Automated Multi-Language Artifact Binding and Rename Refactoring between Java and DSLs used by Java Frameworks

Philip Mayer and Andreas Schroeder

Programming & Software Engineering Group
Ludwig-Maximilians-Universität München, Germany
{mayer,schroeder}@pst.ifi.lmu.de

Abstract. Developing non-trivial software applications involves using multiple programming languages. Although each language is used to describe a particular aspect of the system, artifacts defined inside those languages reference each other across language boundaries; such references are often only resolved at runtime. However, it is important for developers to be aware of these references during development time for programming understanding, bug prevention, and refactoring. In this work, we report on a) an approach and tool for automatically identifying multi-language relevant artifacts, finding references between artifacts in different languages, and (rename-) refactoring them, and b) on an experimental evaluation of the approach on seven open-source case studies which use a total of six languages found in three frameworks. As our main result, we provide insights into the incidence of multi-language bindings in the case studies as well as the feasibility of automated multi-language rename refactorings.

Keywords: multi language software, polyglot programming, Java, domain-specific languages, program comprehension, refactoring, experiment

1 Introduction

The use of multiple programming languages in the development of a software system is a common occurrence in software creation. In many cases, a multitude of languages is used; this includes the well-known general purpose languages Java, C, C++, JavaScript, or Ruby; but also domain-specific languages (DSLs) which are dedicated to certain areas, such as the database field (SQL, HQL, Entity Mapping Files), user interface design (HTML, JSP, JSF, OpenGL, SVG, CSS) or system setup and configuration (WSDL, Spring IOC, OSGi DS).

There are different reasons for using multiple languages in the development of a software system. A usually cited benefit is increased productivity [3] through the use of specialized languages for a certain domain (language-as-a-tool metaphor). Additional reasons lie in the use of legacy code (system integration) and in the expertise of the developers at hand.

Applications consisting of parts written in different programming languages have been called Multi-Language Software Applications (MLSAs) [9]. Each part and thus language is used to encode a particular aspect of the system. However, each of the parts usually also contains software artifacts which, due to certain properties such as their name or position, are relevant *across language borders*, being *bound* to artifacts in a different language. Binding of such artifacts usually happens at runtime by a framework or (virtual) machine.

Unfortunately, such multi-language bindings can lead to problems, both in the initial software creation phase and in maintenance, since they must be kept intact for the system to exhibit the proper behavior. Throughout development and maintenance as well as in program understanding, programmers must be aware not only of the semantics of the individual languages, but also of the semantics of the frameworks which handle the multi-language bindings which leads to added mental load. This is a particular problem in refactoring: If an artifact is changed without changing the referenced artifacts in other languages, the overall system semantics changes (and the system might break altogether). Worrying about such problems may lead developers to be hesitant about refactorings, which means the accumulation of technical debt [15].

We believe that a generic and systematic approach to supporting multi-language software systems can help to improve the situation for developers and increase productivity. In this work, we report on an investigation into such an approach within the Java ecosystem, i.e. in software systems which use Java as their main programming language and employ frameworks with domain-specific languages for implementing the non-Java parts of the system.

Our work contributes to the state of the art in three ways:

- we propose an approach for multi-language support in IDEs in which language artifacts and artifact bindings are handled as top-level entities,
- we present an implementation of this approach, including automated discovery and binding of artifacts at design time as well as rename refactoring across six languages from three frameworks,
- we evaluate our approach and tool on seven open-source case studies, giving empirical evidence of a) incidence of multi-language artifacts and bindings and b) accuracy of discovery, binding, and rename refactoring of our tool.

Our evaluation shows that we can indeed automatically discover, bind, and refactor artifacts in 3783 multi-language bindings — with full success in 95.96% of cases and with well-justified warnings to the user in the remaining cases.

2 Exploration Area and Motivating Example

We have selected the Java ecosystem as our area of investigation, that is software which uses Java as the main programming language and domain-specific languages provided by Java frameworks for encoding specific aspects of the system. The three areas, or domains, of *system configuration*, *database querying*, and *user interface design* each feature a number of such frameworks; from each, we have selected one framework for further investigation:

– The *Spring framework*[1] includes an XML dialect for configuring Java objects, in particular using dependency injection through JavaBean-style properties. We refer to this language as the Spring language.
– The *Hibernate OO mapper*[2] allows definition of the mapping between Java classes and database tables in another XML dialect (HBM). Additionally, the Hibernate Query Language (HQL) is used for querying the database (in the form of the defined entities). We refer to the first as the HBM language, the second as the HQL language.
– The *Wicket UI framework*[3] also contains two conceptual languages. The first is an extension to HTML used for defining HTML rendering templates. These templates are inflated and populated using a corresponding UI component tree providing dynamic data, which is defined in Java using the Wicket API, an internal DSL which we call Wicket/API. Bindings between Wicket/HTML and Wicket/API are established through the use of corresponding identifier strings.

Each of the languages in these frameworks offer two to five artifact types which are potentially bound to artifacts in other languages. These artifacts, which are shown in Figure 1, are defined and exemplified in the corresponding framework documentations and implementations.

The diagram shows the Java language in the center with the artifact types Constructor, Type, Parameter, and Method/Field (mostly, one or the other is used; therefore, they are shown together). These artifacts may be bound to three DSLs: Spring, HBM, and Wicket/API. From HBM and Wicket/API, additional multi-language bindings may lead to HQL and Wicket/HTML, respectively.

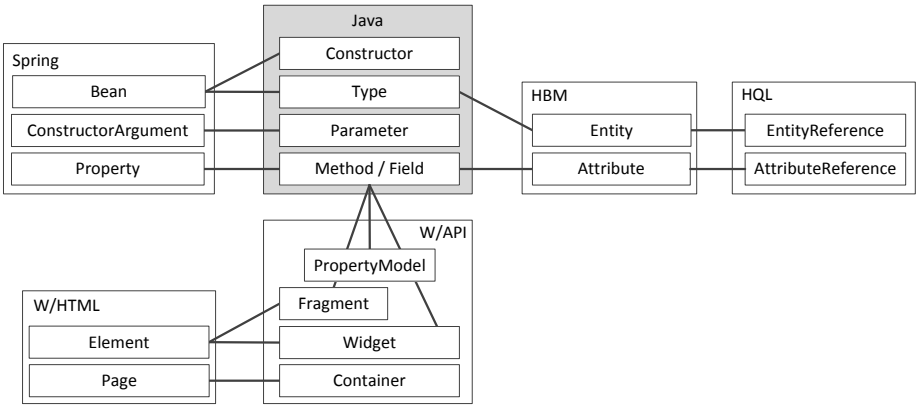


Fig. 1: Artifact Bindings Across Languages

[1] `www.springsource.com`
[2] `www.hibernate.org`
[3] `wicket.apache.org`

Listing 1.1: Hibernate Queries (HQL)

```
public class HibernateJtracDao {
  public int bulkUpdateStatusToOpen(Space sp,
                                    int st) {
    int c = bulkUpdate("update Item i set i.status=?
        where i.status=? and i.space.id=?",
      new Object[] { State.OPEN, st, sp.getId()});
  }
}
```

Listing 1.2: Hibernate Mapping (HBM)

```
<hibernate-mapping package="info.jtrac.domain">
  <class name="Item" table="items">
    <property name="status" column="status"/>
  </class>
  <class name="History" table="history">
    <property name="status" column="status"/>
  </class>
</hibernate-mapping>
```

Listing 1.3: Java

```
public abstract class AbstractItem {
    public Integer getStatus() {
        return status;
    }
    public void setStatus(Integer status) {
        this.status = status;
    }
}
```

Listing 1.4: Wicket Property & Widget Definition (Wicket/API)

```
private class ItemViewForm extends Form {
  public ItemViewForm() {
    History h = new History();
    setModel(new BoundCompoundPropertyModel(h));
    add(new IndicatingDropDownChoice("status",
              states, renderer));
  }
}
```

Listing 1.5: Wicket Page Fragment (Wicket/HTML)

```
<form wicket:id="form">
  <select wicket:id="status"/>
</form>
```

As a motivation for our work, we give an example from one of our case studies which shows how multi-language bindings look like in real life. The example, which is shown in Listings 1.1 to 1.5, shows bindings between Java, HBM, HQL, Wicket/API, and Wicket/HTML.

