A Tree-Based Approach to Support Refactoring in Multi-Language Software Applications

Hagen Schink, David Broneske*, Reimar Schröter*, and Wolfram Fenske*
University of Magdeburg, Germany
Email: hagen.schink@gmail.com, *{ david.broneske, reimar.schroeter, wolfram.fenske}@ovgu.de

***Abstract*—Developers build software applications using different programming languages, so they can benefit from the programming languages' specific advantages. To allow an interaction of different programming languages, each programming language offers Application Programming Interfaces (API) to be called. However, such interactions pose challenges for source-code refactoring across programming languages. To this end, we present a generalized approach to refactoring in multi-language software applications based on graphs of trees. To illustrate the broad application of our approach, we implement a library that builds the foundation for two tools that support the refactoring of database applications implemented in Java and Java applications that invoke code of the functional programming language Clojure.**

Keywords–refactoring; multi-language software application; Java, Clojure, Relational Database

I. INTRODUCTION

Programming languages provide different language constructs for the description of algorithms. Depending on the language constructs, a developer's effort to implement an algorithm may differ between programming languages. Hence, if a developer is able to choose another programming language for each problem in a single system, she can describe the solution with a minimum of effort. Consequently, developers use different programming languages in concert to implement software applications [1]–[8]. We call such a software application implemented by means of different programming languages a multi-language software application (MLSA) [5].

Irrespective of the programming language at hand, refactoring is a common technique to modify a source-code's structure while preserving the source-code's semantics [9]. Refactorings are used to improve the maintainability and extensibility of a code base. A number of refactoring transformations exist for different programming languages and programming paradigms, such as object-oriented programming-languages [9][10], functional programming-languages [11], and relational schemata [12].

However, refactoring transformations are defined for single programming languages and do not consider the interaction of languages in an MLSA. Thus, applying a refactoring on source code of one language can break the interaction of languages within an MLSA. For instance, in a database application, renaming a table breaks the application code that depends on the original table name [13]. Since compilers do not check language interaction at compile time, developers need a sufficient test coverage to detect the broken interaction or, otherwise, the broken system goes into production.

In this paper, we present a generally applicable concept based on graphs of trees that supports developers in checking and preserving language interaction within an MLSA. To show the practical applicability of our concept, we present two prototypes that support refactoring in applications that use (1) Java and a relational database and (2) Java and the functional programming language Clojure [14]. Additionally, we provide a brief discussion of the concept's performance and discuss the concept's generality in respect to different MLSA setups.

The paper is structured as follows: In Section II, we introduce different realizations of MLSAs and give examples of how refactoring affects language interaction in MLSAs. In Section III, we describe and justify our concept for supporting refactoring in MLSAs. In Section IV, we present two tools, sql-schema-comparer and clojure-java-interface-checker, which implement our concept for two different language combinations. We discuss different aspects of the concept in Section V. Finally, we present related work in Section VI before we conclude the paper in Section VII.

II. BACKGROUND

In this section, we first describe different approaches to implement language interaction in MSLAs. Then, we describe how refactoring can break language interaction in MLSAs by means of a database application.

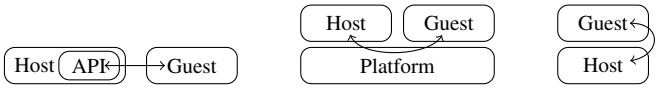
A. Implementations of Language Interaction in MLSAs

In general, a software application contains source code written in one programming language that initializes that application. In an MLSA, we call the programming language in which the application's initialization code is written the application's *host language*. From the code of the host language, developers invoke source code implemented in other languages. We call the invoked languages *guest languages*. Based on this definition, we distinguish three realizations of language interaction:

1) Foreign Function Interface (FFI) [15]
2) Host and guest language share the same platform
3) Guest language is implemented in the host language

The first realization, FFI, describes APIs, which allow developers to use host language syntax elements for accessing syntax elements in a guest language (cf. Figure 1a). For instance, in Java the Java Database Connectivity (JDBC) and the Java Persistence API (JPA) allow developers to query relational databases via SQL or an object-relational mapping, and the Java Native Interface (JNI) allows developers to invoke C/C++ functions. Platforms such as Java and .NET represent the second realization, in which both, host and guest language, share the same platform (see Figure 1b). For instance, the programming languages C#, Visual Basic .NET, and F# can all be compiled to the Common Intermediate Language (CIL) and

 ISBN: 978-1-61208-458-9



(a) FFI (b) Platform (c) Language

Figure 1. Relations between host and guest language.

all languages can invoke CIL code [16]. Thus, by compiling the source code of the guest language to the platform of the host language, a developer can call the code implemented in the guest language from the host language. The third realization (see Figure 1c) describes programming languages such as Lisp or Ruby, which provide meta-programming features that allow developers to implement new language elements or DSLs that represent the guest language [17]–[19].

