A taxonomy of cross-language linking mechanisms in open source frameworks

Philip Mayer

Received: 4 December 2015 / Accepted: 30 November 2016

Abstract Non-trivial software systems are written using multiple programming languages. While the logic of a system is encoded using one or several general-purpose languages, more specialized parts of the systems are realized using domain-specific languages for aspects such as the user interface, configuration mechanisms, querying of databases, or support for internationalization. To bind all of these different parts together, the artifacts in individual languages are connected by using cross-language links which address artifacts across language boundaries. Many different ways for specifying and using such links have been conceived, and developers have to adhere to the concrete rules mandated by the runtime, framework or library which later performs the link resolution. In this paper, we present a taxonomy of the mechanisms of encoding cross-language linking in well-known open source frameworks from a developers perspective, which shows the choices that have been made and the options available in practice. We describe the process we followed, which is based in part on a survey of language combinations on GitHub and a survey of professional developers, list the dimensions and characteristics of our taxonomy in full, show the classifications of 22 frameworks and mechanisms, and discuss the impact of the choices on application developers.

Keywords multi-language development · cross-language linking · taxonomy · polyglot programming · software maintenance · classification · DSLs · GPLs · open-source software · frameworks

Philip Mayer
Programming & Software Engineering Group
Ludwig-Maximilians-Universität München
Munich, Germany Tel.: +49-89-2180-9376
E-mail: mayer@pst.ifi.lmu.de

1 Introduction

Non-trivial software systems are written using multiple programming languages, which includes general-purpose (GPLs) and domain-specific languages (DSLs). The artifacts written in the individual languages of a software system do not stand alone; rather, they are *linked* to one another as required by the programmer to achieve the system goals. For example, two GPLs may be connected in order to execute code which is closer to the operating system in a language such as C if the main system is written in Java. Another common use case are links between GPLs and DSLs, for example when using a HTML templating language like ASP in which values from a GPL such as C# are to be displayed. Finally, the DSLs themselves may also be linked, for example by using style classes declared in CSS in HTML.

Cross-language links are created by using *shared names* across language borders, which are specified by developers according to the rules of a runtime, library or framework which later perform their resolution at runtime. An example of a simple cross-language link between JavaScript and HTML by means of the jQuery library jQuery Foundation (2016) is shown in Figure 1. The identifier used is this case is `friend_plan_list` which is declared in HTML as the ID of the `div` element, and used in JavaScript by means of the jQuery framework to retrieve said element and change its style.

Listing 1 Defining a HTML div element
```
<div id="friend_plan_list" style="display:none;" />
```

Listing 2 Accessing a HTML element using jQuery
```
$('#friend_plan_list').css('display', 'block');
```

Fig. 1 A basic cross-language link between JavaScript and HTML, using jQuery

Being aware of exactly how cross-language links work in a system is key to program understanding, program maintenance and implementation of new features. However, when we look at the field of cross-language linking in practice we can see that there is a large number of runtimes, frameworks, and libraries which specify many different ways to express cross-language links in code, making it harder for developers to understand a system at hand.

It is thus the goal of this work to create a systematic taxonomy of cross-language linking *mechanisms*, that is, of the underlying choices taken by the individual frameworks and libraries to the specification of cross-language linking between languages via shared names. We are specifically interested in classifying *existing* cross-language linking implementations and thus base our analysis on well-known frameworks from the open-source world, and on how their mechanisms are intended to be used by *application developers*.

We use a standard taxonomy creating technique which will be discussed in section 2. The taxonomy itself is developed in section 3 and discussed in section 4. We compare with related work in section 5, and conclude in section 6.

2 Methods

Creating a taxonomy requires a systematic process which ensures — by construction — certain desirable properties of the taxonomy such as conciseness, robustness, comprehensibility, and explanatory nature. In this work, we have followed the process for taxonomy development created by Nickerson et al (2013), who suggest a 7-step iterative approach to taxonomy development. We describe these individual steps as we have followed them in section 3.

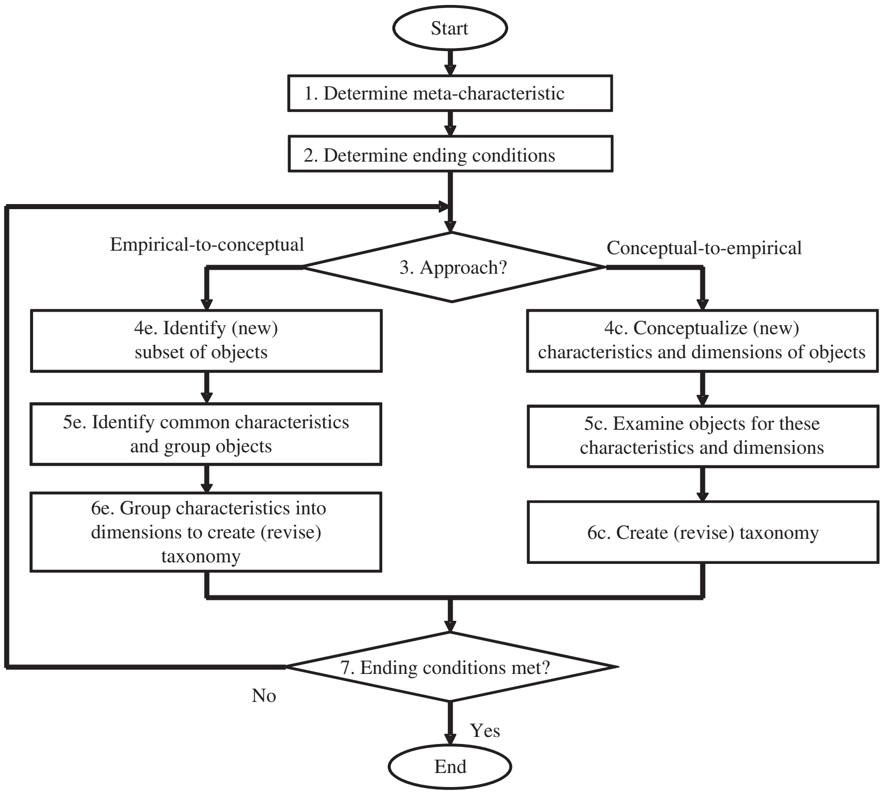
Start
1. Determine meta-characteristic
2. Determine ending conditions
Empirical-to-conceptual
3. Approach?
Conceptual-to-empirical
4e. Identify (new) subset of objects
5e. Identify common characteristics and group objects
6e. Group characteristics into dimensions to create (revise) taxonomy
4c. Conceptualize (new) characteristics and dimensions of objects
5c. Examine objects for these characteristics and dimensions
6c. Create (revise) taxonomy
7. Ending conditions met?
No
Yes
End

Fig. 2 Steps for Creating a Taxonomy, from Nickerson et al (2013)

2.1 Preliminaries

Before we start, we define several terms for use in the remainder of this work.

Mechanism vs. Framework Most cross-language links are established at runtime through a framework or library which is most often written in the GPL involved. Examples of frameworks are Gettext (de Mauro, 1999), Hibernate Red Hat (2016), or Wicket The Apache Project (2016). Each framework implements a certain type of linking mechanism, which is a more general concept and described the principle of linking. Our taxonomy will represent a classification of mechanisms; each framework we investigate implements an instance of a mechanism. The objects we place in the taxonomy are thus frameworks, but the leaves of the taxonomy represent mechanisms.

GPL and DSL We purposefully use very broad definitions of *general-purpose* and *domain-specific* languages: We classify each language which can be used and is regularly used to implement *arbitrary application code* as *general-purpose.* This includes languages such as Java, C/C++, JavaScript, or PHP. So-called *internal DSLs*, which actually use the syntax of the host language and thus can use its complete tool support are seen as part of the GPL.

