Patterns of Cross-Language Linking in Java Frameworks

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

***Abstract*—The term *Cross-Language Linking* refers to the ability to specify, locate, navigate, and keep intact the connections between artifacts defined in different programming languages used for building one software application. Although understanding cross-language links and keeping them intact during development and maintenance activities is an important productivity issue, there has been little research on understanding the characteristics of such connections. We have thus built a theory from case studies, specifically, three theory-selected Java cross-language frameworks, each of which links artifacts written in the Java programming language to artifacts written in a declarative, framework-specific domain specific language. Our main contribution is to identify, from these experiences, common patterns of cross-language linking in the domain of Java frameworks with DSLs, which besides their informative nature can also be seen as requirements for designing and building a linking language and tooling infrastructure.**

***Index Terms*—Cross-language, artifact linking, semantic models, language design, patterns, case study**

I. Introduction

The use of multiple programming languages to build one software system is common for most applications today [1]. Usually, a multitude of languages is used in the development, deployment, and maintenance of software — this includes imperative, functional, and object-oriented general-purpose languages, database querying languages, UI specification languages, system configuration languages, and a multitude of even more domain-specific languages [2].

The languages in such *Multi-Language Software Applications* (MLSAs) [3] do not stand alone. Rather, they integrate and reference each others artifacts to implement the desired system behavior. This creates *cross-language semantic links* [4] between artifacts of different languages, and the system depends on the integrity of such links for correct operation.

Cross-language links are by definition out of scope for tools written for any one programming language. It has been argued that the lack of explicit support for cross-language links is detrimental to developer productivity and overall stability of the system [1], [4], [5], [6]. An experiment with students [7] has already shown that using a support tool for such *polyglot programming* is beneficial for developers.

Different approaches to handling cross-language links have been discussed in the literature for different purposes such as analysis, code understanding, and refactoring [4]. These approaches differ with regard to which frameworks are targeted and how the rules governing cross-language links are described. What is clear is that formulating such rules, and even more creating a formal language for expressing such rules, is a complex task — especially if the aim of the language is enabling not only analysis and program understanding, but refactoring as well.

We believe that this task requires a *thorough understanding of the structure of cross-language links*, which has not yet been studied in a rigorous way. In the paper at hand, we describe our efforts to contribute to this understanding. We first carve out a subdomain of this area, namely frameworks based on Java with their own respective domain specific languages. Then, we look at the properties cross-language links exhibit in real-world, industrial Java frameworks that make use of cross-language linking — and try to describe and generalize on these properties. We do not, in this work, address other cross-language links such as between two general-purpose languages or between two declarative, domain-specific languages.

Using the approach for building theory from case studies [8], we have investigated three theory-selected cross-language frameworks, each of which links artifacts written in the Java programming language to artifacts written in a declarative, framework-specific domain specific language. Our main contribution is to identify common patterns of cross-language linking in this context, which besides their informative nature can also be seen as requirements for designing a linking language and tooling infrastructure.

This paper is structured as follows: We first describe our research method in Sect. II. Afterwards, we discuss the steps we have followed in more detail (Sect. III). The patterns of cross-language linking are reported in Sect. IV. We discuss our findings in Sect. V, present related work in Sect. VI, and follow up with a conclusion in Sect. VII.

II. Research Method

Our research aim is understanding how links between artifacts written in different programming languages are formed, what their characteristics are, and how they can be described. To gather such knowledge, we have chosen to use the research method for building theory from case studies, as outlined by Eisenhardt [8].

As we have pointed out in the introduction, the term *cross-language link* is very general and can basically be applied to link artifacts between *any* two languages; restricting to formal languages obviously does not help. Although our overarching

978-1-4673-3092-3/13/$31.00 © 2013 IEEE

aim is to get an understanding of all cross-language linking efforts, we acknowledge that we cannot, as of yet, describe the characteristics and categories of such links in *general*.

Instead, we start smaller: Following the method of Eisenhardt, we have attempted to identify a population or sub-area to focus our efforts on. We have decided to define our population as consisting of software frameworks which a) are written in the Java programming language; b) offer their services to developers in the Java programming language; and c) include a domain-specific, declarative language in which parts of the system are written and whose artifacts affect, or are affected by, Java artifacts. The research quesion we finally chose to address for this work is:

What are the characteristics of cross-language links between Java and domain-specific DSLs in Java frameworks? How can we describe and categorize their properties?

The reason for selecting this subset is threefold. First, Java is a well-known programming language with both academic and industrial relevance. Second, there are many open-source frameworks and applications in the Java world which are amenable to analysis. Thirdly, there exists good tool support in the form of IDEs for both Java and for many of the Java frameworks which can be easily extended.

Having selected our population, the question of sampling arises. As Eisenhardt points out, theory building from case studies requires focusing the effort on useful cases for the theory — thus, instead of random sampling, theoretical sampling is used. Surveying the area of frameworks with the above features, we have identified the three general areas with different purposes in which DSLs are common. These areas are a) *UI specification*, b) *system configuration*, and c) *data querying*. There are several DSL-enabled Java frameworks in each area, rendering those areas representative for the type of cross-language interactions we intend to look at.