We begin with the property `status` in Listing 1.3, which is defined with JavaBean-style getters and setters in the abstract class `AbstractItem`. This class is the common superclass of the classes `Item` and `History` (which are not shown). However, these two classes are also Hibernate entities and defined as such in Listing 1.2 along with a `status` property each, which is bound to the getters and setters for `status` as shown. This Hibernate property is used in HQL queries in Listing 1.1; in this case, on the entity `Item` or rather its alias `i`.

The property `status` of the `History` subclass of `AbstractItem`, as defined in Listing 1.3, is also referenced from Wicket/API (Listing 1.4, line 4) through a `BoundComponentPropertyModel`, where `History` is defined as a data source for the `DropDownChoice` widget in line 5. The `status` string used in line 5 is also used as a widget identifier between Wicket/API and Wicket/HTML; the latter is shown in Listing 1.5.

Thus, the example shows a total of nine multi-language bindings between eight artifacts across five languages (or one general-purpose language and two frameworks). If any of these elements is renamed, all of the others must be renamed as well since the bindings are created (among other things) due to name resolution. It is additionally to be expected that the getters and setters from Java are also referenced elsewhere such that any invocations must be refactored as well. The same applies to any other use of the Hibernate HBM property definition and the ID of the Wicket/API widget.

To evaluate automated discovery and rename refactoring of such bindings, we employed seven open-source case studies with a size between 6k and 110k LOC (see Table 1). Two of the case studies use mainly Spring; two use mainly Hibernate; and two use mainly Wicket. A seventh uses all three frameworks in combination. We discuss how the case studies were used for evaluation in Section 4.

3 Multi-Language Artifact Binding and Rename Refactoring

The use of multiple additional languages in Java software systems is not new. Accordingly, existing Java IDEs already contain some support for finding artifacts bindings and for refactoring across language borders. However, such solutions are usually rather isolated: There is sporadic support in the form of IDE plug-ins for particular frameworks such as Spring or Hibernate, for example for Eclipse. There are three major drawbacks to current implementations.

Firstly, there is no generic support for multi-language artifact binding *per se*; that is, IDEs are generally unaware of such bindings unless a plug-in contributes them, in which case each plug-in must provide its own data structure, navigation menus, and so on.

Secondly, current Java IDEs (Eclipse, IDEA, NetBeans) use a *participant* approach to refactoring non-Java elements: Usually, DSL refactorings are implemented as add-ons to existing Java refactorings instead of refactorings in their own right since there is specific support for such participation. By contrast, implementing completely new DSL refactorings with participant support of their own involves significant additional effort. Also, refactoring changes might need to be propagated back and forth between languages as new bindings are identified which is also difficult to implement using current participant approaches (first, participants need to be enabled and disabled depending on where a refactoring is started; second, changes must be gathered in a sort of feedback loop to ensure that every participant may react to changes by others).

Finally, there is no systematic support for handling more than two languages, in particular if they are not directly bound to Java. As indicated above, the artifacts of some DSLs (HQL) might bind to artifacts of other DSLs instead of Java (namely HBM), thus creating cascades of bindings (and thus refactorings).

We believe that a generic, systematic approach to multi-language artifact binding and refactoring will make it easier to implement MLSA support in IDEs and thus lead to better support for developers. We thus investigate an approach to handling multiple languages in IDEs which

- treats all languages, including DSLs, as equals and offers the infrastructure to make artifacts of each language centrally available,
- defers handling of binding resolution between each of the languages to dedicated binding resolvers,
- and allows per-language refactorings to trigger, and be triggered from, generic refactoring routines which propagate changes based on artifact bindings.

We discuss this approach, and a prototype implementation within Eclipse, in the next three sections. Note that the tool we have implemented is not intended to be a product; rather, its aim is demonstrating feasibility and generating the data about real-life software analysis and refactorings.

Section 3.1 discusses artifact discovery, i.e. reading source code and providing artifacts. Section 3.2 discusses the dedicated binding implementations which resolve multi-language bindings. Finally, Section 3.3 discusses how refactorings across language borders are implemented.

3.1 Artifact Discovery

Our approach is based on automated discovery of artifacts relevant for multi-language bindings. With the term *artifact* we refer to a representation of a concept used in the source code, such as, for Java, `TypeDeclaration`s which in turn contain `MethodDeclaration`s, `LocalVariableDeclaration`s, `Statement`s, and so on. These artifacts form a *semantic model* [5] in which references or in-language bindings between them are already resolved; for example, a Java `MethodInvocation` is bound to its `MethodDeclaration`.

The benefit of using semantic models is a simplification of the navigation within the code base as well as uniquely identifying artifacts; it is furthermore

useful in code analysis and visualization and, in our case, in separating the difficulties in finding artifacts within a language with other multi-language concerns.

A semantic model must be extracted from the source code — a parser is required as well as resolution mechanisms for in-language bindings. We call this process *model discovery*; for each language, a model discoverer must be written and registered with the platform. In our approach, we use one meta-model per language, not a single language-agnostic model. Analysis is later done by analyzing relationships between pairs of individual language models.

Required Effort The effort required for model discovery routines depends entirely on the language which is analyzed. Regarding difficulty, we can separate the languages we have investigated into two groups. The first group contains Java (the base language, without data flow), Spring, Hibernate Mapping (HBM), and Wicket/HTML. In these languages, it is relatively easy to extract the structure and all artifacts from the source code; furthermore, in-language bindings between elements may be complex, but are based on straightforward and exact rules. Parsers with in-language binding resolution already exist for Java; Spring and HBM are XML-based files such that existing XML parsers may be used to extract data, with a follow-up of in-language binding resolution implemented by hand.

The second group contains HQL and Wicket/API; in other words, languages whose source fully or partially consist of strings handled by Java statements. In both cases, identifiers or query fragments may be combined using an arbitrary number and combination of loops, decisions, and values passed in from the outside. In case of Wicket/API, data and control flow is furthermore used to (manually) construct a UI tree out of objects in memory. This tree of Java objects is required to correspond to the tree created in HTML out of HTML widgets.

Finding and resolving all HQL and Wicket identifiers and artifacts is generally undecidable in the environment they live in, i.e. in a general purpose programming language. We have gone to great length in the model discovery for these languages; however, some elements could not be extracted which in turn leads to various problems in artifact binding and refactoring.

An example of this from one of the case studies is shown in Listing 1.6, where the variable `field` is used in the construction of the query. The contents of `field`, in this case, cannot be resolved in general since it represents a custom contributed database table column. Thus, it is known that *some* attribute of `Item` is accessed, but not which one. In such cases, a specific *unresolved artifact* is added to the model to make this problem explicit (this information is later used to add refactoring warnings).

Listing 1.6: Dynamic Query in HQL

```
1    bulkUpdate("update Item item set item." +
         field.getName() + " = null");
```

Besides unresolved elements, there may also be *orphan artifacts*. Contrary to the example above, these are artifacts whose reference *from* other model elements cannot be resolved — that is, for example, a widget creation is found in the code, but it is unknown to which page or parent element it belongs. Like unresolved elements, these are reported explicitly in the model.

Static Analysis For discovering HQL and Wicket artifacts embedded in API calls, we have created a custom static analysis approach on top of the MoDisco Java semantic model[4]. In terms of data flow analysis, our approach is interprocedural (in that it treats method invocations non-atomic) and flow-sensitive (in that it considers the order of statements).