By contrast, *domain-specific* languages are those which have a very distinct application area, either based on a technical or a business background, and have their own syntax. In this group, we find languages such as HTML for UI specification, Make or Maven for building the system, or querying languages such as SQL. We also explicitly include languages which are not traditionally seen as programming languages, that is configuration formats such as YAML, JSON, and XML. These languages are very common in bigger open-source projects, and almost always contain cross-language links. In the cases of these generic languages we always talk about the concrete *dialect* used. For example, the Google Android framework uses an XML-based format for UI specification. It is this dialect — written later as XML/AndroidUI — that is of interest.

Cross-language links and identifiers A cross-language link is any place in the system in which two languages are connected by identifiers which are used across language borders; and these must not change if the link is to be in working condition. We call such identifiers *cross-language identifiers.*

Note that we are are only interested in mechanisms here which establish direct links between the artifacts in two languages, not places in the code where such links merely exist *by implication*, which may happen, for example, when referencing an environmental variable from two or more languages. While consistency should be kept in those cases, there is no direct artifact link.

Declaration Side It is important to note that cross-language links themselves — i.e. the connection — is almost never explicitly specified in cross-language linking. Rather, what is specified are the end points of the links, that is, the artifacts which are to be linked. In GPL/DSL links, this creates an interesting question: Since one side of the link is a declaration and the other a reference, which side is the declaration on? We regard the *declaration side* of the link as the side on which the artifact may *stand alone*, that is exist without the link. The other side is a reference, i.e. it refers to the stand-alone artifact in the declaring language.

String vs. Element Specification Finally, we have found in previous research that a specific concern arises if cross-language links are specified in strings within GPLs, that is, concatenations of characters which can be arbitrarily manipulated by the language — i.e. generated from scratch, mixed, duplicated, passed around, and changed. This is a major difference to the use of language elements, the latter being in general static and stored in a fixed place. Thus, we will specifically address this distinction in the following. It is only relevant on the GPL side: Very few DSLs include the typical GPL string-manipulation capabilities, and we have not found an instance where they were used to actually manipulate cross-language links.

2.2 Users, Meta-Characteristics, and Ending Conditions

As stated in the introduction, the goal of this work to create a systematic taxonomy of cross-language linking mechanisms, that is, of ways of connecting languages, both GPL and DSL, via shared names. Thus, the taxonomy should be able to classify these ways, and by extension, also allow classification of concrete implementations of such mechanisms, that is, language-connecting implementations, which mostly come in the form of libraries or frameworks.

The first step of the taxonomy-building approach created by Nickerson et al (2013) requires us to determine the meta-characteristic of the taxonomy, which helps us determine the objects of interest. This in turn first necessitates specifying the intended *users* and the *purpose* of the taxonomy.

In our development of the taxonomy, we use our experience report on the creation of a cross-language linking tool Mayer and Schroeder (2014) and two surveys which we previously performed in this area. These surveys are an important input for choosing the users of the taxonomy.

The first survey is a mining study of GitHub projects in which we used an automated process to determining language combinations in open source projects Mayer and Bauer (2015). The study indicates that several languages are used, in parallel, in open source software. The second is a survey of 139 professional software developers who were asked for the languages and cross-language links that they see in their projects and for their opinions on multi-language development and cross-language linking (Mayer et al (2015)). The participants indicated that the existence cross-language linking is in many cases preventing them from making necessary changes to the source code for fear of side effects, and that they would appreciate more tool support. They also indicate that the number of languages used per project is non-negligible, a result which confirms our findings in the GitHub mining study and also (as we shall see later) matches the results in this work.

We believe, based on the answers of application developers about their problems with cross-language code and based on the large number of language combinations and frameworks out there that a taxonomy of the mechanisms will fill an existing knowledge gap. The main *users* of this taxonomy are thus application developers, who will benefit from the taxonomy as follows:

First, the taxonomy creates an awareness of the different options in cross-language linking in the first place, and the fact that many frameworks implicitly force these choices on application developers, even if they are not documented as such (for example, many frameworks do not even advertise the fact that custom DSLs are introduced as part of the package).

Second, it enables developers to understand the characteristics of linking mechanisms and in particular the pros and cons of each option which are discussed in section 4. For example, using string-based cross-language identifiers require different handling by developers than element-based ones.

Third, the taxonomy helps developers to then select frameworks based on this knowledge, identifying the framework which best fits their overall workflow. For example, a generative approach in cross-language linking may fit into an existing pipeline of source code generation; it may also present a problem if the build process is fully or partially manual up to now.

Two additional user groups can also be identified, which are framework designers as well as researchers. These groups can use the taxonomy as a base categorization scheme for further research, for example into usability differences of certain choices or unexplored alternatives.

Having specified the users, we now come to the *purpose* of the taxonomy, which is to distinguish the cross-language linking mechanisms which regard to how they are used by developers, i.e., how they have to specify the links.

The *meta-characteristic* for the taxonomy development process is thus the possible ways of specifying cross-language links for application developers.

The development of the taxonomy is iterative. In each step, we have the choice of following an empirical-to-conceptual approach (meaning we look at concrete examples and abstract from them), or an conceptual-to-empirical approach (meaning we use our abstract knowledge of the domain to identify new classifiers). Thus, we need an ending condition which tells us when to stop. Nickerson et al (2013) list several ending conditions, both objective and subjective. The method ends when all of them have been met. We discuss all objective and subjective ending conditions at the end of the process; during each iteration, we will restrict ourselves to checking a) that a reasonable subset of objects, i.e. frameworks, have been investigated; and b) that the current iteration has not produced any changes (e.g. additions) to the taxonomy.

3 Results

In this section, we trace the process we followed to creating our taxonomy through five iterations.

3.1 Iteration 1

For our first iteration, we have to choose an approach (step 3) to follow in the following three steps. In a previous publication Mayer and Schroeder (2014), we have implemented an integrative solution to finding, tracking, and allowing refactoring for cross-language links for three well-known frameworks from the Java world — Hibernate, a database access layer Red Hat (2016), Spring, a component configuration framework Johnson et al (2004), and Wicket, a UI library The Apache Project (2016). We thus use an empirical-to-conceptual approach for the first iteration, and select these three frameworks in step 4e.

All three frameworks make use of Java as the GPL; links are created between Java and the corresponding DSL. However, the frameworks differ in two significant ways. Since these align, we create one dimension in our taxonomy with two characteristics (Step 5e). In the first two frameworks — Spring and Hibernate — the declaration side of the linked element is in the GPL. In both cases, Java classes are referenced, by name, from XML/MVC (Spring) and HQL (Hibernate). In both cases, the link end in the GPL is an actual part of the language — a class name — not a string. An example is shown in Figure 3.

Listing 3 Bean declaration in Java
```
package net.xllsrc.transform;
public class FileGenerator { ... }
```

Listing 4 XML file defining a bean
```
<bean class="net.xllsrc.transform.FileGenerator" />
```

Fig. 3 Java GPL element names used in the XML-based Spring DSL

The third framework is different: Wicket allows access to HTML elements from Java. Thus, the declaration side is on the DSL side, and in fact access in the GPL is provided through a string — the name of the HTML element. This is similar to jQuery and thus the code shown in Figure 1.

Although it already seems that declaration side and the element/string dichotomy might be different dimensions, they currently align, and so we only create one dimension called *declaring side* with two characteristics in our newly created taxonomy (step 6e). Table 1 shows the taxonomy so far, with the three frameworks as representatives of the two mechanisms identified.