We have then selected one *case* from each area, i.e. one framework to investigate further and to ground our theory in. In addition to their being part of our population, the frameworks were selected using the following rationale: a) it should be a well-known framework which is in industrial use; b) there should be a reference documentation available online, and c) there should be open-source applications available for testing. The following frameworks have been chosen:

- *Android UI framework*. Widely used on mobile devices, the Android UI framework is surely both well-known and in industrial use. The entire documentation is available online, and there are several open-source applications available which use the UI framework.
- *Spring Inversion-Of-Control*. The Spring IOC container, used for identifying beans and configuring them, is part of the Spring framework, a well-known and widely used Java enterprise framework. There is comprehensive documentation online and there are several large-scale open-source applications available for testing.
- *Hibernate Query Language*. The Hibernate Query Language (HQL), part of the Hibernate Object-Relational Mapper, is used to query objects from a database by referencing Java entities and their properties. Again, Hibernate is well-known, used in industry, documented online, and there are open-source applications available which can serve as a test-bed for our theory.

Having selected our cases, we then proceeded to the main steps of theory building. As discussed by Eisenhardt, these steps, which include data collection, data analysis, shaping hypotheses, and enfolding literature, are partially overlapping and are iterated until it is possible to reach closure.

During our work, we have realized that a good way of actually writing down our theory is using a pattern language, i.e. formulating patterns of cross-language linking which describe the properties of cross-language links. To reach a grounded, saturated list of patterns, we have performed the following steps in iteration:

- Reading the framework documentation to compile a textual description of which artifacts are linked by the framework in which way. We call these descriptions *rules*. In a way, these rules are a handbook for how to establish and check links.
- Implementing the descriptions in Java, running them on open-source demo applications, and checking the results manually against the code to ensure their correctness.
- Harmonizing these descriptions across the cases, i.e. trying to find common themes.
- Annotating (what Eisenhardt calls *coding*) the textual descriptions with what were at first structural findings and which were to become the *patterns* later on, to make sure each pattern is actually grounded in all cases.
- Fleshing out the patterns with descriptions and cross-references to enable them to stand alone, back-referencing to the textual linking descriptions.

To our knowledge, this is the first attempt to describe the characteristics of cross-language links in Java frameworks as described above.

III. Frameworks, Rules, and Applications

Our attempts to discover which cross-language links exist in the three frameworks and how to describe them can be split into three steps. The first of these is establishing a way of talking about language artifacts by using semantic models (discussed in section III-A). Based on this, we can write down, in plain text, where, how, and what to link in each case in the form of rules (section III-B). Finally, these rules can be implemented and tested against actual code written using the frameworks, which is discussed in section III-C.

A. Language Metamodels

When setting out to describing the cross-language links, it quickly becomes apparent that we need some way of precisely specifying which artifacts, and links between artifacts, are being discussed. A good way of doing this is by using semantic models [2], i.e. meta-models of programming languages which capture the domain of the language on a higher and more

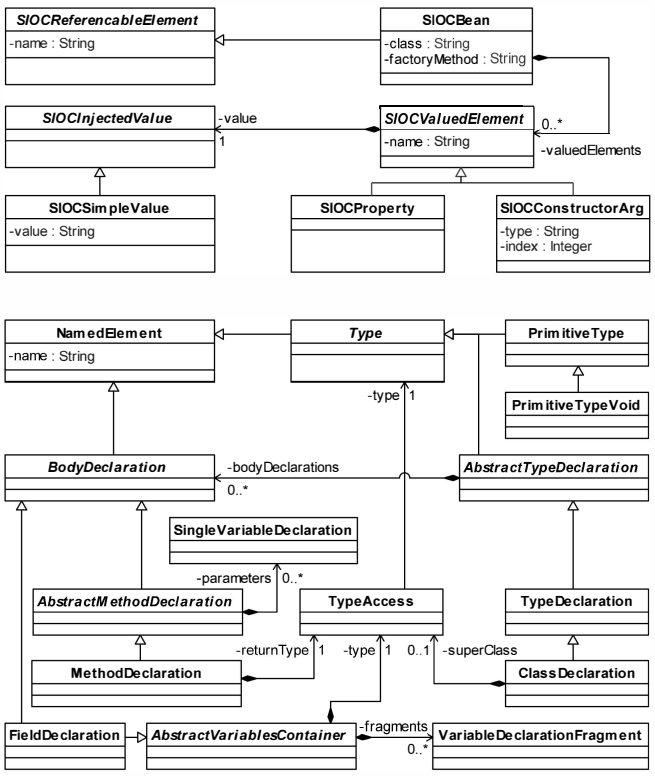

Fig. 1. Spring IOC (top) and Java model fragments (bottom)

conceptual level than an abstract syntax tree, and usually using a graph-based, fully resolved structure.

Unfortunately, such models are available for very few languages. While we have been able to find one for Java (MoDisco, [9]), none are available for the domain-specific languages of our three example cases. We have thus created semantic models for each of them using the eCore meta-modeling facility. These models target the languages themselves, i.e. are without a specific cross-language focus, and represent a fully resolved graph of the source artifacts.

A small fragment of the graphical representation of both our Spring model and the MoDisco Java model can be seen in Fig. 1. The Spring IOC model fragment depicted specifies the existing attributes and relations between Spring beans, their properties and injected values, and is a subset of the full Spring IOC model. Similarly, the MoDisco model fragment describes how (in MoDisco) class declarations contain methods and fields that may be linked to Spring properties.

B. Framework Cross-Language Rules

With these models in place, we have set out to create a textual description of the cross-language links for each of the frameworks. Over the course of several months, these rules went through different iterations. During this process, all relevant sentences have been annotated (coded) with patterns again and again, a process which has finally yielded the patterns discussed in the next section.