In case a guest language is implemented in the host language (cf. Figure 1c), the guest language's source code is actually valid source code of the host language. Thus, developers can reuse existing tools for static code analysis of the host language to check the interaction between source code of the guest and the host language. In case the guest language's source code can be compiled to the host language's platform (cf. Figure 1b), developers can reference the compiled code of the guest language in the host language and the host language's compiler can check the interaction between source code of the guest and the host language [16]. For instance, F# source code can be compiled to a managed DLL which a developer can reference in C# source code. In contrast, if languages interact by means of an FFI (cf.Figure 1a), existing tools for the host language cannot check language interactions.

B. Refactoring of MLSAs that use FFIs

Since there is no tool support for refactoring languages that communicate using FFIs, developers have to assure language interaction manually. Currently, developers must provide automated tests with a reasonable test coverage to reliably detect broken language interactions after source-code modifications such as refactoring. In former work, we discussed problems in the case of Java to SQL interactions [13].

For the Java programming language, there are two APIs to interact with a relational database via SQL: JDBC and JPA. JDBC allows developers to directly execute SQL statements. For instance, Figure 2 shows an SQL query that selects the values of column `label` of table `departments` that start with *M*.

JPA, in turn, provides an interface for object-relational mapping (ORM). The ORM maps Java classes to relational tables and Java methods to table columns. An object relational mapper can automatically create SQL statements based on the ORM. Developers can use JPA to access a relational database without having to write plain SQL. For instance, Figure 3 shows how a developer can map the Java class `Department` to the relational table `departments` with its columns `id` and `label`.

In the following, we explain how refactoring either the Java source code or the relational schema affects the interaction between the Java application and the relational database.

```
1  String stmt = "SELECT label FROM departments "
2              + "WHERE label LIKE ?";
3  PreparedStatement query = con.prepareStatement(stmt);
4
5  query.setString(1, "M%");
6  ResultSet result = query.executeQuery();
```

Figure 2. JDBC: Parameterized query for department names.

```
1  @Entity
2  @Table(name="departments")
3  public class Department implements Serializable {
4
5    private int id;
6    private String label;
7
8    public void setId(int id) { this.id = id; }
9    @Id
10   public int getId() { return id; }
11
12   public void setLabel(String label) { this.label = label; }
13   public String getLabel() { return label; }
14 }
```

Figure 3. JPA: Annotated class `Department`.

a) Host-Language Refactoring: Given that we use JPA to access the relational database, we rename the property `label` of class `Department` (cf. Figure 3) to `name`, because `name` is more specific than the more general `label`. Consequently, to be consistent, we also rename the methods `setLabel` and `getLabel` to `setName` and `getName`, respectively. This refactoring breaks the application because the ORM cannot find a matching column `name` for table `departments` in the database schema.

b) Guest-Language Refactoring: Let us assume we apply a Rename Column refactoring to rename the column `label` of table `departments` to `name`. This refactoring breaks the application regardless of whether we use JDBC or JPA because neither the SQL statement (cf. Figure 2) nor the ORM (cf. Figure 3) reference the renamed column.

III. Checking and Preserving Language Interaction

The main idea of our concept to check and preserve language interactions is as follows. First, we extract those syntax elements from the host and guest language that are involved in language interaction and represent these syntax elements in graphs of trees for the host and the guest language. Second, by comparing the graph of the host and the graph of guest language with each other, we are able to detect broken language interactions. Now, we present this approach in detail.

A. Modeling Language Interaction

For the source code of a guest language C_G and the source code of a host language C_H, we call the references to the guest language's source code extracted from the host language's source code $R_{C_H \to C_G}$. Additionally, we call the syntax elements in the guest language's source code involved in language interaction R_{C_G}. Based on the representation as abstract syntax trees (AST) [20], we describe $R_{C_H \to C_G}$ and R_{C_G} as sets of

labeled trees. Consequently, $r_{C_H \to C_G} \in R_{C_H \to C_G}$ is a tree representing a single invocation of a guest language structure element in the host language and $r_{C_G} \in R_{C_G}$ is a tree representing a single structure element defined in the guest language that is involved in language interaction.

Since guest languages do not define a uniform set of syntax elements for language interaction, syntax elements for language interaction can be different between guest languages. Consequently, we cannot use a single type of tree to represent all syntax elements involved in language interaction, because we neither can oversee all current syntax elements, nor foresee all future elements involved in language interaction. Thus, we need to define specialized types of trees, which only represent the syntax elements in the guest language that are actually involved in language interaction. For instance, with JDBC or JPA, the table and column identifiers defined in the relational database are elements involved in language interaction. In contrast, with JNI, a function's name and parameters in C source code are elements involved in language interaction.