Table 1 Taxonomy after Step 1.

Framework	*Declaring Side*	
	GPL	*DSL*
Spring MVC	X	
Hibernate	X	
Wicket		X

We now step out of the empirical-to-conceptual branch and go to step 7. Since the taxonomy was changed and we have more frameworks to consider, ending conditions are not met and we continue with another iteration.

3.2 Iteration 2

We are back to step 3. For our second iteration, we decided to use the empirical-to-conceptual approach again. As input, we use another previous publication

Mayer and Bauer (2015) in which we analyzed languages and language combinations in a diverse set of 1150 open source projects. In particular, we have identified five common use cases of GPL/DSL interaction in this work, which are user interface, configuration, database access, shell scripting, and build management. We have now investigated frameworks from each category.

We first look (again) at the user interface area, as we wondered whether there might be cases in which the artifact declaration side and the artifact type are not aligned, i.e. whether we could have a DSL-declared artifact which is not accessed via string on the GPL side. In fact, such an example is present in the Windows Presentation Foundation framework Microsoft (2016) which is also a UI framework and which accesses XAML (an XML format) from .NET languages such as C#. In this framework, the UI is declared in XAML, and accessors are *generated* by the framework to be used in C# — in comparison to the entirely manual approaches that we have seen so far.

We note here that first (step 5c), we really need to split artifact declaration side and element/string based access into two new dimensions called *Declaration Side* and *GPL Artifact Type*, and second, we need another dimension which reflects the *manual* or *generated* nature of the mechanism, which we call the *Generation* dimension.

For the configuration area, we selected the Gettext framework (de Mauro, 1999) which is used for internationalization in the C/C++ world (Figure 4).

Listing 5 C code accessing an i18n key

```
printf(_("My name is %s.\n"), my_name);
```

Listing 6 Definition of an i18n key with translation

```
#: src/name.c:36
msgid "My name is %s.\n"
msgstr "Je m'appelle %s.\n"
```

Fig. 4 Using Gettext for internationalization from C

This is an interesting mechanism as it uses string-based reference IDs in the GPL which are generated into the DSL, the line-based Portable Object (PO) format; thus, again, we have a generated-based approach. Secondly, a new dimension comes into play here: Obviously, internationalization means that the links created by the framework may change, i.e., they might be redirected based on a configuration option (selection of a different language). The framework explicitly allows for this feature, which we call *Link Configurability.*

This, in turn, led to an reexamination of our existing frameworks: Do *they* allow configurability? In Gettext, the framework itself contains an explicit mechanisms for switching link targets. Is this the case in the other examined frameworks, that is Spring MVC, Hibernate, Wicket, or WPF? At first glance, this doesn't seem to be the case. However, in the case of Spring, it is possible to use an entirely different XML/SpringMVC file which contains a different configuration for the system, and inject that file via external means. Thus, while the mechanism does not explicitly support configurability, it is possible to configure it *implicitly*. We have chosen to set the *implicit* tag for all frameworks in which complete files can be switched with relative ease to change link targets; all others are marked *manual* as before. Thus, we add the dimension of configurability with three characteristics.

In the database access area, we have already discussed Hibernate, but want to extend our reach to other GPLs as well. We thus investigate the database part (ActiveRecord) of Ruby on Rails Rappin (2008), a Ruby web framework. Ruby on Rails uses a technique in which both Ruby and SQL are specified separately and matched, by the framework, based on the names of the tables and columns. However, it is similar to Hibernate within our taxonomy.

Finally, we come to the two areas which affect development work, which are shell scripting and build management. For shell scripting, we look at the Scala language Odersky et al (2004) and at the common use case of starting and stopping Scala programs using a shell such as Bash or Batch (Burtch, 2004). We find that Scala fully-qualified names are used from within the scripts, that the artifact is declared in the GPL as an element, that specification is manual and linking is non-configurable. An example is shown in Figure 5. The technique falls within our existing taxonomy dimensions and characteristics.

Listing 7 Object declaration in Scala
```
package opennlp.textgrounder
object GeolocateDocumentTagApp  { ... }
```

Listing 8 Using a qualified Scala name in a shell script
```
tag='run opennlp.textgrounder.GeolocateDocumentTagApp'
run-nohup --full-id "tag" tg-geolocate "$@"
```

Fig. 5 Startup of a Scala app from a Shell script

For build management we look at Apache Maven, which is a well-known build management and scripting tool and language based on XML. As in Scala, fully-qualified class names are used within the XML/Maven format with the same properties as above, and thus takes the same place in our taxonomy.

Thus, after going through the five common use cases of GPL/DSL interactions from Mayer and Bauer (2015) in this iteration, we arrive in step 6c at a new taxonomy which is shown in Table 2. We step out of the currently chosen conceptual-to-empirical branch onto step 7 of the taxonomy building, where we again have to decide whether the ending conditions are met. Since we have added new elements to the taxonomy, they are not. We also have more frameworks to classify, so we continue in step 3.

3.3 Iteration 3

In iteration 3, we used the results of our study of professional software developers (Mayer et al (2015)) who were asked for their experiences with multi-language programming and cross-language links. One of the questions asked was about language connections that developers have seen in their last projects. Six combinations were listed by more than 30 people each. These are Java and XML-based languages, Java and SQL, JavaScript and HTML, Java and the .properties format, as well as HTML and CSS. We discuss now the last three, which are new. This is again an empirical-to-conceptual iteration.

Table 2 Taxonomy after Step 2. El: Element; M: Manual; I: Implicit; E: Explicit

Framework	*Declaring Side*		*GPL Artifact*		*Generation*		*Configurability*		
	GPL	*DSL*	*El*	*String*	*No*	*Yes*	*M*	*I*	*E*
Spring MVC	X		X		X			X	
Hibernate	X		X		X		X		
Wicket UI		X		X	X			X	
WPF		X	X			X	X		
Gettext	X			X		X			X
ActiveRecord	X		X		X		X		
Shell/Scala	X		X		X		X		
Maven	X		X		X		X		

Regarding JavaScript and HTML, jQuery is a well-known and used framework for interacting with HTML (and CSS) from JavaScript. In particular, the special function `$()` is used to access HTML elements, which are declared (by ID) in HTML and can then be manipulated from JavaScript, making these artifacts DSL-declared and used via string from the GPL. We have already seen a code example in Figure 1. There is no generation, and link configurability is manual. We do not need to change the taxonomy for this framework.

The .properties format Oracle (2016) is a built-in Java mechanism for reading configuration values from a file. The configuration values are declared in key/value format in the DSL, and later accessed by name (DSL-declared, string-based access). Specification is manual, and configurability is implicit (by changing the file). No changes are needed to the taxonomy.

The last pair which was mentioned often by developers is HTML and CSS Schafer (2005). This is a pair of DSLs, and thus the time has come make a choice whether to keep the taxonomy as it currently is — namely, a taxonomy not of cross-language interactions in general but of GPL/DSL interactions — or whether to add a new dimension which shows the types of languages linked, and which would then also need to include GPL/GPL links.

We have decided to add this new dimension, which we call *Language Types* and which has (as of now) two characteristics, which are *GPL/DSL* and *DSL/DSL*. We note that two of our existing dimensions — namely the GPL artifact type and the declaration side — are specific to GPL/DSL links and thus need to be moved; they are essentially subdimensions of GPL/DSL links. For reasons of presentation however, we keep the existing table format, and grey out the appropriate cells in case of non-GPL/DSL mechanisms.