The domain over which our analysis operates is the domain of method and object environments: we keep track of approximations of valuations of stack and heap variables (i.e. local variables and object fields). The values we approximate are strings for HQL queries as well as custom representations of framework data structures such as UI trees for Wicket and query trees for HQL. The transfer functions we use depend on the source of the code they represent: for application code, we use standard transfer functions that correspond to the semantics of Java. For invocations of framework methods, we use custom transfer functions on the representations of framework data structures. For instance, the transfer function for line 5 in Listing 1.4 creates a new drop down choice component, and adds it as child to the `ItemViewForm` component. Invocations of framework methods that are not analyzed (for example, the Java Collections framework) are treated as atomic and ignored.

Our static analysis only performs a single pass over a sequence of program statements following method invocations, and has therefore two pragmatic limitations: firstly, loops found in program code are handled by performing a single pass of the loop body. Secondly, recursive methods are handled by ignoring back-edges in the call graph, i.e., by skipping recursive method calls. Because of these limitations, our static analysis will miss fabricated identifiers and over-approximates artifacts in looping and recursive program code. In the context of model discovery for multi-language artifact binding, however, this is no severe limitation, as every multi-language binding must point to a corresponding static name in another language, and thus fabrication of names is not encouraged in any of the analyzed frameworks.

The analysis routines we implemented proved sufficient to discover the vast majority of artifacts relevant to multi-language artifact binding, as discussed in the next section. Since artifact discovery in HQL and Wicket/API is only a partial aspect of our work, it was not our goal to provide full coverage. In particular, we did not go as far as creating grammars for string expressions found in Java code as in Christensen et al. [2]. Also, we do not provide support for the Java Collection Framework and do not analyze uses of Java Reflection as in Livshits et al. [10].

[4] `www.eclipse.org/MoDisco`

3.2 Multi-Language Artifact Binding Resolution

Multi-language binding resolution is concerned with associating artifacts from two languages with one another, based on the rules of the underlying framework (such as Spring). Considering our example in Section 2, the two HQL `AttributeReference` artifacts with name `status` must each be bound to the HBM `Property` with the same name.

The name of an artifact is, in most cases, an important aspect of binding, but it is nearly never the only one: All languages we have investigated are strongly hierarchical; thus, the position of artifacts within such hierarchies is crucial. In our example, the HQL attribute `status` only refers to the `status` property of class `Item`; not any other class. It is obvious that a purely textual search will fail in most cases given such a structure. Furthermore, frameworks usually give developers quite a lot of freedom, i.e. different ways of achieving the same thing, optional bindings, or double meanings for identifiers. The binding resolution routines must take these into account and thus depend on the positioning of elements, attribute value grouping, different naming conventions, and so on. Multi-language binding resolution fails if an artifact is found in one language but its required complementary artifact — based on all the framework rules and options — in another language is not.

For each interesting language pair, we have created dedicated binding resolvers which each use their own custom resolution algorithm. These algorithms are based on the binding logic of the underlying frameworks (i.e. Spring, Hibernate, and Wicket); we have discussed some of these algorithms in detail in [11]. As an example, the binding resolver for Spring first binds Spring beans (from the Spring artifact model) to Java classes (from the Java artifact model) based on fully qualified class names. Afterwards, it binds nested Spring properties to members in the previously bound Java classes based on simple names.

The binding resolution algorithms in our approach thus always bind artifacts of two languages together. This yields five resolution implementations for the six languages we have investigated: Spring and Java, HBM and Java, and Wicket/API and Java are the ones grouped around Java; while HQL and HBM and Wicket/HTML and Wicket/API deal with DSLs on both sides.

The binding results in the form of individual artifacts and the links between them are reported in a common language-agnostic *linking model*. This model later allows identifying the necessary rename operations for a change; its contents are centrally available on the IDE level; thus, two binding resolvers may bind the same Java artifact into different languages, which is what happens, for example, with the `status` property in the example (Listing 1.2). In this way, the IDE can support navigation from artifact to artifact and is aware of transitively connected artifacts; in the case of binding errors, it can report and annotate the offending artifact.

As in model discovery, writing a binding resolver requires effort. While the code implementing the actual resolving is already part of each framework implementation (e.g., in Spring, Hibernate, and Wicket), it is written with a focus on runtime and thus not easily extracted. In fact, in our case, we have

re-implemented all resolution code by hand based on the framework documentation and the available framework code. While a good knowledge of the framework involved is certainly a requirement for writing a binding resolver, in most cases the logic, though complex, is not overly difficult and thus does not require a large investment.

3.3 Multi-Language Rename Refactoring

The last part of our approach is support for multi-language rename refactoring. Many refactoring procedures have been defined in the literature [4]. Of these, the most important ones across language borders are rename refactorings, to which we restrict ourselves in this work. These are also the most commonly used automated refactorings: A study in 2012 has shown 44% of all tracked refactorings to be rename refactorings [22].

Renaming artifacts even in one language can get very complex (especially in Java [18]); thus, we believe it is best to re-use existing refactorings rather than implementing new ones for multi-language refactorings. In our work, we therefore assume that each language comes with its own set of automated rename refactorings. This is certainly true for Java in most IDEs; it is less true for the DSLs we have looked at. For some, plug-ins are available which add this functionality, for some, there are not. In the latter cases, we have implemented single-language rename refactorings by hand to ensure an equal setup for all languages (fortunately, the selected DSLs include only limited amounts of in-language bindings and thus the refactorings are rather simple).

Relating Artifact Names To support rename refactoring across language borders, we need to know how the names of the artifacts relate. Using the information from the multi-language artifact binding, we already know which artifacts are bound across language borders. The binding resolvers discussed in the last section also have the information which properties of these artifacts carry the names relevant for the binding, and how the names are related; to support the refactoring step, this information needs to be attached to the bindings (on a meta-level).

In some cases, the relationship is very simple: If a Spring property is bound to a Java field, the names must match exactly. In other cases, some transformation takes place: If the property is bound to a setter, for example, the Java method name must be prefixed with `set` and the first letter of the property name must be uppercased (adhering to the JavaBean convention).

Changing Artifacts Refactoring usually starts from a single change of an artifact property (such as a Java method name) which is triggered by the user. Once known to the multi-language enabled IDE, this change can now be propagated through the multi-language bindings to all artifacts which are (transitively) bound to the artifact in which the change originated, i.e., to its *transitive binding closure*. This closure can be found generically without involving language-specific routines. Note that we may move back and forth between languages in this process: If, for example, we start with `Item.status` from Listing 1.2 and move to

Java, the fact that `status` is not defined in `Item` but in a superclass and is used also in `History` requires us to move back to HBM (and perhaps even HQL).

The resulting transitive closure may contain artifacts from many different languages, each annotated with the information which properties must be changed to which new value. The actual source code-changing rename can now be performed by language-specific refactoring routines as discussed above, which may lead to additional in-language binding renames. For example, renaming a Java method may involve renaming all of the method invocations, which are not relevant across languages but certainly relevant within Java.

Error Conditions Multi-language refactorings can only be executed on artifacts with established multi-language bindings (otherwise, the original artifact is annotated as having a binding error during binding resolution). Still, a refactoring may not always be possible.

Firstly, there are conditions in which we must inform the user that we do not have sufficient information to guarantee a successful refactoring execution: Such *warnings* result from problems in artifact discovery. As mentioned in Section 3.1, there may be cases where we know that there is an artifact reference present but not exactly which one, indicated through the presence of an *unresolved artifact*. In this case, a warning is attached to the refactoring (as, e.g., the Java rename refactoring does in the presence of parsing errors).

Secondly, any of the single-language refactorings invoked may veto a change (for example, due to a restricted name in a language). In this case, the overall refactoring is aborted with an *error*.

4 Experimental Evaluation

The underlying rationale for automated multi-language binding and refactoring support is improving developer productivity. However, productivity is hard to measure directly. We have thus instead opted for measuring a surrogate endpoint, which is given by the fitness for the particular purpose of our approach; in particular that our tool is able to *automatically* and *correctly*:

- identify multi-language relevant artifacts in each individual language,
- identify and establish the bindings between said artifacts,
- refactor the previously bound artifacts across multiple languages.

Our assumption is that a developer with such tool support is more productive than without. It is future work to test the surrogate against the actual endpoint with adequate empirical studies. In the following, we discuss the setup, execution, and results of our evaluation.