The rules cover several pages for each case: The largest being Android (9 pages), followed by Spring (8 pages), and finally HQL (3 pages). We thus cannot discuss each in detail here. Rather, we will outline the basic idea for each of the three cases. An extract from an actual rule is shown in Fig. 2, which describes how bean properties are linked to Java method declarations or field declarations, and makes use of the Spring IOC and Java model fragments shown in Fig. 1.

Spring IOC: The Spring IOC container allows defining, mostly in XML but also through Java annotations, so-called *beans*, which are (usually named) instances of a Java type that is managed and configured by the Spring framework. A bean instantiation may take three forms: A *constructor based bean* is instantiated directly using a Java constructor; a *factory-class based bean* uses a class with a (static) factory method to retrieve the instance, and a *factory-bean based class* uses a (non-static) factory method in another bean. A bean specification may include *constructor arguments*, i.e. values which are given to the constructor or factory method as parameters. Furthermore, each bean specification may include *properties*, which contain values to be set on the instantiated bean using setter methods or directly on fields. Values can be either basic or again beans or references to beans.

The main cross-language links in this case are a) linking a bean specification to a Java type, and b) linking properties to Java setters or fields and linking constructor arguments to constructor and factory method parameters.

Hibernate Query Language: The Hibernate Query Language is an SQL-like language for querying a (relational) database, but using Java entities, entity property names, and navigation style. Thus, the FROM clause in HQL references Java entities, giving them aliases in the process (for example, `FROM User u`). As in the example, an entity can be referenced by simple name (*named entity reference*); another possibility is inferring the type by navigating properties (for example, `IN(blog.user) u`, where `blog` is an entity alias and `user` a property access) (*inferred entity reference*). The same kind of navigation as in this example can also be used in other parts of the query, for example when selecting elements using WHERE, and can involve multiple levels of navigation; we call such expressions *path expressions*; all except the first one being *attribute accesses*.

The two main cross-language links in this case are a) linking named or inferred entity references to Java entities (i.e. properly annotated Java types), and b) linking attribute accesses to Java fields.

Android UI: The Android UI framework allows specifying the user interface of Android apps using an XML dialect. The interface is separated into *layouts*, each within its own XML file; each layout file contains *widgets* which may have (String) identifiers attached. Layouts are loaded explicitly, usually with one of two methods (`setContentView` and `inflate`); within a layout, a widget can be located using the `findViewById` method. The parameters of these methods are constants generated by an SDK tool whose names correspond to the layout file names and widget IDs, respectively. A widget returned by `findViewById` can be directly cast to the corresponding (known) Java type for the widget type.

Linking Bean Properties

This section describes how to evaluate and link SIOCProperty artifacts to artifacts in Java. This depends on the parent SIOCBean **b** of p having already been linked to a TypeDeclaration **td** before.

For all SIOCProperty instances **p** (from b.valuedElements), we try to find either a matching MethodDeclaration or a (FieldDeclaration-based) VariableDeclarationFragment inside td. This includes methods and fields defined in possible super classes: Using "Retrieving Inherited Body Declarations" (with td), we arrive at a list **l** of all BodyDeclarations visible in td.

Option 1: We try to find a MethodDeclaration **md** instance in the list l such that:

- md.returnType.type is instance of PrimitiveTypeVoid
- count(md.parameters) is 1
- If p.value (SIOClnjectedValue) **val** set: the type of val matches the type of the parameter md. parameters[0].type.type **t**, as discussed in "Type Checking Injected Elements" (with val and t).
- md.name is equal to the concatenation of "set" + the first-letter-uppercased version of p.name.

Option 2: If we have not found such an md, we try to find a FieldDeclaration **fd** in l such that:

- If p.value (SIOClnjectedValue) **val** is set: the type of val matches the type fd.type.type **t**, as discussed in "Type Checking Injected Elements" (with val and t)
- Find a fragment **frag** in fd.fragments whose frag.name is equal to p.name.

If such an md is found, we link p and md ("Setter Method"). If such a frag is found, we link p and frag ("Bean Property"). Else, we report an error for p ("No Method or Field found for Property").

Fig. 2. Spring Rule: Linking Beans

Again, we have two main cross-language links in this case: a) layout references are linked to layout files; and b) widget references are linked to widget declarations — the latter within the correct layout and with the correct type.

C. Implementing and Testing Adapters and Rules

Textually writing down the cross-language linking rules for the cases outlined in the previous subsection proved very useful since the freedom in using a non-formal language prevented us from following any one language paradigm too strictly. However, such a description is not executable. Thus, to validate our ideas, we have additionally implemented the language metamodels and support routines as well as the rules themselves in Java and applied them to real-life, open source applications implemented in each of the three frameworks discussed above.

We have chosen three applications for this purpose. The first is *Apache OpenMeetings*[1], an open-source web conferencing system which uses the Spring IOC container; the second is *JTrac*[2], an issue tracking system which uses Hibernate and HQL queries for database; and finally *Wordpress for Android*[3], an Android interface for the blogging software Wordpress. The size of both applications in lines of code (LOC) and number of artifacts in our semantic models (Artifacts) is shown in Tbl. I.

The implementation consists of two parts; firstly, the language support; second, the linking rules themselves.