Coming back to the case of HTML and CSS links: These are links which are managed by the browser. Essentially, style classes are declared in CSS

Table 3 Taxonomy after Step 3. GPL El: GPL Element; Gen: Generation; Conf: Configurability; G/D: GPL/DSL; D/D: DSL/DSL; El: Element; Str: String; N: No; Y: Yes; M: Manual; I: Implicit; E: Explicit

Framework	*Lang Types*		*Declaring Side*		*GPL Art.*		*Gen.*		*Conf.*		
	G/D	*D/D*	*GPL*	*DSL*	*El*	*Str*	*N*	*Y*	*M*	*I*	*E*
Spring MVC	X		X		X		X			X	
Hibernate	X		X		X		X		X		
Wicket UI	X			X		X	X			X	
WPF	X			X	X			X	X		
Gettext	X		X			X		X			X
ActiveRecord	X		X		X		X		X		
Shell/Scala	X		X		X		X		X		
Maven	X		X		X		X		X		
jQuery	X			X		X	X		X		
Java Prop.	X			X	X		X				X
HTML Styles		X					X			X	

to be applied to HTML elements with the `class` tag, as shown in Figure 6. No generation is involved; however, we have implicit configurability: A simple exchange of the CSS file leads to a different display style.

Listing 9 Definition of a style class in CSS
```
.logoff { width: 100%; margin-top: 20px; }
```

Listing 10 Use of a CSS style class from HTML
```
<div class="logoff" />
```

Fig. 6 A cross-language link between DSLs: CSS and HTML

We thus add one new dimension to the taxonomy (the language types) with two characteristics (GPL/DSL and DSL/DSL), noting that we will have to revisit this in the next iteration. The resulting taxonomy is shown in Table 3.

We note that we have both added a dimension and need to investigate further frameworks and thus move to iteration 4.

3.4 Iteration 4

In iteration 3, we noted that we missed GPL/GPL interactions in our taxonomy. We thus want to add this area to the taxonomy. To make sure we do not miss any important GPLs, we select at least one example from each of the top 10 GPL languages that we found in Mayer and Bauer (2015), i.e. the top 10 GPLs in our diverse set of open source projects from GitHub (except PHP, which we will discuss in the next iteration), making this an empirical-to-conceptual iteration.

In particular, we select our frameworks from three common areas of GPL/-GPL interactions within this language set: Those on common language runtimes (such as the Java VM (Yellin and Lindholm, 1996) and the .NET CLR

(Box and Pattison, 2002)), those which directly link two languages with separate runtimes, and generic frameworks which support arbitrary GPL interactions through plug-ins or library add-ons.

We thus discuss six frameworks in total, two from each area. From the first area, we look at interactions on the Java VM and on the Common Language Runtime, as well as the (trivial) connections between C, C++, and Objective-C. For the second area, we look at direct connections between languages on the example of the Java Native Interface (JNI, (Gordon and Essential, 1998)) which connects Java and C/C++. For the third area, we discuss generic frameworks with support for several GPL languages; the *Thrift* framework Slee et al (2015), and the *Swig* framework SWIG Developers (2016) for connecting arbitrary languages to C/C++ (where we look at Perl and Python).

The Java VM, the .NET CLR, and the C/C++/Objective-C infrastructure are similar in their approaches to cross-language linking. Essentially, linking is transparent: It is simply possible to use artifacts from another language by importing them in the language-typical syntax. Thus, for example, a C# class can be directly re-used in VB.NET, or a Scala class in Java. In Objective-C, C and C++ it is even possible to directly mix syntax within files, and call the declared elements by name. Thus, all three cases fall (obviously) into the new characteristic *GPL/GPL*; we thus have to add this categorization to the taxonomy. There is no generation involved, since both sides are implemented manually, and there is no configurability.

Listing 11 Thrift declaration file
```
service Calculator extends shared.SharedService {
  i32 add(1:i32 num1, 2:i32 num2)
```

Listing 12 Java server implementation
```
public class Calculator {
  public interface Iface extends SharedService.Iface {
  public int add(int n1, int n2) throws TException;
```

Listing 13 Python client implementation
```
class Client(shared.SharedService.Client, IFace)
  def add(self, num1, num2):
    self.send_add(num1, num2)
    return self.recv_add()
```

Fig. 7 Declaration of GPL/GPL links in Thrift

In the second area, we look at the Java Native Interface which connects Java and C. Here, we have a generation approach: Out of the native Java methods, C (stub) code is generated. Again, however, there is no configurability.

Third, we look at generic frameworks which support multiple GPLs regardless of common runtime. The first is the well-known *Swig* framework which supports, among other languages, Perl and Python and allows turning C/C++ declarations into scripting language interfaces. Swig requires an interface dec-

Table 4 Taxonomy after Step 4 (top) and 5 (including bottom part). L.Type: Language Types; Side: Declaration Side; GPL A: GPL Artifact; Gen: Generation; Conf: Configurability; GD: GPL/DSL; DD: DSL/DSL; GG: GPL/GPL; G: GPL; D: DSL; El: Element; Str: String; N: No; P: Partial; F: Full; M: Manual; I: Implicit; E: Explicit

Framework	*L. Type*			*Side*		*GPL A.*		*Gen.*			*Conf.*		
	GD	*DD*	*GG*	*G*	*D*	*El*	*Str*	*N*	*P*	*F*	*M*	*I*	*E*
Spring MVC	X			X		X		X				X	
Hibernate	X			X		X		X			X		
Wicket UI	X				X		X	X				X	
WPF	X				X	X			X		X		
Gettext	X			X			X		X				X
ActiveRecord	X			X		X		X			X		
Shell/Scala	X			X		X		X			X		
Maven	X			X		X		X			X		
Android UI	X				X	X			X				X
jQuery	X				X		X	X			X		
Java Prop.	X				X	X		X				X	
HTML Styles		X						X				X	
Java VM			X					X			X		
.NET CLR			X					X			X		
C/Obj/++			X					X			X		
JNI			X					X	X		X		
Swig			X						X		X		
Thrift			X					X		X		X	
Android UI	X				X	X			X				X
C++/.cfg	X				X	X		X				X	
Twig	X			X			X	X			X		
JEE (JSP)	X			X		X		X			X		
Maven Prop.		X						X				X	

laration, but generates the actual interface code, making it a generative approach. Again, there is no configurability, neither implicit nor explicit.

The final framework we look at is *Thrift* (Slee et al, 2015). Thrift is of particular interest since it does not, as JNI or Swig, fix one or both languages (Java and C in JNI; C/C++ in Swig) — instead, it generates code on both sides of (relatively) arbitrary languages in a networked environment, i.e., it generates a client and a server where the languages can be chosen from a non-constrained list. This is something new: Up to now, all we had were generation approaches which generated code on one side of the language gulf. Thrift, however, uses a *third* artifact — an interface file declared in a custom DSL — from which code in both GPLs is generated (example in Figure 7).

We thus change the *generation* dimension to three categorizations: *No generation* (*manual*), *partial generation*, and *full generation*; we place Thrift in the latter category. As Thrift uses URLs to point to the other side which can be easily changed, we have an implicit mechanism.

Thus, we add two new categorizations to our taxonomy: The *GPL/GPL* category to the language types dimension, and the *full generation* category to generation. The new taxonomy is shown in Table 4 (top part). We have added two new categorizations to the taxonomy and are thus not done yet.

3.5 Iteration 5

In the fifth and, as it will turn out, final iteration, we used our own background in software engineering to determine whether there are any important frameworks or language combinations we have missed. Thus, in this iteration we use a conceptual-to-empirical approach.