4.1 Experimental Setup

Case Study Setup We selected seven open-source applications which make use of Java and at least one of the three frameworks we support. Our sampling process was as follows: After having selected the frameworks to investigate

(Spring, Hibernate, and Wicket), we performed a search for applications using these frameworks (two for each) in the ohloh.net repository. Unfortunately, the population size was quite small, i.e. we did not find many applications which were a) not trivially sized, b) not frameworks or framework extensions themselves, and c) in a compilable and unit-testable state. We selected the first seven applications for which a) to c) were satisfied.

Table 1: Case Studies

Case	**Domain**	**Languages**	**Version**	**LOC**
Plazma	ERP+CRM solution	Spring	1.0.2	78k
Tudu Lists	Todo Lists Management	Spring, Hibernate	2.3	6k
itracker	Issue Tracker	Hibernate, Spring	3.1.5	110k
PicketLink	Identity Management	Hibernate	1.3.1	42k
Brix	Content Managament	Wicket	2013-08-21	31k
gidooCMS	Content Management	Wicket	2013-08-21	10k
JTrac	Issue Tracker	All	2.1.0	14k

The cases are listed in Table 1. If possible, the latest stable version was downloaded either from the repository or from a release website. In the case of gidooCMS and Brix, no such version was available; a snapshot was instead taken from the repository. The cases were prepared for analysis as follows:

- Since our tool is Eclipse-based, all case studies were converted to Eclipse projects such that the code compiles with the JDT and unit tests could be run within the Eclipse environment, and without any other build system (such as Maven).
- Since we use the MoDisco discoverer for Java, all classes relevant for discovery must be available as source code; the relevant library classes were thus extracted and added as source files (and removed from the libraries).
- For refactoring testing, we require a dense test net for all multi-language relevant artifacts. Where such was not available, we have implemented additional tests by hand.

Creating tests for each case study ranged from trivial to demanding. In the case of Spring and HBM, it was sufficient to instantiate the frameworks since they perform start-up tests to ensure integrity of the bound artifacts. However, for HQL as well as for Wicket, no such start-up test mechanisms were available. However, executing the code containing the artifacts leads the framework to fail; thus, tests were implemented to cover all the relevant lines of code.

Tool Setup Our tool provides three types of results: artifacts, bindings, and refactoring closures. Firstly, the artifact discovery results in a list of all artifacts potentially multi-language relevant which are available in a project. Since it is not possible to test the correct discovery of these artifacts automatically, we have created tool support (in addition to orphan detection) specifically for the

task of annotating artifacts and have manually read through the source code to ensure that all relevant artifacts were covered.

Secondly, the binding resolvers discover relationships between artifacts from different languages. As above, these bindings need to be checked by hand. Again, we have created specific tool support for the visualization of such bindings, which allowed us to manually iterate over the bindings, checking both successful bindings and error cases.

The final result are transitive artifact closures and their refactoring. In this case, we have opted for automated verification in the same spirit as in "normal" refactorings: By using unit tests. As mentioned in the previous chapter, refactorings are executed on the transitive closures of refactoring changes, i.e. on a group of artifacts which need to be renamed together. To verify whether the renamings performed are correct, the following process was run for all closures, one after the other:

- All unit tests were first run on the unchanged source code. The tests were expected to pass.
- In the second phase, artifacts were grouped by language. One by one, these artifact groups were renamed individually, i.e. first the Java artifacts, then the Spring artifacts, and so on. After each rename, the tests were run and expected to fail to ensure that renaming the artifacts individually actually introduces problems. After each test, the changes were undone again.
- Finally, all artifacts in the current closure were renamed and the tests were run one more time; in this case, they were expected to succeed. This change was undone as well before proceeding to the next closure.

This process is a very thorough test on many parts of the framework: It requires that the correct artifacts are grouped into closures; that each artifact is correctly resolved and has the right source code location attached; that each individual language refactoring is triggered correctly; and (on the case study side) that there are indeed tests which cover the multi-language bindings of the closure.

4.2 Results

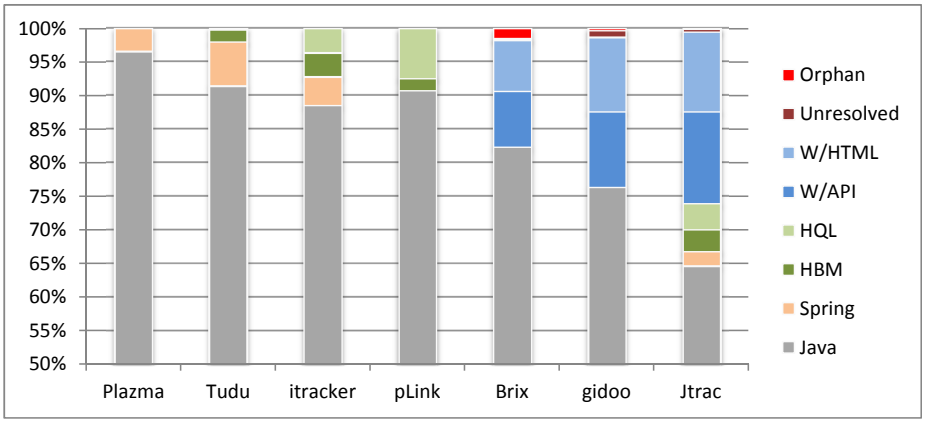
Artifact Discovery The result of artifact discovery is a list of potentially multi-language relevant artifacts (see again Figure 1). The discovery results are shown in Table 2 (note that the figure starts at 50%, the remainder being Java artifacts).

Since all of the cases are written using Java as the main language, the percentage of Java artifacts is rather high. In Java, potentially relevant artifacts include all types, public methods, fields, etc. Considering this, it is quite interesting to see that DSL artifacts amount to as much as they do (from 3.47% to 34.11%, mean 12.67%).

The *unresolved* and *orphan* rows in Table 2 show artifacts which could not be fully discovered. These numbers only occur in the projects which make use

Table 2: Discovered Artifacts for each Case Study per Language

Language	**Plazma**	**Tudu**	**itracker**	**pLink**	**Brix**	**gidoo**	**JTrac**	**Total**
Java	12 525	1083	4922	2963	5342	1469	3027	**31 331**
Spring	450	78	236	0	0	0	102	**866**
HBM	0	21	198	59	0	0	150	**428**
HQL	0	3	205	244	0	0	182	**634**
W/API	0	0	0	0	574	224	648	**1446**
W/HTML	0	0	0	0	368	183	466	**1017**
Unresolved	0	0	0	0	18	19	19	**56**
Orphan	0	0	0	0	92	6	0	**98**
Total	**12 975**	**1185**	**5561**	**3266**	**6394**	**1901**	**4594**	**35 876**



of HQL and Wicket/API and are indeed related to these two languages. As discussed in Section 3.1, it is not possible in these two languages to statically discover and properly place each element in the language models.

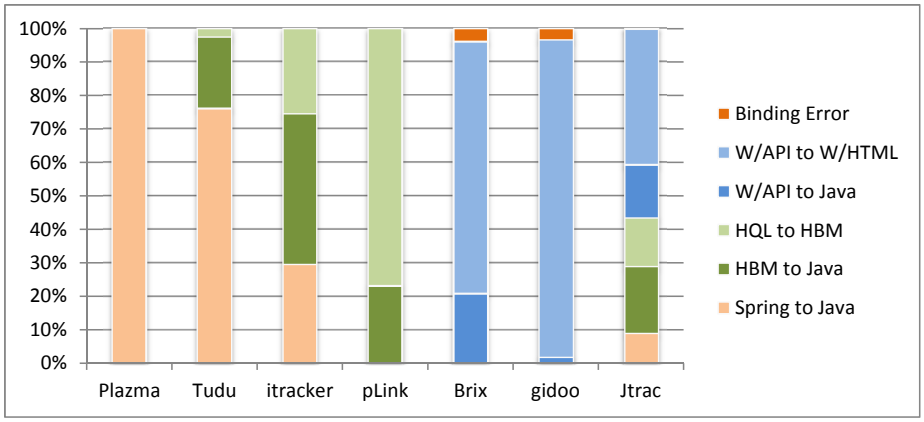
First, unresolved elements (56) refer to cases where a reference to an artifact was found but its value could not be determined — as in the HQL query example in Section 3.1. These elements are later important in refactoring, where they induce warnings for refactorings; however, due to their incomplete nature (in particular, their lack of name or identifier) they can not be bound.