[1]Version 2.0.0, http://openmeetings.apache.org/
[2]Version 2.1.0, http://www.jtrac.info/
[3]Version 2.3.3, http://android.wordpress.org/

TABLE I
MLSAS: LINES OF CODE

App. (Framework)	LOC/Artifacts (Java)	LOC/Artifacts (DSL)
OpenMeetings (Spring)	55698 / 238391	405 / 377
JTrac (HQL)	17737 / 82317	27 / 375
Wordpress for Android	20609 / 97007	3633 / 1173

TABLE II
MLSAS: CROSS-LANGUAGE LINKS DISCOVERED

Framework	Top-level links (a)	Second-level links (b)
Spring	133 (beans)	79+9 (propert. / constr.)
HQL	30+3 (named / inferred ent.)	79 (attributes)
Android	36 (layouts)	172 (widgets)

Regarding the language support, we use *language adapters* written in Java to implement a) creating an instance of a metamodel from source, and b) being able to *navigate* back to the source given an artifact. In case of Java, the first is readily provided by MoDisco, while the second is greatly helped by its management of text source positions. In the case of Spring and AndroidUI, we make use of the XML editor of the Web Tools Platform (WTP) of Eclipse to implement our own adapter. HQL, finally, uses the HQL AST parser of the Hibernate Eclipse plugin. We have manually checked that all (main, see previous section) artifacts reported in the model instances are actually there in source (and only those); and that back-navigation was possible for all artifacts.

Regarding the rules themselves, we have attempted to translate the textual descriptions we created as directly to Java as possible. The rules are implemented in Eclipse plugins and make use of the generated EMF language models and language

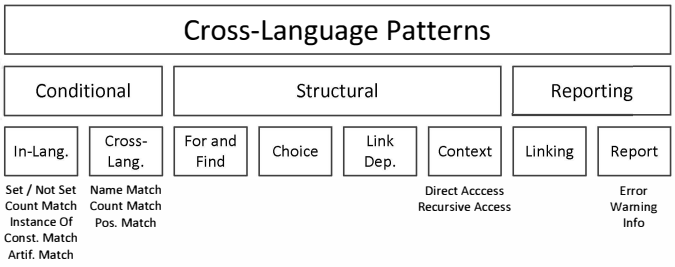

Fig. 3. Pattern Hierarchy

adapters discussed above. We do not use any domain-specific languages in this implementation since this is, in our mind, a next step after this work.

We have run the linking implementation on the applications listed above, which has resulted in links and reports. We have manually checked all generated links and link reports against the source code to ensure that the Java implementation was correct, and thus, by extension, our textual descriptions. Tbl. II shows a count of the cross-language links discovered, reduced to the main links we have identified in the previous section.

IV. Patterns

The main results of this work is our theory of the structure of cross-language links, presented in the form of patterns in this section. As discussed in the methods section, these patterns have been extracted in an iterative process from the data, i.e. the rules written for our three cases, and thus reflect how these rules are structured. We believe these patterns to be of two main uses.

- *For Program Understanding*. First, we came to realize that the domain of cross-language linking, as it exists today in Java frameworks with domain specific languages, is quite complex. Thus, these patterns aid in understanding the intricacies of linking artifacts in MLSAs.
- *As Language Requirements*. Second, it would be beneficial in the future to be able to specify cross-language linking rules not in plain text or in a generic language such as Java, but in a domain-specific language making them easier to write and read while still keeping them machine-readable. The patterns presented here serve as requirements for such a language.

We have identified eight distinct patterns organized into three categories. All patterns occur in all cases in different quantities. An overview of all patterns is shown in Fig. 3, while a numerical listing of the patterns as they occur in our three case applications is shown in Tbl. III.

In this section, we list all identified patterns. For each pattern, the name is given in the title. We add a description, examples, and other patterns to consider. A discussion is found in the subsequent section.

Structural Patterns

We begin with the *structural patterns*, since they form the framework for all other patterns. All four of these patterns can be combined in various ways and interleave each other.

A. For and Find

Name & Description: For and Find is actually two patterns, which refer to the two steps involved in searching for artifacts. We usually begin with iterating *all* elements of a certain type (the *for* part), looking for *one* other artifact which matches (the *find* part). This pair shares its semantics, on a more abstract level, with *for all* and *exists* from first order logic. This is, not surprisingly, a common theme which can be found at multiple locations in all three of our cases.

However, the cases show that a simple for-then-find does not suffice in practice. Rather, depending on the language structure, we see different combinations of *for* and *find*.

Still, each linking process starts with a *for* for the element to be linked and will end with a *find* for the target element, even if additional *for* steps are involved to make sure the target element is the right one.

Examples: Spring necessitates that we find the proper factory method for a factory-based bean, given a number of constructor arguments. Here, *for* all beans, we *find* one method where we must check *for* all Spring constructor arguments that we *find* one formal parameter in the corresponding method (*for-find-for-find*). All of this is a prerequisite to just linking a bean to a class.

In HQL, linking entities requires finding a class declaration with a certain name and a certain annotation. Since a class may have many annotations, this translates to a *for-find-find*.

The simplest example can be found in the Android framework, where we must go through all MethodInvocations (*for*) looking for an interesting method (say, *setContentView*). Once we have found that, we can directly *find* one matching LayoutFile (*for-find*).

See also: This pattern is the core structural pattern which encloses all others. One should note that all of the other patterns may occur after any element in a chain of *for* and *find*, not only after the last one.

B. Choice

Name & Description: The *Choice* pattern name refers to the fact that for many links, there are multiple linking options in cross-language linking. It could also be named *Switch* or *Disjunction*.

Cross-language frameworks allow for a large amount of linking options: For one source artifact, depending on conditions both on the *source* and *target side*, there might be different artifacts or artifact properties to look for in the target, and different artifacts or artifact properties to compare with one another. Choices of how to link can occur at any point in the *for-find* hierarchy — after the initial *for*, but even after the last *find*. This might create a quite intricate tree of options, some leading to successful links, some to problems, and some to simply ignoring an element.