The first framework we found comes from the mobile world, namely in the Android framework (Burnette, 2009) which makes use of its own dialect of XML for user interface description. In Android, screens or parts of screens are written by hand in XML. Each element has an ID which is later referenced from Java; in fact, a tool generates a Java file (the `R` file) which contains all of these IDs as public field names. Thus, the link declaration side is on the DSL side, and the GPL artifact type is an element. One interesting note here is that Android allows different screen configurations, for example landscape and portrait, with the same screens. In this case, two XML files can be placed in different directories which are exchanged on demand; we thus have *explicit configurability.* No new dimensions or characteristics need to be added.

Listing 14 Defining a value for a template in PHP

```
return $this->render('ElemRefEtu.html.twig', array(
   'modules' => $moduleElements, ...));
```

Listing 15 Using a value in a Twig template

```
{% if modules is empty %}
  <div class="alert alert-danger" ..." />
{% else %}
```

Fig. 8 Twig DSL code accessing declarations in PHP code

Secondly, we have so far only looked at configuration approaches in the Java world; however, other GPLs also make use of configuration files, such as the common .cfg files in the C++ world. However, as expected, the mechanisms are entirely similar: We have a GPL/DSL link with the declaring side in the DSL, the use of a string to access it, no generation, and implicit configurability.

One GPL language which was missing in our discussion so far, and is part of the top 10 GPLs from Mayer and Bauer (2015), is PHP. Thus, we looked at the Twig framework Potencier et al (2015) which allows the use of HTML-based templates from PHP. Twig is an interesting framework in that elements to be used in the template are declared as strings in PHP and then used in the DSL: The declaration side is thus the GPL, but the declaration is in strings, which is a new combination. The code is shown in Figure 8. Compare, for example, the JEE framework and the use of JSPs from Java man Chung (2013): Here, the the JSP template uses elements from the GPL, not strings. We add both frameworks to the table; no new elements are required.

Finally, we have only one example yet of a DSL/DSL link, namely between HTML and CSS. Another example is the use of .property declarations in Maven. Here, key/value pairs are declared as usual in the property file and then imported and used by name in XML/Maven. No generation is involved, and as the file can be easily replaced, we have implicit configurability.

We have not added any new dimensions or categories in this iteration, and we have no new language combinations or frameworks to look at. We also believe that we have met the other ending conditions (discussed below) and thus move to the end of the steps in Figure 2. The final set of frameworks identified in the taxonomy is shown in Table 4 (top and bottom).

3.6 Ending Conditions

Nickerson et al (2013), as part of their taxonomy process, list eight objective and five subjective ending conditions. Some were already discussed above; i.e. we have not added, merged, or split any new dimensions in the last iteration. At least one object is classified under every characteristics of every dimension, and every dimension, characteristic, and cell is unique (i.e. there is no duplication).

The final objective ending condition requires that all, or a representative set of objects be examined. Obviously, the domain we investigated is huge and unknown in its extent. We have therefore taken care to investigate, by design, frameworks and mechanisms which are well-known and used in the open-source world. This process has been shown above. In short, we have examined a set of 22 frameworks and mechanisms from the open source world. We started out with a set of frameworks which were thoroughly investigated to provide tool support. We haven then extended this set with a conceptual sample, namely the five top DSL use cases from our EASE study. Then, we have reached out to the industrial community for their top language combinations. Finally, we came back to the EASE study to look at the top 10 languages to be sure we did not miss anything important. We thus believe that we have created a taxonomy which mirrors the existing dimensions and characteristics for those languages and language combinations in use in the open-source world.

The subjective ending conditions listed by Nickerson et al (2013) are conciseness, robustness, comprehensiveness, extensibility, and explanatory nature. We have created a taxonomy with five dimensions total, each with two to three characteristics, which we believe to be meaningful without being unwieldy or overwhelming and thus *concise*. It is *robust* and *comprehensive* in that it provides meaningful differentiation between the objects as we have seen in the above discussions, and gives us the ability to talk about all identified characteristics of the frameworks involved which are important for their users.

Furthermore, the individual dimensions and characteristics explain how the mechanism is to be used by developers, i.e. which language combinations must be used, where the declaring side is placed, how the referencing mechanism works, whether the approach is generative and whether it can be configured. Thus, the taxonomy does not describe the objects in complete detail but rather provide useful *explanations* of the nature of the objects both existing and for future classification. Finally, we believe that the taxonomy is open for *extension* as we have a fairly flat hierarchy.

3.7 Final Taxonomy

The final taxonomy is shown in feature diagram notation Batory (2005) in figure 9. Each cross-language linking mechanism can be categorized according to five different dimensions (in grey); for each, one of multiple mutually exclusive characteristics must be chosen. The two lower features can only be selected if the GPL/DSL characteristics was chosen in the *Language Types* dimension.

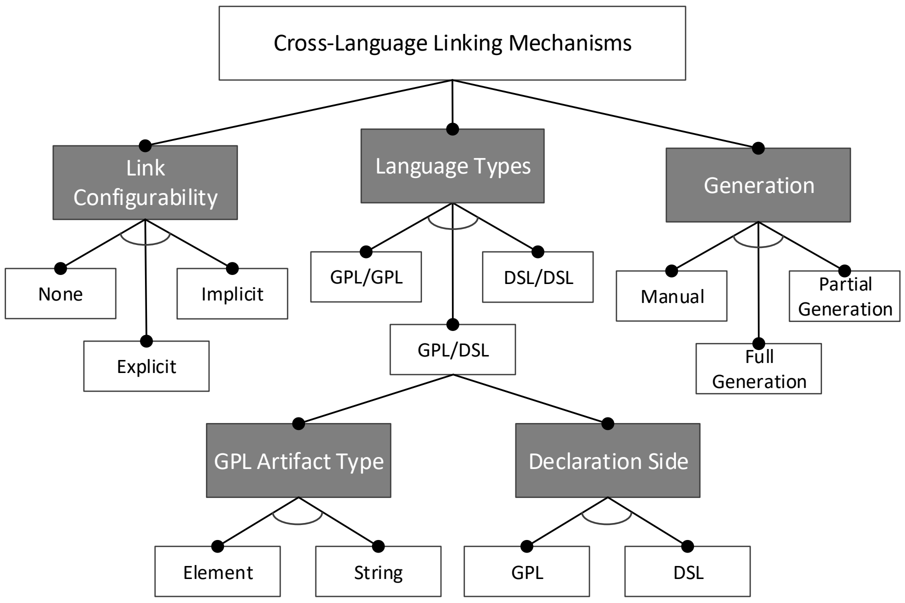

Fig. 9 The taxonomy in feature diagram notation (see Batory (2005))

A short textural summary can be given as follows:

– *Language Types* Which language types are involved in the link? Does the link exist between general-purpose languages (*GPL/GPL*), domain-specific languages (*DSL/DSL*), or a combination of the two (*GPL/DSL*)?
– *GPL Artifact Type* (only for GPL/GPL combinations) Is the artifact on the GPL side a *program element*, or is it a *string*?
– *Declaration Side* (only for GPL/GPL combinations) A cross-language link has a declaring and a referencing side. On which side is the declaration placed? On the *GPL side*, or on the *DSL side*?
– *Generation* Are both sides of the link specified *manually* by a programmer? Is *one side generated*? Or are *both sides generated* from an external specification?
– *Link Configurability* Is the linking mechanism based on *hardcoded links*? Or can it be configured, that is, are multiple linking targets possible? Is this indirection *implicit* or *explicit?*

We discuss the impact of choosing each characteristics on the user in the following section.

4 Discussion

4.1 Discussion of the Taxonomy Dimensions

We will now discuss the individual dimensions of the taxonomy and their impact on the intended users of the taxonomy, i.e. application developers. Where appropriate, we will also discuss the impact on our secondary users, that is framework developers and researchers.

