclipse is only available for Java and, to a limited extent, Ruby [27]. Since the XLL refactoring approach is dependent on language-restricted refactorings, we have implemented simple, side-effect additional refactorings for our prototype; we hope to be able to use more external refactorings in the future. Another goal for the XLL framework is adding more generic language adapters and sample link specifications.

VI. Conclusion

In this paper, we have discussed the problem of cross-language links between artifacts in different languages which form one *Multi-Language Software System* (MLSA) [4]. These links are often neither explicitly specified nor checked by single-language tools. In fact, the specification of these links is mostly deeply embedded within framework code and only resolved at runtime, which means that errors are only found at runtime as well [21]. We believe that this situation should be changed, as it lessens maintainability, creates fear of changes, and in general leads to brittle code.

After a thorough review of related work, we have concluded that a generic approach to cross-language link specification with encompasses program understanding, analysis, and refactoring has not yet been attempted. We have then set out to create such an approach, termed *XLL (Cross-Language Links)*.

The XLL approach is based on *explicitly specifying semantic links* between artifacts written in different languages using QVT/R. The specification is evaluated based on the meta-model and model instances of individual languages which are provided on-the-fly by language adapters. Evaluation creates successful and unsuccessful cross-language links which are used for code navigation and error annotation within an IDE as suggested e.g. in [13], [17].

If a refactoring in any single language affects an artifact in a successful link, the semantic link specification can be used again to propose additional changes in the code to keep the link intact. If this is not possible, an error is reported.

Our approach leaves a lot of freedom regarding the design of the language-specific meta-model design to adapter providers; thus, adding new languages is straightforward. With regard to the link specification, the QVT/R language provides a user-friendly way of writing down constraints. QVT/R specifications can be used bidirectionally (cf. [19]), which further eases development. For refactorings, we re-use the algorithms and implementations present in existing languages.

We believe that the XLL approach can help developers gain more confidence in multi-language code since errors are reported during design time, artifact navigation is possible across languages, and refactorings show warnings or suggestions where cross-language links are involved.

Acknowledgements: This work has been partially sponsored by the EU project ASCENS, 257414.

References

[1] T. C. Jones, *Estimating Software Costs*, 2nd ed. New York, NY, USA: McGraw-Hill, Inc., 2007.

[2] H.-C. Fjeldberg, "Polyglot programming. a business perspective," Master Thesis, Norwegian University of Science and Technology, 2008.

[3] M. Fowler, *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[4] P. Linos, W. Lucas, S. Myers, and E. Maier, "A metrics tool for multi-language software," in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, ser. SEA '07. Anaheim, CA, USA: ACTA Press, 2007, pp. 324–329.

[5] D. Thomas, D. Hansson, L. Breedt, M. Clark, J. D. Davidson, J. Gehtland, and A. Schwarz, *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006.

[6] R. Meier, *Professional Android 2 Application Development*, 1st ed. Birmingham, UK, UK: Wrox Press Ltd., 2010.

[7] P. Linos, "Polycare: A tool for re-engineering multi-language program integrations," in *Engineering Complex Computer Systems*. Ft. Lauderdale, Florida, USA: IEEE Computer Society Press, November 1995, pp. 338–341.

[8] P. K. Linos, Z.-h. Chen, S. Berrier, and B. O'Rourke, "A tool for understanding multi-language program dependencies," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, ser. IWPC '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 64–73.

[9] B. Kullbach, A. Winter, P. Dahm, and J. Ebert, "Program comprehension in multi-language systems," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, ser. WCRE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 135–144.

[10] L. Deruelle, N. Melab, M. Bouneffa, and H. Basson, "Analysis and manipulation of distributed multi-language software code," in *SCAM*. IEEE Computer Society, 2001, pp. 45–56.

[11] D. L. Moise and K. Wong, "Extracting and representing cross-language dependencies in diverse software systems," in *Proceedings of the 12th Working Conference on Reverse Engineering*, ser. WCRE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 209–218.

[12] Mo DeJong and Erin Odenweiller and Syd Pol and Ian Roxborough, "The Source Navigator IDE," 2008, http://sourcenav.sourceforge.net/.

[13] B. Cossette and R. J. Walker, "Dsketch: lightweight, adaptable dependency analysis," in *SIGSOFT FSE*, G.-C. Roman and K. J. Sullivan, Eds. ACM, 2010, pp. 297–306.

[14] Eclipse Foundation, "The Eclipse Project," 2012, http://www.eclipse.org/.

[15] R.-H. Pfeiffer and A. Wasowski, "Taming the confusion of languages," in *Proceedings of the 7th European conference on Modelling foundations and applications*, ser. ECMFA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 312–328. [Online]. Available: http://dl.acm.org/citation.cfm?id=2023522.2023552

[16] Eclipse Foundation, "EMF: The Eclipse Modeling Framework," 2010, http://eclipse.org/modeling/emf/.

[17] D. Strein, H. Kratz, and W. Lowe, "Cross-language program analysis and refactoring," in *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, ser. SCAM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 207–216.

[18] D. Strein, R. Lincke, J. Lundberg, and W. Löwe, "An extensible meta-model for program analysis," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 592–607, Sep. 2007.

[19] N. Chen and R. Johnson, "Toward refactoring in a polyglot world: extending automated refactoring support across java and xml," in *Proceedings of the 2nd Workshop on Refactoring Tools*, ser. WRT '08. New York, NY, USA: ACM, 2008, pp. 4:1–4:4.

[20] Eclipse Foundation, "Eclipse Java Development Tools (JDT)," 2012, http://www.eclipse.org/jdt/.

[21] M. Kempf, R. Kleeb, M. Klenk, and P. Sommerlad, "Cross language refactoring for eclipse plug-ins," in *Proceedings of the 2nd Workshop on Refactoring Tools*, ser. WRT '08. New York, NY, USA: ACM, 2008, pp. 1:1–1:4.

[22] H. Schink, M. Kuhlemann, G. Saake, and R. Lämmel, "Hurdles in multi-language refactoring of hibernate applications," in *ICSOFT (2)*, M. J. E. Cuaresma, B. Shishkov, and J. Cordeiro, Eds. SciTePress, 2011, pp. 129–134.

[23] K. Kontogiannis, P. Linos, and K. Wong, "Comprehension and maintenance of large-scale multi-language software applications," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, ser. ICSM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 497–500.

[24] Atlant, "Ecore Zoo," 2012, http://www.emn.fr/z-info/atlanmod/index.php/Ecore.

[25] OMG (Object Management Group), "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification 1.1," OMG (Object Management Group), Specification, 1 2011, http://www.omg.org/spec/QVT/.

[26] ——, "Object Constraint Language 2.3.1," OMG (Object Management Group), Specification, 1 2012, http://www.omg.org/spec/OCL/.

[27] T. Corbat, L. Felber, and M. Stocker, "Refactoring support for the eclipse ruby development tools," Diploma Thesis, HSR University of Applied Sciences Rapperswil, 2006.