The *find* part of *for-find* and the *choice* pattern serve different purposes. In the former, we want to select one element from a uniform list based on some conditions, stopping as soon as we find one. In *choice*, we generally select a linking path (not just one artifact) based on conditions of an artifact.

TABLE III
PATTERN OCCURRENCES

Pattern Name	Category	Spring	HQL	Android
In-Language	Comparative	73	16	72
Cross-Language	Comparative	11	2	2
For and Find	Structural	20	6	6
Choice	Structural	40	15	25
Dependency	Structural	4	2	1
Context	Structural	82	38	67
Establishing Links	Reporting	18	4	5
Reporting Problems	Reporting	16	5	9

Examples: In Spring, we begin with three options depending on the existing properties of the bean — which leads to different checking options in the case of constructor-based, static-factory-based, and dynamic-factory-based beans. This is an example of a source option (i.e. on the source side) based on artifact properties.

An interesting case of a target option can be found at another place in Spring, when we try to match bean properties. In this case, there are no source options, but a bean property can be either matched to a Java setter method or a Java field. However, one of those must be available.

In HQL, we have path expressions in which a Java field is referenced, possibly via many other fields (via chaining). Such expressions need to be resolved recursively, and at each point, we may have to deal with different artifacts (in particular, entity references and field accesses).

In Android we start out with generic method invocation artifacts and narrow our artifacts down based on method name and defining class. What exactly is linked on the source side also depends on the method expression — i.e. whether the method sets a layout for the entire class (*setContentView*) or returns an element which is assigned to a variable (*inflate*).

See also: A choice is always coupled with a condition, i.e. one of the *conditional* patterns shown in Fig. 3. Within our cases, all choices were based on in-language conditions.

C. Link Dependency

Name & Description: The name *Link Dependency* refers to relationships between *links*. The hierarchical nature of the languages we talk about usually means that some artifacts can – or may – only be linked if other artifacts, usually their parents, have already been linked before. In other words, linking (or checking) some artifacts requires that a link between some other artifacts has been established before.

A *link dependency* only works one-way: Some (child) artifacts can only be linked if some other (parent) artifacts have already been linked, but a link failure on the children does not affect the parent link.

It should be noted that naturally, many artifact links are fundamentally dependent on other artifacts, be it their parents or their children: Without them fulfilling some conditions, no link is established. This is however not a link dependency, but a combination of the context and conditional patterns.

Link dependencies occur in all cases. Their number is usually small but important nonetheless.

Examples: In Spring, we can only match bean properties if the bean has been matched to a class before, since we would otherwise have no reference to look for methods or fields. Furthermore, beans may be nested or referenced, which also requires previous links.

HQL shows an interesting dependency pattern: In a query, an entity may either be referenced directly (by name) or indirectly (by attribute access to another entity). Thus, a link to a dependent entity must have been established before linking an indirectly referenced entity, which is again a prerequisite for other attribute accesses.

The Android case is rather simple: Linking widget references to widgets is only possible if a layout reference has been linked to a layout before, so we only have one dependency.

See Also: Successfully linking an artifact may also depend on conditions placed on other artifacts — not links. This is a combination of one of the two *conditional* patterns placed on a *context* artifact. See below for these three patterns.

D. Context

Name & Description: The *context* pattern — one of the most occurring patterns — describes that linking artifacts usually requires conditions not only on those artifacts, but also on their context, i.e. referenced artifacts which may include children or parents in the model.

From a high-level perspective, linking elements in two languages is as simple as "for all bean specifications, find a class declaration such that". Mostly, however, selecting the artifacts which are eligible for linking requires more context in the sense of where an element is located in the model, i.e. in the model hierarchy, as indicated by references between artifacts.

Given the nature of language models, the hierarchy in which elements are placed is usually a containment tree with side references turning it into a graph. Thus, there is always a notion of a container, and of children, which has turned out to be helpful to write down context conditions in many cases.

We can distinguish context access by simple references (direct access) and by recursive references (recursive access). A directly accessed artifact is a previously known, fixed number of steps away from its source, while a recursive access involves an arbitrarily deep navigation to a target element.

It is important to note that cleverly picking artifacts can greatly reduce these *navigation steps*, but it will rarely be possible to do without.

Examples: A classical example occurs in Spring: To find a certain class, we must check its fully qualified name — which means recursively scanning the packages of the type, although those are not linked and are not important for any other reason.

In HQL we need to make sure that a referenced class is actually annotated with the *Entity* annotation type, which involves navigating the type parameters of the class.

Finally, when linking Android artifacts, we have to navigate from method invocation to the invoked method declaration to the type it is declared in to find out how to link an artifact — but neither the declaration nor the type are linked.

See also: *Context* accesses occur inside of *conditional patterns*, whether they are placed inside of a *choice* pattern or used to directly used to decide whether an artifact qualifies for a link.

Conditional Patterns

Conditional patterns occur inside the structural patterns and deal with conditions: An element, or an element property, is compared either within one language (with a constant, for example) or across languages (with an artifact property in another language) to decide how to continue.

E. Intra-Language Conditions

Name & Description: Most of the conditions we place on artifacts and artifact properties in cross-language linking are not, themselves, cross-language. Rather, to find the correct artifacts we want to link, both in the source and target language, we require *intra-language conditions*, i.e. only affecting artifacts in one language, which select the correct artifacts for us.