Second, orphan elements (98) refer to cases where elements were found but their context, contrary to unresolved elements, is unknown such that a reference could not be added to the model. Multi-language binding resolution is not possible with orphans, thus no navigation is available and the orphans cannot take part in refactoring. In all of the cases reported here (as manually verified), the elements would not have been resolved across languages or taken part in refactoring anyway: Most of the orphans lie in Wicket JUnit test case implementations, where their IDs are ignored (54 cases); the remaining occurrences either use passed-in IDs (39) or lie within Wicket library code (5). Thus, there is no further impact of missing these elements here (this may obviously be different in other cases).

Artifact Binding Resolution In binding resolution, artifact bindings from one language to artifacts from a second language are resolved based on the rules of the underlying framework. The resulting numbers are shown in Table 3.

Table 3: Discovered Multi-Language Bindings for each Case Study per Resolver

Resolver	**Plazma**	**Tudu**	**itracker**	**pLink**	**Brix**	**gidoo**	**JTrac**	**Total**
Spring to Java	517	89	241	0	0	0	123	**970**
HBM to Java	0	25	151	65	0	0	271	**512**
HQL to HBM	0	3	429	215	0	0	199	**846**
W/API to Java	0	0	0	0	108	4	216	**328**
W/API to HTML	0	0	0	0	382	192	553	**1127**
Total	**517**	**117**	**821**	**280**	**490**	**196**	**1362**	**3783**
Binding Error	0	0	0	0	26	11	5	**42**



Again, due to the nature of the underlying case studies, we see a progression from Spring-related bindings via Hibernate-related bindings to Wicket-related bindings from left to right, with the exception of JTrac on the far right which uses all frameworks. In total, we have found 3783 bindings between elements in all case studies; the average number of bindings per case is 540. Considering that these bindings must be intact for the software to work correctly and are only partially supported by tools, possibilities for things to go wrong abound. From a purely statistical point of view, in the case with the most bindings (JTrac), this would amount to *one multi-language binding every 11 lines of code.*

In the three projects to the right, we see a total of 42 binding errors; as mentioned in Section 3.2, a binding error is reported if an artifact was found without its expected corresponding artifact in another language. All errors are reported on HQL and Wicket artifacts.

In 14 cases, these errors are actual problems in the observed code (JTrac: 2 / gidooCMS: 5 / Brix: 7), that is, an artifact should indeed have a partner in another language but did not. The other 28 cases are erroneously reported errors and can be separated into three categories. The first category is the originally expected one: Binding errors due to unresolved artifacts, of which there are

only two (0/2/0). The second category contains problems due to the if-then-else over-approximation in the HQL and Wicket/API static analysis; i.e. some artifacts are present in more than one position in the model of which some are inaccurate (3/0/15). The third category contains missing bindings which are due to references in Wicket library code (0/4/4) which is due to the use of MoDisco.

Thus, all 28 erroneous reports (0.74%) are due to shortcomings in the static analysis. The two binding errors due to unresolved artifacts will lead, additionally and independently, to refactoring warnings. The others, which are due to over-approximation or use of library code and refer to "missing" artifacts do not carry identifiers and thus have no further impact on refactoring.

Refactoring Change Closures If an element, or rather a name property of an element is selected for a rename refactoring, all existing multi-language bindings must be traversed to find the transitive closure of artifacts which are affected by the rename, and which must be changed as well. In the following, we first report on results from this traversal. Afterwards, we present the results of actually executing refactorings for each closure.

Closure Discovery Results The results from the discovery of closures and thus the incidence of languages, artifacts, and edits is shown in Table 4.

Table 4: Discovered Transitive Refactoring Closures per Case Study

Case	**Plazma**	**Tudu**	**itracker**	**PicketLink**
Closures	122	72	214	55
Ø Languages	2.00 ± 0	2.03 ± 0.17	2.28 ± 0.45	2.56 ± 0.50
Language Max	2	3	3	3
Ø Artifacts	4.69 ± 12.00	2.21 ± 0.47	4.32 ± 5.54	6.09 ± 6.08
Ø Edits	4.28 ± 12.06	4.42 ± 12.41	28.58 ± 68.32	16.93 ± 19.47
Artifacts > 2 (%)	4.92	18.06	69.63	72.73

Case	**Brix**	**gidoo**	**JTrac**	**Total**
Closures	398	174	674	**1709**
Ø Languages	2.05 ± 0.21	2.00 ± 0	2.14 ± 0.48	$\mathbf{2.15 \pm 0.39}$
Language Max	3	2	5	**5**
Ø Artifacts	2.22 ± 0.79	2.13 ± 0.79	2.90 ± 2.70	$\mathbf{3.51 \pm 4.39}$
Ø Edits	3.49 ± 2.91	3.60 ± 5.08	6.07 ± 17.04	$\mathbf{9.62 \pm 28.25}$
Artifacts > 2 (%)	11.56	4.02	32.34	**28.03**

Overall, the number of transitive closures range from 55 (PicketLink) to 674 (JTrac); the total is 1709 closures with an average of 244 closure per case. The average of artifacts in all closures is 3.51; however, with a standard deviation of 4.39. In Plazma, itracker, PicketLink, and JTrac, the standard deviation is quite high, while in Tudu, Brix, and gidooCMS it is quite low. The maximum number of artifacts in a closure is 61 (in Plazma), where the Spring property `dataSource` is renamed; this property is injected into 60 beans which share a common setter method (hence 61 artifacts).

An interesting observation is the number of closures in which there are more than 2 artifacts. Here, the case studies seem to fall into three groups: The first group includes itracker and PicketLink with around 70% of closures with more than 2 artifacts; the second group consisting of just JTrac with around 30%, and finally all others with less than 20%. Since itracker and PicketLink mostly use HBM/HQL and JTrac has a HBM/HQL part, we have investigated whether this phenomenon is language-specific. Figure 2 shows closure artifact counts per language (that is, size of closures which have artifacts in the given language), where it becomes clearly visible that we mostly deal with only 2 artifacts in Spring, Wicket/HTML and Wicket/API. In HBM and HQL, however, the majority of closures have 3 or more artifacts.

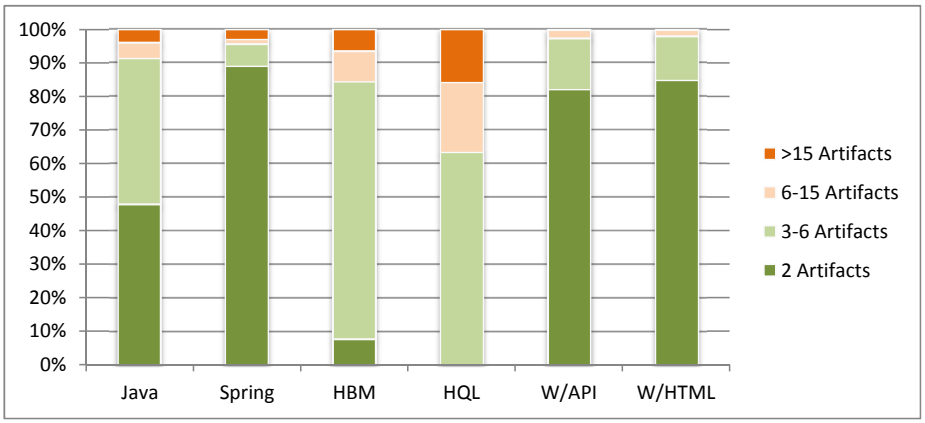


Fig. 2: Artifacts per Closure and Language

Another observation regards the number of languages involved in the closures also shown in Table 4. In four case studies, we mostly or exclusively deal with two languages per closure: In Plazma, only Spring and Java are involved; in gidooCMS, only Wicket/API and Wicket/HTML. There are two closures with HQL and HBM artifacts in Tudu and 19 closures with Java in Brix, which is why the maximum number of languages is three in these cases. The highest number of languages involved in one closure is 5 in JTrac, which nevertheless only has an average number of 2.14 languages per closure. The remaining two case studies lie in the middle with 3 maximum languages and an average of 2.28 (itracker) and 2.56 (PicketLink) artifacts per closure. An example of a closure with 5 languages (from JTrac) has already been shown in Section 2 (Listings 1.1 to 1.5).