4.1.1 Language Types

A rather general dimension of a cross-language linking mechanism is the types of languages that are linked. In many cases, a decision about the types cannot be made independently but is a consequence of other choices. For example, the requirement to use several GPLs in a project may stem from the available execution environments; consider the use of JavaScript in web application clients where there isn't a lot a choice.

Similarly, domain-specific languages are often introduced to a project by a framework — that is, the choice is made to use a certain framework due to organization issues or customer requirements, and it is the framework which requires the use of a certain DSL (for example, the Maven XML dialect, or the use of a certain HTML templating language). In general, the question of which language is best suited for a particular purpose — in particular, when using DSLs — is hard to answer and out of scope for this taxonomy.

As our taxonomy focuses on linking between language types, we however want to add a word of caution: Not only maintenance and tool support for the DSLs must be considered. Additionally, the *interaction* between any DSL and the main GPL of a project are an additional maintenance and evolution issue (and even more so than between GPLs). Thus, it is important for application developers to be aware of the concrete languages introduced when choosing a framework. Furthermore, for researchers, the linking mechanisms between the languages should be included as a factor in future experiments on language comprehensibility.

4.1.2 GPL Artifact Type

The selection of the GPL artifact type has key consequences for the use of a cross-language linking mechanism. In particular, the use of strings opens up a whole range of link manipulation options which are not available if program elements are employed as link end points. As they can be placed in variables, strings may be passed around in the program and thus, the same place in the code may be linked differently depending on the data flow. While cross-language artifact names can be placed in constants and are thus immutable, there is also the option to assemble link identifiers at runtime, including the use of user input to determine the linked elements.

Note that most frameworks do not encourage the use of string manipulation; however, the fact that it is possible usually means that it will be used. As an example, consider the JTrac project (see Mayer and Schroeder (2014) for a detailed discussion) which uses Hibernate for database access. As part of a search feature, the column names to be selected are based on user input. While this approach is flexible and generic, it is also very difficult to follow and maintain such code, not to mention the security issues involved.

Besides issues of understandability and maintainability, the use of string identifier manipulation is also a problem for the creation of design-time tool support for cross-language linking, a topic which has been investigated in detail in Pfeiffer and Wasowski (2015). For such support, it must be possible to determine statically where linking occurs, for which the linked identifiers on both sides of the language gulf must be determined. In the presence of string manipulations, full-blown data flow analysis may be required to determine the actual value of a linked identifier. In Mayer and Schroeder (2014), the authors describe the problems involved in finding such identifiers in Java code using the Hibernate and Wicket frameworks.

However, the use of strings makes one thing clear to application developers, namely that the identifier used are definitely not part of the current language and thus must be treated carefully when renaming. In other words, their special status and cross-language semantics may be more visible, which in turn, may not be the case with program element names. For example, the fact that a class or method name is referenced from a DSL is usually not apparent in the code, and thus developers may not even be aware of its cross-language nature.

This is a problem since they may rename such identifiers without realizing that this breaks a link. The fact that the implementations of automated rename refactorings suggest that all references are taken care of (which is usually not the case across language borders) is not helpful in this situation either.

4.1.3 Declaration Side

The declaring side of the artifact in a cross-language link determines how application developers think about their systems, and have an impact on the development workflow, including which parts of the system are designed, or coded, first.

It is interesting to see that there is great variance in this dimension across the frameworks we have examined, even if the frameworks share the same purpose. For example, classic (non-web) UI frameworks tend to define all cross-language link artifacts on the DSL side (consider the Android framework), while web frameworks which use HTML may also use GPL-declared artifacts for output (using templates, such as in the Twig framework). In system configuration, we also see both directions: Dependency injection containers such as Spring reference GPL-declared classes, while other approaches retrieve DSL-declared keys to determine a certain configuration value.

In some cases, the declaring side is a result of technical constraints, for example the need to use HTML in web applications or the history of desktop or mobile user interfaces which is widget-based. However, we believe it is also interesting to ask whether a certain direction is ultimately easier to work with, understand, and maintain by developers. The answer to this question may obviously be different depending on the domain area; determining the properties of this distinction is important future work for researchers.

If the current trend towards the creation of domain-specific languages continues — for example, by a continued interest in language workbenches (Völter and Visser, 2010) and project-specific, small languages — being able to give advice on benefits and problems of the declaring side of artifacts will be useful, in particular to framework developers.

4.1.4 Generation

Similar to the artifact declaration side, the availability of generation has implications on how application developers think about approaching the issue of cross-language linking. In fact, the ability to generate code adds an additional abstraction layer which has the specific aim of disguising the intricacies of cross-language resolution. For example, the .NET approach of UI access allows the use of normal C# classes and fields instead of having to deal with an XML UI specification.

Code generation tools have the major advantage that the generated code makes a framework convenient to use. Mostly, GPL code will be generated which eases access to DSL code by encapsulating the technicalities of DSL parsing and allow the use of program elements instead of strings. Code generation also guarantees that the link itself is correct — as long as the regeneration is always run after a change to the specification.

However, it is important to note that code generation — even in the full generation case — only ever generates *handles* which are later used manually; for example, the C# field mentioned above will be referenced in other, manually written C# classes which contain the actual logic of the program, i.e. what to do with the values provided in the UI (or setting them). This is important since it means that not only a change in the original name (without regeneration) will break a link, but also that a regeneration after changing a name is not enough, but all references to the name need to be changed as well, which is often not supported by the generation tools.

Using a mechanism which includes code generation introduces additional tools into the tool chain, which must be kept updated as well. In long-lived projects, this may be a problem if changes must be made to generated code and the original tool chain is no longer functioning.

It is important to note here that using code generation does not mean that the *links themselves* are already established at design time. Code generation is merely used as a development tool — ease of workflow — while the actual link is still established at runtime by the framework. If code is changed partially and without regeneration, the application will thus still fail at runtime.

4.1.5 Link Configurability

The final dimension of the taxonomy is whether and how configurability is natively supported by a mechanism. Explicit link configurability extends cross-language linking by another dimension — not only is there a cross-language link, but it is also a point of divergence for system behavior.

In some cases, it is natural to expect and provide for such divergence. Consider, for example, the Ruby on Rails database configuration approach, which explicitly support multiple environments. Such explicit support means that basic design decisions of a mechanism reflect the retargetability of links and thus program understandability does not suffer.

This may not be the case in mechanisms in which implicit configurability is possible. Link retargeting in this case may be performed on multiple levels: By swapping out entire groups of links (as can e.g. be done by loading different DSL files) or by adapting individual links. The latter is for example always possible if the GPL artifact type is string-based.

We believe that implicit configurability may thus be a problem for program understanding and maintenance and thus for application developers. In case link redirection is an intended feature, it should be supported explicitly by the mechanisms. Whether this is always possible is another question, as is the actual severity of its impact on system maintenance.

In general, researchers as well as framework developers should investigate whether the point where languages are crossed could not be made more explicit, thus creating real interfaces instead of the rather implicit links that are currently in place.

4.2 Threats to Validity

The creation of a taxonomy is based on the analysis of a selected set of objects (open-source frameworks, in our case), which are categorized by a process such as the one we have followed in Section 2 based on a certain viewpoint, which in our case was that of the eventual application developer. Naturally, this process includes a subjective viewpoint and can raise concerns and objections if the viewpoint of readers is different.