Intra-language conditions do just that: They compare one artifact, or (more often) a property value of an artifact, to a value. We can distinguish the following forms of intra-language checking:

- *Type check* By far the most common application is type checking — i.e. whether a certain artifact has a certain type. This is mostly due to the object-oriented structure of the semantic models used, and the fact that choices are often made based on the type of an element.
- *Constants* Comparisons with constants are the second most common type of comparisons. The constants compared are either taken from the language (for example, Java enum values) or simple values such as strings or numbers.
- *Set/Not Set* Sometimes it is important to know whether some property is set or not set to continue.
- *Counting* It may also be important to select an artifact based on the count of elements in one of its (collection) properties.
- *Artifact Match* Besides comparing values of properties of artifacts, we can also directly compare artifacts by object identity.

It should be noted that intra-language conditions can be used both on the source and on the target side, and both on artifacts to be linked and on contextual elements.

Examples: Examples for intra-language conditions abound. Some examples from Spring include the question of whether the class, factoryMethod and factoryBean attributes of a bean are set or that, in a method declaration, the return type is an instance of PrimitiveTypeVoid, an enum-constant-like singleton class from the Java model.

HQL rules include the condition that the name of an annotation type must equal *Entity*. Another example is deciding, based on the type of an element in an attribute reference expression, how to resolve it.

Finally, in Android, we have the condition that the count of a method's parameters must be exactly 1, and that the element at index 0 must be an instance of SingleVariableAccess.

See Also: The high number of intra-language comparisons matches those of applications of the *context* pattern. The difference in numbers between in-language and *cross-language conditions* is also quite high; both of which shows that a lot of contextual conditions need to hold true for a link to be successfully established.

F. Cross-Language Conditions

Name & Description: Cross-language conditions are those conditions we require all of the other conditions and patterns to get to: Finally comparing artifacts from two different languages with one another. In the end, this is surprisingly simple and involves only three types of checking: Matching names, matching counts, or matching positions of elements.

The most common type are name comparisons — either of simple names directly retrieved from two artifacts, names which have been built by contextual navigation, such as fully qualified Java names, or names with are concatenations of several strings, for example for forming a Java bean setter name out of the constant string "set" and the first-letter-uppercased version of another name.

There are two additional types of comparisons: The first is matching counts of elements — this occurs, for example, when matching method parameters. The second type is matching positions, which also occurs when matching parameters, where two elements must be at the same index in a collection.

With the small number of cross-language comparisons, it is easy to forget that context is relevant as well and is, in a way, also a cross-language requirement. The cross-language conditions are just the tip of the iceberg.

Examples: In Spring, we compare fully qualified names of classes to the class property of a bean, or the name of a property to either a field name or the properly constructed name of a setter method. When checking constructor arguments, indices (positions) matter if the index property is set on an argument.

HQL queries directly use the simple names of Java classes for reference. Similarly, names of attributes are used to reference Java fields.

In the Android case, both layouts and widget ids are strings, which can be used to compare against file names (layout id concatenated with ".xml") and the IDs extracted from widgets defined in XML elements.

See Also: All of the *structural patterns* as well as the *intra-language conditions* are preconditions for this pattern. Usually, the outcome of a *cross-language pattern* determines the final verdict on whether or not a link is established.

Reporting Patterns

One interesting insight about cross-language linking is that establishing links and reporting problems are hard to press into a single structural pattern. As indicated by the last two pattern categories, there are many different facets for linking, different options and navigation possibilities until we reach

the target element we are interested in, and even the source element might need to be identified properly before we start.

Establishing of links and reporting problems is thus less of a conjugated action in cross-language linking than would initially seem. We have therefore purposefully separated linking and reporting as distinct patterns.

G. Linking

Name & Description: Linking means stating that artifacts from two different languages were matched, i.e. a connection or link is created between them.

When discussing actual cross-language links, we need to consider use cases — i.e., how will the links be used by the system and/or by the user? In particular, we have to address two aspects when thinking of links:

- Firstly, one artifact may need to be linked to multiple other artifacts. It is important to distinguish between these artifacts, especially when they have the same type, for example by assigning an identifier. This is necessary not only because they may serve different purposes in the framework but also for the end user (who may wish to navigate or view the links and understand why they were established).
- Secondly, one artifact may not always be linked to the same target type. Depending on linking options, the target may be one of several artifacts from different locations in the target model. Also, during the investigation of a link, different source artifact types may also play a role, so neither side is fixed a priori.

Another important thing to consider with multiple links is their dependency relationship. If a link is based on another link, this fact should be registered for reporting. If a link cannot be established (see next pattern) but has a successful parent artifact link, this fact should be noted as well as it may help with finding the fault.

Examples: In the case of Spring, we link, for example, a bean specification to a Java type. If the bean specification is factory-based, we also link the factory method, and possibly even more methods based on additional options. It becomes important here to distinguish these links since they serve different purposes.

With regard to links, HQL is the most classic of cases: An entity reference — whether direct or indirect — is linked to a Java class, while property references are linked to Java fields.

In the Android framework, we have the interesting situation that for a layout link, the source artifact is not always of the same type. In one case we link a class (if it is a subclass of Activity or View); in another we link a variable (in case the layout was inflated). As target, however, we can always use either a layout file or a widget.

See also: In many cases, but not always, the *linking* pattern occurs together with the *report* pattern and at the end of a long branching tree of *structural* linking patterns and *conditions*.