An inverse representation of these numbers is shown in Figure 3 which shows the percentage of closures with 2 to 5 languages having at least one element from the chosen language. This correlates with the number of artifacts: In Spring, Wicket/API and Wicket/HTML, closures mostly deal with two languages. In HBM, over 40% of closures deal with three and more languages. As expected, HQL-affected closures always use at least three languages (HQL, HBM, and Java), with extensions into Wicket/API and Wicket/HTML.

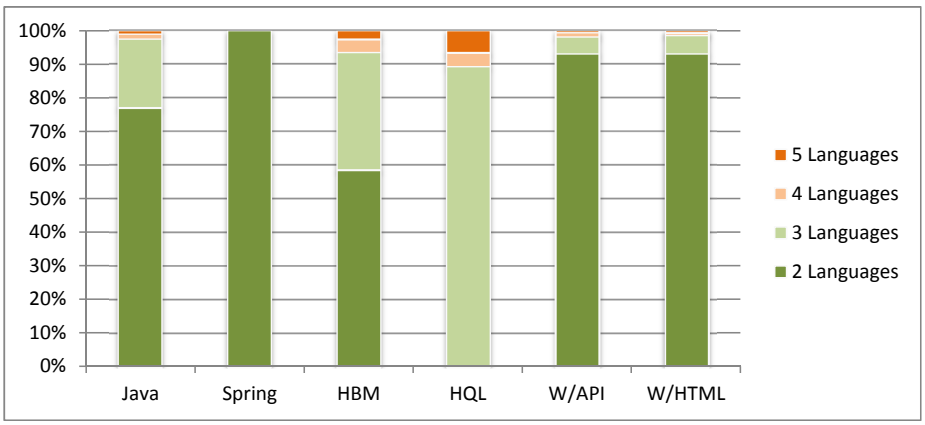


Fig. 3: Languages per Closure

As an insight into the technical underpinnings, Table 4 also gives the average number of actual text edits performed per closure, which also varies greatly. This number depends not only the number of artifacts relevant for multi-language bindings, but also on all the additional changes the per-language refactorings had to add. The maximum number of text changes for a closure is 620 (in itracker), where an `id` attribute is renamed: This attribute is defined in an abstract entity superclass and is thus used in each HBM entity definition as well as all HQL queries for any of these elements; furthermore, the `id` getters and setters in Java must be renamed which are again heavily used in the code base.

Refactoring Results The results from executing the refactoring actions on all of the closures found is shown in Table 5. Note that the figure starts from 80%, the remainder being successful tests.

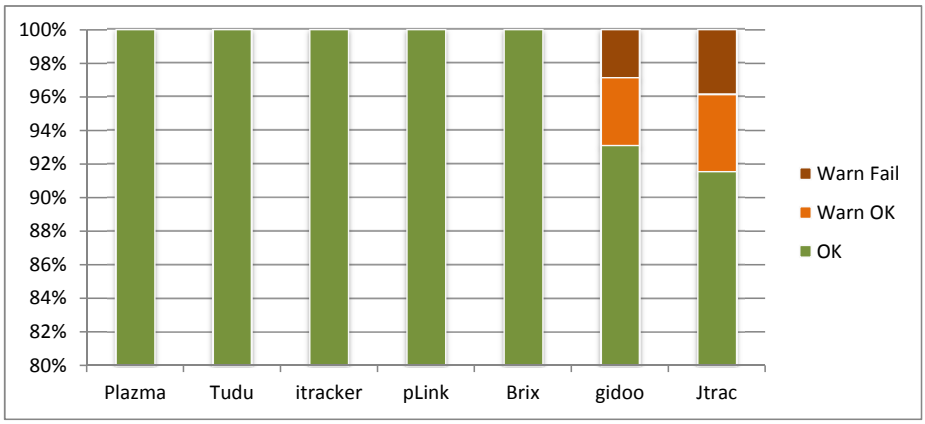
The results show that the tests after refactoring each closure succeeded in 95.96% of cases across all case studies (1640 of 1709 closures). In the remaining 69 cases, a warning was attached to the closure (based on incomplete artifact discovery, as discussed above); in 38 of these cases, the tests still succeeded; thus, the warning was unnecessary; in 31 cases the test failed, thus the warning was accurate. No closure test failed without a warning being attached. In all cases, problems are again down to artifact discovery in HQL and/or Wicket/API; in all of these cases, we can warn the user of potential problems.

The last row in Table 5 shows the number of closures in which at least one single-language tests succeeded. Recalling from the experimental setup, each closure was refactored multiple times: Firstly, each language on its own (which should lead to test failures), and finally all together (which should succeed). Thus, it is interesting to look at why the single-language tests succeeded despite having not renamed all elements.

First, there is one succeeding test in PicketLink on an HQL attribute called `binaryValue`. This attribute is used in an unreachable part of code which thus could not be tested. In Brix, gidooCMS, and JTrac, most (56) of the test suc-

Table 5: Results from Refactoring Transitive Closures

Language	**Plazma**	**Tudu**	**itracker**	**pLink**	**Brix**	**gidoo**	**JTrac**	**Total**
OK	122	72	214	55	398	162	617	**1640**
OK (warned)	0	0	0	0	0	7	31	**38**
FAIL (warned)	0	0	0	0	0	5	26	**31**
Total	**122**	**72**	**214**	**55**	**398**	**174**	**674**	**1709**
Single Language Success	0	0	0	1	17	3	39	60



cesses refer to the Java language and are due to fallback behavior of Wicket: If a Java getter/setter is not found for a certain entity, Wicket looks for a field of the same name. Since we only rename the JavaBean-style getters and setters and all relevant fields have exactly the same property name, the field is still found by Wicket and thus the tests succeed. Manual renaming of the field leads to test failures in every case. In the remaining three cases (two from Brix, one from gidooCMS), the success affects both Wicket/HTML and Wicket/API. In the first two cases, the references are overwritten by generated HTML code (Brix), in the second, we deal with dead code (gidooCMS); thus, they are not testable.

4.3 Discussion

Our experimental evaluation has covered six languages across seven case studies; each language was present in at least three case studies.

The artifact discovery process has reported 35876 multi-language relevant artifacts across all case studies. As discussed, 154 these (56 unresolved elements, 98 orphans) from the languages HQL and Wicket/API could not be extracted due to static analysis limitations in these languages.

In the follow-up multi-language binding resolution, 3783 unique bindings between artifacts across languages have been automatically resolved. Again due to limitations in the static analysis, some bindings have been erroneously reported as missing in 18 cases (0.74%).

Regarding the refactoring step, we were able to automatically refactor all 1709 closures found. The test run results after refactoring the closures show

a success rate of 95.96%, i.e. in 1640 of 1709 closures, the tests after a full refactoring succeeded. The remaining 69 cases were attached with warnings due to incomplete artifact discovery; in about half of them, the refactorings and subsequent tests succeeded despite warnings.

The single-language test runs which were supposed to show that single-language changes do not suffice only succeeded in 60 cases, of which most are due Wicket's fallback mechanisms; the remaining four are due to unreachable or dead code. This clearly shows that the system functionality, as far as the tests are concerned, is only kept intact by multi-language refactoring.