A first threat to validity may arise from the selection of the objects used for creating the taxonomy, i.e., the selection of relevant frameworks (empirical basis). As shown above, we have investigated a total number of 22 open source framework from a variety of domains. We have based our selection on two empirical sources: First, our previous investigation in language co-occurrence on GitHub Mayer and Bauer (2015), from which we have taken the most important DSL domains as well as the top 10 GPLs found. The GitHub project selection was based on a *diverse* approach which ensured that the projects selected represent a full spread across the spectrum based on attributes such as size, main language, commit history etc. Mining GitHub is not without its problems Kalliamvakou et al (2016); however, our study focused on very concrete data (programming language use) which can, as attested in Kalliamvakou et al (2016), be viewed as solid information. We also believe that our diverse selection approach and follow-up statistical analysis which takes the meta-data into consideration yields valid and useful information which is, in particular, adequate as the basis for the current paper.

The second source was our survey of 139 professional software developers Mayer et al (2015). Various questions were posed to developers, of which two are relevant here. The first is that developers were asked to list the languages they see linked in practice; we have then made sure to include frameworks which cover these languages as well. In each selection case, we have attempted to use well-known and stable frameworks from the open source domain as input, as they will be most relevant to the intended target audience (application developers). The number of developers surveyed is 139, which is not a large sample; however, we believe that the inclusion of this data as a further basis for framework selection and thus using a mix of open source mining and developer feedback strengthens our claim that the frameworks used for building the taxonomy represent a diverse and comprehensive selection.

Obviously, the framework list is not complete — there are more frameworks out there. On the other hand, we have seen no indications of additional dimensions or categories to be discussed in the last iteration, and we have chosen this as the point to stop given the multitude of languages and frameworks out there. Nevertheless, it might be possible that a different taxonomy emerges with another sample. We look forward to other researchers attempting classifications in this area. If their taxonomies do not agree with ours, we have to investigate the differences and attempt to merge the schemata appropriately.

A second threat arises from the taxonomy itself (construct validity) — the chosen dimensions and characterizations — i.e. whether those are appropriate to the domain, whether all relevant dimensions and characteristics have been found, and whether each of them represents an independent concept (relevance in dimensions, mutual exclusiveness in the characteristics). As we have shown in Section 2, we have followed a careful and incremental approach to building the dimensions and characteristics of this taxonomy, always having the goal (usefulness to developers) in mind. In the creation of a taxonomy, one must strive for a compromise between too general concepts and too specific "cells" of classification. We believe that the dimensions and characteristics we have identified follow quite naturally from the selected objects, and based on our own experiences as software developers and users of many of these frameworks that they are relevant and useful to the users of such frameworks. However, as has been discussed before, the real usefulness of a taxonomy can only be determined by whether it is actually used by others.

A slightly different concern is the generalizability of the taxonomy (external validity). Obviously this is a consequence of the two discussion above, i.e. whether a valid set of inputs has been used and whether the dimensions and characteristics chosen are appropriate. During the development of the taxonomy we have found many instances of frameworks which already fit into the taxonomy, particularly (and obviously) in the last step. We also believe that since we have included frameworks from very diverse domains (use cases, such as UI, Build, Scripting, etc.) across many GPLs and DSLs, we have a solid basis for future classifications. Nevertheless, the future classification of frameworks may show that there are indeed other mechanisms in use, and the taxonomy may need to be changed in response.

Finally, there is also the threat that we have misclassified the frameworks during our creation of the taxonomy, i.e. they do not actually fall within the selected dimensions or characterizations. To confront this problem, we have investigated both the description of each framework through documentation material (i.e., available help web sites, books, and tutorials) as well as their actual use in open-source projects. Most of the snippets shown in the previous sections come from independent open source projects — i.e., projects not associated with the framework developers. This cross-check serves to establish more trust in the classifications we have made.

5 Related Work

There are several works in the literature already which deal with cross-language linking; some are furthermore related in the sense that they deal with multiple languages in general without a focus on linking. As we shall see, none of them addresses the area we have outlined above.

First, there are two taxonomies in the general area of our work which are relevant. The closest is Tomassetti et al (2013), who identified language interactions by analyzing the commits of the open source project Hadoop, and use six categories to classify each individual link. The major differences lies in the fact that we want to classify complete mechanisms, while their focus is on the individual interaction. In a similar direction, Pfeiffer and Wasowski (2015) have investigated and classified features of *design-time tool support* for cross-language linking. Thus, their taxonomy is focused on how to find and relate links programmatically to aid visualization, navigation, refactoring, and static checking. This taxonomy is complementary to the one we are proposing here, which is focused on the linking mechanism specifications themselves.

Second, several authors discuss and/or quantify the extent of multi-language programming in general, that is the occurrence of many languages without a specific focus on their connections. Three recent data mining studies on open source software (Mayer and Bauer, 2015)(Tomassetti and Torchiano, 2014)(Delorey et al, 2007) have investigated this area, focussing on co-occurence of languages. In the first two studies, GitHub was used as the provider; in the third, SourceForge. All studies find multi-language programming to be common in practice and including a mixture of GPLs and DSLs. Another recent study Ray et al (2014) compared code quality in different GPL languages on GitHub, also without addressing their connections specifically. A key finding was that defect proneness does depend on certain factors such as the language type (functional, procedural), typing (static vs. dynamic) and memory management (managed, unmanaged). These studies all consistently reinforce the need for further research into multi-language programming based on the extent of its use in practise.

Third, several authors have suggested new and better ways of *designing language interactions*. Favre et al (2012) and Lämmel and Varanovich (2014) have investigated the question of the linguistic architecture of projects with a specific focus on *modeling*, that is, design-time support. Such support may help alleviate problems of understandability in such systems. Then, several works have focussed on the technicalities of GPL/GPL interactions. Ekman et al (2007) have investigated language interoperability on the Java VM, looking at *technical* realizations of calls between Smalltalk, Java, and BETA, with the aim of making them transparent to use. Gybels et al (2006) discuss inter-language reflection, where again the aim is to be able to use reflection in a transparent way across languages (in this case Agora/Java and SOUL/Smalltalk). Zdun (2004) discusses split objects which are conceptual entities that reside in two languages at once, thus building a bridge which can be used for interoperability. Due to the overhead involved, the authors only recommend this approach for maintenance and reengineering tasks. Lastly, Zdun (2006) discusses five pattern of component and language integration, which are focused on component interactions between GPL, or GPL-like languages. These patterns are to be used in the technical implementation of frameworks which allow the use of multiple languages for a certain purpose.

On a meta level, we have used the approach by Nickerson et al. Nickerson et al (2013) as the basis for building our taxonomy. This approach is a recent addition to the taxonomy development literature which is in itself a hybrid of previous methods, including those imported from other disciplines (in particular, the social sciences). We believe that this integrative method represents the state of the art. The existing taxonomies linked above have used either a custom method defined in the publication itself (Tomassetti et al (2013)) or have not explicitly stated a method (Pfeiffer and Wasowski (2015)). Another option would have been the creation of a pattern language Alexander et al (1977) instead of a taxonomy. However, we believe that such a language is better suited for encoding best practices, while our taxonomy is an attempt at describing the existing state of the art.

6 Conclusion

This work has investigated the area of cross-language linking mechanisms across general-purpose and domain-specific languages.

Our contribution is a taxonomy of cross-language linking mechanisms with five dimensions, which are language types, GPL artifact type, declaration side, generation, and link configurability. Each dimension has 2-3 mutually exclusive characteristics. Our taxonomy has been carefully constructed using a 7-step approach based on an investigation of 22 well-known open-source frameworks, and focuses on the application developers perspective, which shows the choices that have been made and the options available to such developers in practice.