H. Report

Name & Description: Reporting means any kind of information given to developers in case one or more links could not be established. The basic form of such a report is an *error*, indicating that a partner could not be found for an artifact. This is the classic opponent to the link pattern. However, during our work it has turned out that in many cases, *warnings* may be important to be reported as well, and in some cases, even *informational* reports.

The report pattern may initially sound suspiciously technical — after all, some reporting must always be done to keep the user informed. However, we believe that the report pattern is indeed a pattern in its own right for the following reasons:

First, it is important to note that due to the many options and requirements during linking, many alternatives may be investigated, leading do various outcomes for the individual artifacts considered: We could create a link, we could fail, or we could simply ignore an element. Here, the report pattern is necessary for distinguishing between failing and ignoring.

Second, in case a link is established based on a number of conditions, it is not always the case that it is an error, warning, or informational message if those same conditions fail. This decoupling from link and report is our main reason for distinguishing between them: Implementors and designers of a possible linking language should be aware of this issue.

Still, the report pattern does have some technical aspects, since it is important to help developers quickly understand why a link failed, and where to look for a remedy. For this reason, we suggest that problem reports not only include the offending artifact, but a *context* and a *reason* as well.

Examples: In Spring, a basic example for an error is a bean for which the corresponding class could not be found. An example for a warning report is a property which references another bean which could not be resolved (an instance of the *dependency* pattern). In this case, a warning notifies the developer that the property could not be checked further due to an error in another link.

In HQL, we use error reports if, for a named entity, the corresponding class cannot be found. Another example is a missing attribute in an otherwise defined class. HQL links are also interdependent, and recursively so. Thus, if we can link some of the elements but get stuck at some point, the context will make it easier for a developer to understand where investigate.

With regard to Android, failure to find a widget identifier referenced from Java in the proper layout file must yield an error report. It is important to attach this report to the previously linked layout, since it is only in this context that the report is understandable for developers. An example for an informational report in Android is a widget which is referenced from Java but not assigned to a variable: In this case, the widget cannot be not checked for the correct type.

See also: The *report* pattern often occurs in combination with the *link* pattern and is based on some final, often cross-language, condition which has either failed or succeeded. However, especially in the case of warning and informational reports, the report pattern may occur during *context* evaluation and thus at a different place than the linking pattern.

V. Discussion

In this section, we discuss the results of our study, in particular in light of the research question outlined in Sect. II.

A. Main Findings

A first main observation of our study has been the complexity of cross-language bindings. While cross-language rules often appear to be simple from first glance at the framework description, actually writing down the rules reveals an often intricate, rich structure of cross-language links.

This becomes apparent when we look at the patterns identified, and at the number of occurrences for each pattern. Five patterns deal with issues that are not directly cross-language: This includes all of the structural patterns as well as the in-language conditional pattern. When looking at the numbers, the highest counts can be found for the context pattern, the in-language comparative pattern, and the choice pattern — in all three cases. All three of these patterns do not compare across languages, do not link artifacts, and do not report on errors — they simply establish which structure, on both sides of the language gulf, must be available for linking to proceed.

Among the patterns with the lowest number of occurrences is the pattern of cross-language conditions. The actual relation between the found artifacts is a last step after establishing the structure, and is very often (but not always) established through string equality.

This finding has important implications for the future goal of creating a formal language specifically geared at modeling cross-language links. What such a language must support, specifically, is dealing with *structure*: Allowing context navigation, placing constraints on non-linked artifacts which may be recursively removed from the linked artifacts, and allowing for nested options at each structural level.

A second finding is that while some of the patterns identified seem quite obvious, many are not. Surely, the most obvious patterns in our catalog are the for and find pattern and the cross-language conditional pattern — corresponding to searching for artifacts and directly comparing them. By contrast, what may be easily overlooked are patterns such as choice and dependency: The amount of options in cross-language linking is quite high, and the relationship between different types of links is important both for better understanding links and for structuring them for output.

It is also worth pointing to the fact that we have identified linking and reporting as patterns in their own right which are a) distinct from one another and b) distinct from the other patterns. We have attempted both to integrate the patterns into one and to integrate the patterns with others; specifically, the for and find pattern. However, the structure of the cross-language links in our cases did not allow this: It is simply not the case that problem reports always occur at the same place as links, and they also do not always occur at a successful or unsuccessful find.

Both of this implies that we should watch out, in the future, for these less-obvious patterns and pattern relationships when discussing cross-language links.

A third finding relates to the relationship between link descriptions and languages. Our whole work in this study is based on the assumption that we deal with language artifacts "as is": We use semantic models for each language which directly reflect the (semantic, not syntactic) language structure in all its complexity. In other words, the semantic models have not been created specifically for cross-language linking purposes, which would have allowed reducing complexity of links by moving it into the source extraction process (performed by language adapters).

We stress this point because only using language constructs and artifacts as they appear in the original language drags all aspects of cross-language linking out into the open, which we feel is important in terms of understanding them. We must deal with the fact that languages have a certain structure not under our control, and so do the semantic models which reflect these structures.

To sum up, the explicit formulation and characterization of cross-language links for our three cases has been most informative for us, and we hope that our patterns help others in understanding the nature of cross-language links as well.

B. Threats to Validity

The scope of our theory, i.e. our patterns, is fundamentally restricted by first the chosen population and second the three frameworks we have used as our cases.

As usual in building theory from case studies, our cases have been selected based on theoretical grounds, and not by random sampling. We believe that our three frameworks are quite representative of their areas of application and are well-known and used in the community, ensuring applicability. However, it may well be possible that frameworks which support similar features follow a completely different structure for linking, which could be tested by successfully or unsuccessfully discovering the patterns in yet more cases.