Thus, we believe that the fitness for the intended purpose of our tool (i.e., discovery, binding, and refactoring for multi-language software applications) has been established. The automated refactoring and testing approach we have used in this experimental evaluation has been helpful in reaching this goal, in debugging, and in establishing trust in our own system, since all aspects of the system must work together to lead to succeeding tests after refactoring.

The numbers also show that, in particular regarding HQL and HBM applications, more than two artifacts and languages are involved in multi-language transitive refactoring closures. We believe that this shows the need for generic support for languages and language bindings instead of a participant-based two-language approach.

Finally, the implementation and validation of our tool has shown which language features are particularly difficult to support. In fact, all artifact discovery, binding, and refactoring problems ultimately originate in the languages HQL and Wicket/API and the fact that the statements of these languages are embedded in Java and it is allowed to manipulate them using Java control flow constructs, necessitating extensive (and expensive) static analysis. Compared to the length to which one has to go to to support such languages, supporting external languages such as Spring, HBM, or Wicket/HTML is almost trivial.

It is interesting to ask the question if the ability to manipulate the language constructs in HQL and Wicket/API in this manner is really necessary, i.e. if it is crucial to the usability or fitness for purpose of these languages. This is to be investigated in the future. If no reasons can be found, we recommend not allowing such manipulations in future languages, and suggest using external, clearly separated languages instead.

Although our tool showed good results on the languages we investigated, it is unclear how these results translate to other languages. Several frameworks within the Java world deal with similar domains (i.e. system configuration, querying, and UI); these would present a good starting point for further evaluation.

Regarding refactorings, a possible conceptual difficulty of our approach lies in the re-use of existing refactorings, which might change elements in the source code, affecting artifacts that are part of another multi-language binding. Since this change is encapsulated behind the refactoring, it is not possible to react before the change has been committed. Note that this is only a problem if the artifacts changed are not bound in the semantic model as well (otherwise, the

propagation algorithm would have found them). We have not encountered this problem in our test cases, but it is conceivable that such situations may occur.

4.4 Threats to Validity

Our claim is the fitness for the intended purpose of our tool, i.e. that artifact discovery, binding, and refactoring works as expected when tested on real-life cases for the languages involved. Obviously, we can only claim this for the seven case studies we have investigated; however, we believe that they represent a good spread of cases; we have also taken care to implement a general solution. Still, other cases may lead to different and possibly more error cases in each of the areas of artifact discovery, binding, and refactoring.

Regarding our evaluation, artifact discovery and binding resolution have been executed manually, i.e. we investigated the source code to determine whether discovery and binding were accurate. We have created and used tool support specifically for this task for artifact annotation and orphan detection and have taken care to find all elements; however, it is still possible that we have missed artifacts and bindings during this process. Refactoring success was tested by JUnit tests; although we have checked coverage regarding the artifacts found, some closures may still be incomplete without us noticing. In the other direction, some closures may also be too extensive, i.e. include elements which would not have needed renaming. Since the renaming and tests of the individual language artifact groups failed as described, there is a strong indication that all bindings in each closure are relevant; however, we did not unit-test each binding individually.

Since our model discovery partially relies on an incomplete static analysis, some artifacts are reported as unresolved or orphans; also, several binding errors are reported which do not in fact exist. However, the number of such problems is rather low; thus we believe that we can still claim usefulness of our tool.

A final issue is how well our approach can be adapted to an interactive mode, since our refactoring tests have been carried out in batch mode. Model discovery and binding is currently a non-incremental process and takes up to a minute, depending on the number of bindings in the code. It is future work to investigate incremental discovery (as, for example, in the JDT compiler) as well as incremental artifact binding routines.

5 Related Work

We discuss adjacent existing work in three parts: Firstly, works which focus on a particular language binding or bindings; secondly, works which focus on multiple, but similar languages, and finally, other related work.

To our knowledge, our work is the first which ranges across six languages and three frameworks, focuses on a generic framework to be placed inside an IDE to treat all languages and language bindings in an equal way, and contains a systematic unit-test based evaluation of multi-language refactorings on seven open-source case studies.

Domain- or Language Specific Approaches Firstly, there are several works which implement support for individual language pairings which usually go deeper into individual language semantics whereas our approach is focused on a generic integration architecture.

In 2008, the workshop on refactoring tools has drawn two papers on cross-language refactoring. Chen and Johnson [1] present an approach for refactoring references to Java in XML code (examples given are Spring, Struts, and Hibernate). XPath expressions are used to locate references in XML code, and rename refactorings are considered. Kempf et al. [6] have discussed cross-language refactoring between Java and the Groovy programming language, also with a focus on renaming. A follow-up paper in 2009 [7] has shown these refactorings to be completely automatable, and the implementation is now part of the official Groovy Eclipse plugin. In both cases, the implementations presented are specific to the target languages (i.e. XML and Groovy); our own approach could be used to integrate these languages and binding implementations with others.

A similar work which deals with interactions between two particular languages is Tatlock et al. [21] (2008). They use the term deep refactoring for their approach to the refactoring of Java applications using a JPA-based framework. Both class and field renames between Java entities and JPA queries are considered. Their approach uses data flow analysis to also collect partial queries. Furthermore, they include a type checking algorithm for verifying inputs and outputs of queries, i.e. whether the correct Java types are used as parameters and returning elements in JPA. Thus, their approach goes beyond what we offer for JPA, but is in turn restricted to JPA and queries, since the grammar-based approach they use is not easily extensible to other, non-query languages.

Schink et al. [16] have presented, in 2011, an approach to refactor Hibernate applications which include entity definitions (in this case, as annotations) and queries. The refactorings analyzed are *Rename Method*, *Pull Up Method* and *Introduce Default Value*. An interesting aspect here is the discussion of the data present in the database, and of the impact of refactorings (such as pull up method) on such data. Thus, this approach goes beyond rename and even uses a dedicated database refactoring; again, it would be interesting to integrate these efforts (and investigate different refactorings for additional languages).

In 2012, Nguyen et al. [13] have presented the tool BabelRef which handles cross-language function calls and widget references in the web application languages PHP, JavaScript, and HTML with the goal of renaming elements. Their specific focus is on the partial nature of HTML page parts in PHP, where they use symbolic execution to create a single tree structure called D-model; by contrast, our own approach uses an artifact model with separate bindings.

Refactorings on Similar Languages Secondly, there is some work on refactoring multiple languages which share similar concepts, such as being object-oriented. Such approaches can take advantage of language similarities, which however makes them specific to this context.

The first two of these are by Strein et al.[19,20] (2006). Here, a generic framework based on a common meta-model is presented with the aim of renaming ele-

ments, in particular methods, across languages. They present the tool X-Develop, which implements these refactorings for the languages of the .NET framework (C#, J#, and Visual Basic). A key difference in this approach is the use of a common meta-model, which is beneficial if the target languages are similar, as in .NET — all languages share the same or very similar concepts such as types, methods, and properties. Thus, elements from all languages are represented in the same way on this level, and it is indeed feasible to write refactorings on this level. By contrast, we have investigated very different languages in which there are few common concepts; we therefore use per-language models and re-use existing individual refactoring implementations for each language.

Sobernig and Zdun [17] (2010) discuss multi-language refactoring as an evaluation technique for implementing multi-language method calls in a scenario in which one OO language is embedded into another (in this case, the Frag language into Java). The main goal of this work is a comparison between reflective and generative integration techniques, where the amount of effort required for implementing refactorings can be used for comparison. Besides renaming, they also consider the very interesting refactorings *replace embedded with host object* and *replace embedded with host method*, as well as *remove host method*. Such refactorings are naturally only possible in languages with similar functionality; not between a general-purpose and a domain-specific language as in our case.

Other related work We have investigated patterns of cross-language linking between Java and DSLs before [11] (namely, Spring, Hibernate, and Android), identifying how to describe and implement binding resolution between artifacts in these frameworks. In a follow-up paper [12], we have presented results from the application of such resolution on a single case study. Building on these results, the current paper provides a comprehensive description of an approach and implementation of a generic and systematic multi-language support framework, and includes a thorough empirical investigation on seven case studies.