We have shown each characteristic of the taxonomy with examples from open-source frameworks, and have furthermore discussed the impact of each characteristic on application developers, which serves to raise awareness of the choices available and hopefully leads to a better appreciation of the benefits and problems. Where appropriate, we have pointed out the usefulness of the taxonomy to framework developers and researchers, too.

We believe that this taxonomy presents novel insights and a new viewpoint in the area of cross-language linking in practice by shedding light on how open-source frameworks differ in their linking specifications and the impact this has on application developers. We hope that this taxonomy will be used for further classifications in the future, thus adding more data to either confirm the taxonomies dimensions and characteristics, or extend them.

References

Alexander C, Ishikawa S, Silverstein M (1977) A Pattern Language: Towns, Buildings, Construction. Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series, OUP USA

Batory D (2005) Feature models, grammars, and propositional formulas. In: Proceedings of the 9th International Conference on Software Product Lines, Springer-Verlag, Berlin, Heidelberg, SPLC'05, pp 7–20, DOI 10.1007/11554844_3

Box D, Pattison T (2002) Essential .NET: The Common Language Runtime. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Burnette E (2009) Hello, Android: Introducing Google's Mobile Development Platform, 2nd edn. Pragmatic Bookshelf

Burtch K (2004) Linux Shell Scripting with Bash. Pearson Higher Education

man Chung K (2013) Java Server Pages Specification, Maintenance Release v2.3. URL `http://www.oracle.com/technetwork/java/javaee/jsp/index.html`

Delorey DP, Knutson CD, Giraud-Carrier C (2007) Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects. In: WoPDaSD 2007, Springer, pp 1–5

Ekman T, Mechlenborg P, Schultz UP (2007) Flexible language interoperability. Journal of Object Technology 6(8):95–116, DOI 10.5381/jot.2007.6.8.a2

Favre J, Lämmel R, Varanovich A (2012) Modeling the linguistic architecture of software products. In: France RB, Kazmeier J, Breu R, Atkinson C (eds) Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings, Springer, Lecture Notes in Computer Science, vol 7590, pp 151–167, DOI 10.1007/978-3-642-33666-9_11

jQuery Foundation T (2016) jQuery API Documentation. URL `http://api.jquery.com/`

Gordon R, Essential J (1998) Java native interface. Prentince Hall PTR

Gybels K, Wuyts R, Ducasse S, D'Hondt M (2006) Inter-language reflection: A conceptual model and its implementation. Computer Languages, Systems & Structures 32(2-3):109–124, DOI 10.1016/j.cl.2005.10.003

Johnson R, Hoeller J, Donald K, Sampaleanu C, Harrop R, Risberg T, Arendsen A, Davison D, Kopylenko D, Pollack M, et al (2004) The spring framework, reference documentation. URL `http://static.springframework.org/spring/docs/2.5x/reference`

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2016) An in-depth study of the promises and perils of mining github. Empirical Software Engineering 21(5):2035–2071, DOI 10.1007/s10664-015-9393-5

Lämmel R, Varanovich A (2014) Interpretation of linguistic architecture. In: Cabot J, Rubin J (eds) Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, York, UK, July 21-25, 2014. Proceedings, Springer, Lecture Notes in Computer Science, vol 8569, pp 67–82, DOI 10.1007/978-3-319-09195-2_5

de Mauro P (1999) Internationalizing messages in linux programs. Linux J 1999(59es)
Mayer P, Bauer A (2015) An empirical analysis of the utilization of multiple programming languages in open source projects. In: Lv J, Zhang HJ, Babar MA (eds) Proceedings of EASE 2015, Nanjing, China, April 27-29, 2015, ACM, pp 4:1–4:10, DOI doi:10.1145/2745802.2745805
Mayer P, Schroeder A (2014) Automated multi-language artifact binding and rename refactoring between java and dsls used by java frameworks. In: Jones R (ed) ECOOP 2014, Uppsala, Sweden, July 28 - August 1, 2014, Springer, Lecture Notes in Computer Science, vol 8586, pp 437–462, DOI doi:10.1007/978-3-662-44202-9_18
Mayer P, Kirsch M, Le MA (2015) On multi-language software development, cross-language links and accompanying tools: A survey of professional software developers. Tech. Rep. TR-2015-09-00, Institute for Informatics, Ludwig-Maximilians-Universität München, Oettingenstr 67, 80538 München, Germany, URL `http://www.pst.ifi.lmu.de/~mayer/papers/TR-2015-09-00.pdf`
Microsoft (2016) Windows Presentation Foundation. URL `https://msdn.microsoft.com/en-us/library/ms754130%28v=vs.110%29.aspx`
Nickerson RC, Varshney U, Muntermann J (2013) A method for taxonomy development and its application in information systems. EJIS 22(3):336–359, DOI 10.1057/ejis.2012.26
Odersky M, Altherr P, Cremet V, Emir B, Micheloud S, Mihaylov N, Schinz M, Stenman E, Zenger M (2004) The scala language specification
Oracle (2016) The Java Properties Format. URL `https://docs.oracle.com/javase/tutorial/essential/environment/properties.html`
Pfeiffer R, Wasowski A (2015) The design space of multi-language development environments. Software and System Modeling 14(1):383–411, DOI 10.1007/s10270-013-0376-y
Potencier F, Hason M, Blanc AL, Schultze T (2015) The twig template engine for php. URL `http://twig.sensiolabs.org/`
Rappin N (2008) Professional Ruby on Rails. Wrox Press Ltd., Birmingham, UK, UK
Ray B, Posnett D, Filkov V, Devanbu PT (2014) A large scale study of programming languages and code quality in github. In: Cheung S, Orso A, Storey MD (eds) Proceedings of the FSE-22, Hong Kong, China, November 16 - 22, 2014, ACM, pp 155–165, DOI 10.1145/2635868.2635922
Red Hat (2016) Hibernate Object-Relational Mapper. URL `http://hibernate.org/orm/`
Schafer S (2005) Web Standards Programmer's Reference: HTML, CSS, JavaScript, Perl, Python, and PHP. Wrox Press Ltd., Birmingham, UK, UK
Slee M, Reiss D, Agarwal A, Kwiatkowski M, Wang J, Piro C, Maurer B, Clark K, Luciani J, Duxbury B, Fernandez E, Lipcon T, McGeachie A, Molinaro A, Meier R, Farrell J, Geyer J, Yeksigian C, Abernethy R, Grochowski K (2015) The Apache Thrift Framework. URL `https://thrift.apache.org/`
SWIG Developers (2016) Simplified Wrapper and Interface Generator. URL `http://www.swig.org/`
The Apache Project (2016) Apache Wicket. URL `http://wicket.apache.org/`
Tomassetti F, Torchiano M (2014) An empirical assessment of polyglot-ism in github. In: Shepperd MJ, Hall T, Myrtveit I (eds) 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13-14, 2014, ACM, pp 17:1–17:4, DOI 10.1145/2601248.2601269
Tomassetti F, Torchiano M, Vetro A (2013) Classification of language interactions. In: Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on, pp 287–290
Völter M, Visser E (2010) Language extension and composition with language workbenches. In: Proceedings of OOPSLA 2010, ACM, New York, NY, USA, OOPSLA '10, pp 301–304
Yellin F, Lindholm T (1996) The java virtual machine specification. Addison-W esley
Zdun U (2004) Using split objects for maintenance and reengineering tasks. In: 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), 24-26 March 2004, Tampere, Finland, Proceedings, IEEE Computer Society, pp 105–114, DOI 10.1109/CSMR.2004.1281411
Zdun U (2006) Patterns of component and language integration. In: Manolescu D, Voelter M, Noble J (eds) Pattern Languages of Program Design 5, Addison-Wesley Professional, pp 357–397