All patterns identified can be found in all three frameworks, and in equal numbers relative to one another — despite their rather different application areas. This indicates that the patterns are indeed applicable across different framework goals. In addition, we take this fact as a good sign that the patterns identified are not flukes. In this context, it should be pointed out the count of patterns within each case will vary depending on how the rules are written down, and should not be considered as absolute numbers, but as relative within each case.

Whether the theory is applicable to other frameworks within the population, and especially outside our population, has to be validated. This requires further testing, which is possible by assuming that the patterns are predictions about how other cross-language links are structured and testing these predictions against the reality of other cross-language frameworks.

Our theory can be further tested by attempting to create a dedicated linking language for describing the links (instead of using Java) and evaluating how helpful the patterns have been in this regard (which we plan to do).

VI. Related Work

We can identify three areas of related work for our pattern description. The area most directly related is other approaches to characterizing and classifying aspects of cross-language linking. A second area concerns the approach to linking, i.e., *what exactly* is being linked. Finally, there are works which deal with tools for cross-language linking.

Regarding the first area, we are not aware of any work which directly attempts to characterize cross-language link patterns. However, there is other work on structuring the area of cross-language linking on a higher abstraction level than what we have looked at. The taxonomy of multi-language development environments in [10] forms a description of the entire cross-language linking problem area. We specialize in one of their addressed areas, namely the discussion of *relation types*. A similar approach to categorizing the choices to be made when analyzing multi-language applications is shown in [11]; again, our work is a deep dive into one of their categories, namely requisites on how to perform (automated) identification of references. Linguistic architectures [12] are a modeling approach to aid understanding of the *structural relationship* of languages taking part in certain MLSAs. In this context, our work specifically addresses how complex intermodel mappings between artifacts look like and how they might be established. It will be interesting in the future to discuss the nature of cross-language links (i.e. whether they are correspondences, realizations, simple references, etc.), and how our patterns can be used in this context.

In the second related area, cross-language linking may use different entity types as the start- and endpoints of links. As discussed above, we have used (semantic) meta-models to be able to talk about the languages involved: We are relating artifact from different per-language models to one another. The same approach of one meta-model per language has been used in [1]. However, there are other options: Regular expressions and heuristics [5] have been used for directly linking source code (i.e., *text*). There is also the option of using just *one* model for all languages involved, for example by using text fragments as model elements [10]; or using higher-level artifacts such as methods, fields, etc., for example by using tree grammars [13], [14]. The individual benefits and shortcomings of these approaches have not yet been compared in any rigorous way.

Finally, there are several works which deal with *tools* for cross-language analysis and refactoring. We refer the reader to our discussion of these in section III of [4].

VII. Conclusion

This paper has reported our work on understanding cross-language links and link types that occur in Java application frameworks in industrial use. By using a theory from case studies research approach, we have identified eight patterns of cross-language linking based on descriptions of the rules governing the links established by three frameworks from the Java world. For validation, the rules have been implemented in Java and applied to three example applications.

We believe that the patterns identified are helpful for two reasons. First, they describe a previously undocumented aspect of multi-language software applications and give insights into the complex nature that cross-language links exhibit in practice.

Second, they form requirements for any language intended to allow developers to formulate cross-language checking rules. In particular, it is our hope that researchers attempting to create a language (and accompanying software infrastructure) for formulating cross-language linking rules with the goal of supporting analysis, program understanding, and refactoring will benefit from the patterns discussed in this paper.

The semantic model implementations, language adapters, full rule specifications, and pattern annotations mentioned in this paper can be found online at http://www.xllsrc.net/.

Acknowledgment

We thank Thomas Neumeier for his contributions to the semantic models and language adapters created for this work.

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

References

[1] R.-H. Pfeiffer and A. Wasowski, "Taming the confusion of languages," in *Proceedings of the ECMFA 2011*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 312–328.

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

[3] P. Linos, W. Lucas, S. Myers, and E. Maier, "A metrics tool for multi-language software," in *Proceedings of the SEA 2007*. Anaheim, CA, USA: ACTA Press, 2007, pp. 324–329.

[4] P. Mayer and A. Schroeder, "Cross-language code analysis and refactoring," in *Proceedings of SCAM 2012*. IEEE, 2012, pp. 94–103.

[5] 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.

[6] 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.

[7] R.-H. Pfeiffer and A. Wasowski, "Cross-language support mechanisms significantly aid software development," in *Proceedings of MoDELS 2012*. Springer, 2012, pp. 168–184.

[8] K. M. Eisenhardt, "Building theories from case study research," *Academy of management review*, pp. 532–550, 1989.

[9] Eclipse Foundation, "Eclipse Modeling: MoDisco," 2013, http://www.eclipse.org/MoDisco/.

[10] R.-H. Pfeiffer and A. Wasowski, "Texmo: A multi-language development environment," in *Proceedings of ECMFA 2012*. Springer, 2012, pp. 178–193.

[11] H. Lochmann and A. Hessellund, "An integrated view on modeling with multiple domain-specific languages," in *Proceedings of SE 2009*, 2009, pp. 1–10.

[12] J.-M. Favre, R. Lämmel, and A. Varanovich, "Modeling the linguistic architecture of software products," in *Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 151–167.

[13] D. Strein, H. Kratz, and W. Lowe, "Cross-language program analysis and refactoring," in *Proceedings of SCAM 2006*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 207–216.

[14] 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.