In 2012, Pfeiffer and Wasowski [14] have executed a user study to show multi-language support mechanisms in general to aid software developers. This experiment with 22 participants has evaluated the tool TexMo, which includes support for links between Java, Hibernate and Wicket, and is based on the JTrac case study which we use as well. The main differences lie in the fact that artifact bindings in TexMo are manually established, and are based on a common, text-based model. In comparison, our tests show the feasibility of automation as well as the usefulness of cross-language renames from the unit testing perspective.

A more general discussion of program comprehension and maintenance of multi-language application can be found in Kontogiannis et al. in 2006 [8]. Open issues relate to gathering data, formalization and modeling of multi-language systems, extraction, discovery and storage of extracted information, and how to support exploration, queries, and knowledge management.

6 Conclusion

Multi-language software applications (MLSAs) are a common occurrence for which systematic tool support is lacking in today's IDEs. We believe that such support can make a significant difference for developers, and have thus investigated an approach and tool for multi-language software. Our implementation supports six languages (Java, Spring, HBM, HQL, Wicket/API and Wicket/HTML) across three frameworks (Spring, Hibernate, and Wicket).

Our approach treats languages and language bindings as first-level entities, and includes systematic support for multiple (in particular, more than two at a time) languages without language bias. We provide a generic refactoring algorithm which re-uses existing single-language refactorings and propagates changes across languages based on artifact bindings.

Using manual inspection for artifact and binding discovery as well as automated refactoring and unit testing, we have evaluated our tool on seven open source case studies with a total of 3768 bindings between artifacts in different languages. The automated refactorings succeeded in 95.96% of the 1709 transitive closures of artifacts which must be renamed together. The remaining cases were annotated with warnings such that the user is aware of potential problems.

Through our experiments, we have shown that the tool is fit for the purpose it was created, i.e. automatically and correctly finding multi-language relevant artifacts, discovering the bindings between them, and (rename) refactoring element across language borders.

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

References

1. Chen, N., Johnson, R.: Toward Refactoring in a Polyglot World. In: Proceedings of the 2nd Workshop on Refactoring Tools. pp. 1–4. ACM (2008)
2. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Proceedings of the 10th International Conference on Static Analysis. pp. 1–18. Springer (2003)
3. Fjeldberg, H.C.: Polyglot Programming. A Business Perspective. Master thesis, Norwegian University of Science and Technology (2008)
4. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Pearson Education (2012)
5. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
6. Kempf, M., Kleeb, R., Klenk, M., Sommerlad, P.: Cross language refactoring for Eclipse plug-ins. In: Proceedings of the 2nd Workshop on Refactoring Tools. pp. 1:1–1:4. ACM (2008)
7. Klenk, M., Kleeb, R., Kempf, M., Sommerlad, P.: Refactoring support for the Groovy-Eclipse plug-in. In: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. pp. 727–728. ACM (2008)
8. Kontogiannis, K., Linos, P., Wong, K.: Comprehension and Maintenance of Large-Scale Multi-Language Software Applications. In: Proceedings of the 22nd IEEE

International Conference on Software Maintenance. pp. 497–500. IEEE Computer Society (2006)
9. Linos, P.: PolyCare: A Tool for Re-engineering Multi-language Program Integrations. In: Proceeding of the 1st IEEE International Conference on Engineering of Complex Computer Systems. pp. 338–341. IEEE Computer Society Press (1995)
10. Livshits, B., Whaley, J., Lam, M.S.: Reflection Analysis for Java. In: Proceedings of the 3rd Asian Conference on Programming Languages and Systems. pp. 139–160. Springer (2005)
11. Mayer, P., Schroeder, A.: Patterns of Cross-Language Linking in Java Frameworks. In: Proceedings of the 21st IEEE International Conference on Program Comprehension. pp. 113–122 (2013)
12. Mayer, P., Schroeder, A.: Towards Automated Cross-Language Refactorings between Java and DSLs used by Java Frameworks. In: Proceedings of the 6th ACM Workshop on Refactoring Tools. pp. 1–4 (2013)
13. Nguyen, H.V., Nguyen, H.A., Nguyen, T.T., Nguyen, T.N.: BabelRef: detection and renaming tool for cross-language program entities in dynamic web applications. In: Proceedings of the 34th International Conference on Software Engineering. pp. 1391–1394. IEEE Press (2012)
14. Pfeiffer, R.H., Wasowski, A.: Cross-Language Support Mechanisms Significantly Aid Software Development. In: Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems. LNCS, vol. 7590, pp. 168–184. Springer (2012)
15. Pfeiffer, R.H., Wasowski, A.: TexMo: A Multi-language Development Environment. In: Proceedings of the 8th European Conference Modelling Foundations and Applications. LNCS, vol. 7349, pp. 178–193. Springer (2012)
16. Schink, H., Kuhlemann, M., Saake, G., Lämmel, R.: Hurdles in Multi-language Refactoring of Hibernate Applications. In: Proceedings of the 6th International Conference on Software and Data Technologies. pp. 129–134. SciTePress (2011)
17. Sobernig, S., Zdun, U.: Evaluating java runtime reflection for implementing cross-language method invocations. In: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java. pp. 139–147. ACM (2010)
18. Steimann, F., Thies, A.: From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In: Proceedings of the 23rd European Conference on Object-Oriented Programming. LNCS, vol. 5653, pp. 419–443. Springer (2009)
19. Strein, D., Kratz, H., Lowe, W.: Cross-Language Program Analysis and Refactoring. In: Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation. pp. 207–216. IEEE Computer Society (2006)
20. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An Extensible Meta-Model for Program Analysis. IEEE Transactions on Software Engineering 33(9), 592–607 (Sep 2007)
21. Tatlock, Z., Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: Deep typechecking and refactoring. In: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. pp. 37–52. ACM (2008)
22. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.: Use, disuse, and misuse of automated refactorings. In: Proceedings of the 34th International Conference on Software Engineering. pp. 233–243. IEEE Press (2012)

A Artifact Description

Authors of the artifact. Design and Core Implementation: Philip Mayer, Andreas Schroeder. Language Metamodels and Parsers: Thomas Neumeier

Summary. This aim of this artifact is demonstrating feasibility of an implementation for analysis and refactoring of multi-language software systems (MLSAs), and for collecting data from this process. As such, the routines for gathering artifacts, artifact bindings, and for executing refactorings are targeted at batch processing, using (lengthy) tables with CSV export functionality as output. Support is also available for graphically visualizing some aspects of the data, and for navigating to the source code positions of artifacts, bindings, and closures.

The implementation is realized as a set of Eclipse plug-ins. These plug-ins include EMF meta-models, model parsers, language binding implementations, and refactoring add-ons for six languages (Java, Spring, Hibernate/HBM, Hibernate/HQL, Wicket/API, and Wicket/HTML). Additional code (views, editors, actions) provides the user interfaces and glue code required to use the core routines as well as automated regression tests for all case studies.

The artifact package provides a virtual machine image designed to support repeatability of the experiments in the paper and thus regenerating the data we have presented. It also includes the code of the seven case studies we have used in our analysis.

Content. The artifact package consists of a VirtualBox VM image which hosts:

- an Eclipse installation with all required plug-ins and the source code of our tool implementation in the workspace
- a configured runtime Eclipse launch configuration with all seven case studies in the workspace to be used for testing
- detailed instructions for using the artifact to test some interesting language links as well as reproducing the data used in the paper

Getting the artifact. The artifact endorsed by the Artifact Evaluation Committee is available free of charge as supplementary material of this paper on SpringerLink. Additionally, the source code and instructions for installation are available on our website: `http://www.xllsrc.net/`.

Tested platforms. Being Java- and Eclipse-based, the artifact should work on all major platforms. The virtual machine image is known to work on any platform running Oracle VirtualBox (with around 4GB of main memory).

License. EPL-1.0 (`http://www.eclipse.org/legal/epl-v10.html`)

MD5 sum of the artifact. e4be341e2b4a02b9bd118a5488125ba5

Size of the artifact. 4.03 GB