B. Checking the Referential Integrity between Languages

We check the referential integrity between languages by comparing all trees in $R_{C_H \to C_G}$ with the trees in R_{C_G}. To ensure the referential integrity, for all $r_{C_H \to C_G} \in R_{C_H \to C_G}$ there must be one $r_{C_G} \in R_{C_G}$, so that the following condition is satisfied:

$$r_{C_H \to C_G} \text{ is a top-down subtree of } r_{C_G} \quad (1)$$

However, this precondition is not sufficient because nodes may be missing in $r_{C_H \to C_G}$ that are mandatory for language interaction. For instance, in JDBC, if a developer defines an *INSERT* statement, this statement must provide values for all columns of the referenced tables with a Not Null constraint, or else the statement fails. Hence, for the set of mandatory nodes r_M, we additionally have to check if

$$r_M \subseteq r_{C_H \to C_G} \quad (2)$$

The set r_M is defined as follows for mandatory nodes m and the function $parent(x)$, which returns x's parent node:

$$r_M = \{m \mid m \in r_{C_G} \wedge parent(m) \in r_{C_H \to C_G}\} \quad (3)$$

In Figure 4, we illustrate the process of checking language interaction: First, we need to extract $R_{C_H \to C_G}$ from C_H and R_{C_G} from C_G. In accordance with (3), we compute the set of mandatory nodes R_M for R_{C_G} and $R_{C_H \to C_G}$. Then, we can compare the extracted references. The comparison returns a result which contains the possibly empty sets $R_{C_H \to C_G} \setminus R_{C_G}$ and $R_M \setminus R_{C_H \to C_G}$, i.e., the elements, which were extracted from the host language source code but are missing in the guest language source code, as well as the mandatory elements defined in the guest language source code that are not referenced in the host language source code. Language interaction is preserved if both sets are empty. Otherwise, the sets contain the syntax elements, which are involved in the broken language interaction.

IV. Implementing a General Approach to MLR

In this section, we first introduce the structure-graph library [21]. The structure-graph library implements an algorithm which we use to implement the sql-schema-comparer and the clojure-java-interface-checker. The sql-schema-comparer checks the language interaction between Java source code and a relational database schema and the clojure-java-interface-checker checks the language interaction between Java and Clojure source code.

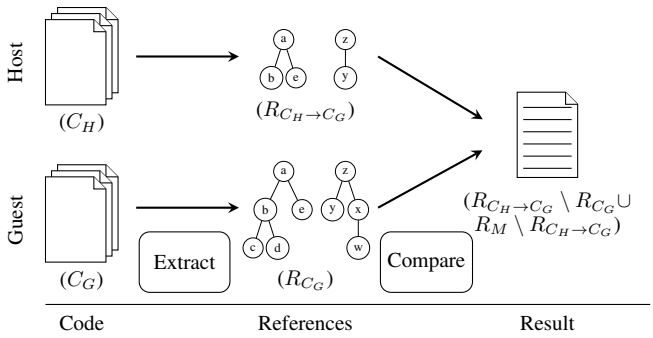


Figure 4. The process of checking language interaction.

A. The Structure-Graph Library

The structure-graph library provides a prototypical implementation of a comparison algorithm for trees of syntax elements as defined in Section III-A. The library takes two arguments: a source graph G_S and a target graph G_T. Both graphs represent sets of trees. Each node v of a tree has a *name* and a *path*. A node is uniquely identified by its name and path; thus, name and path represent the node's *id*. The comparison of G_S and G_T returns a list of modified nodes. The library distinguishes two node modifications: *Added* and *Deleted*. A node v is *added* if $id(v) \in G_T \wedge id(v) \notin G_S$. Conversely, a node v is *removed* if $id(v) \notin G_T \wedge id(v) \in G_S$. For checking language interaction, we pass $R_{C_H \to C_G}$ as G_S and R_{C_G} as G_T to the library. Consequently, $R_{C_H \to C_G} \setminus R_{C_G}$ is the set of added nodes and $R_M \setminus R_{C_H \to C_G}$ is the set of removed nodes that are marked as mandatory, respectively.

B. Java to Relational Database Interaction

In [22], we presented the sql-schema-comparer (SSC) library that applies the structure-graph library to detect mismatches between a relational database schema and a schema expected by the interacting Java source code. To this end, SSC extracts the expected schema $R_{C_H \to C_G}$ from SQL statements and JPA entities defined in the Java source code and the actual schema R_{C_G} from a relational database. The representation of R_{C_G} contains a tree for each table. Each tree contains a root that holds the table's name. The root has child nodes for each table column, and each column has child nodes for each column constraint (cf. Figure 5). Since SSC marks a column as mandatory if a Not Null constraint is specified on that column, SSC is able to check if all columns required for an insertion are referenced by an SQL statement or JPA entity. The representation of $R_{C_H \to C_G}$ contains a tree for each SQL statement and JPA entity defined in the interacting Java source code. The trees of the expected schema only contain nodes for the referenced table names and columns, because constraints are not referenced in the Java source code. For brevity, we do not discuss column type information here.

C. Java to Clojure Language Interaction

Now, we introduce the Clojure programming language and describe how the clojure-java-interface-checker preserves the interaction between Java and Clojure source code.

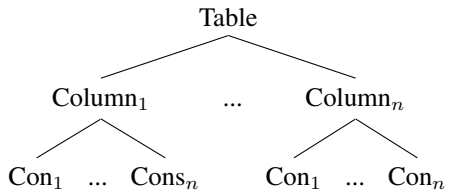


Figure 5. Database Schema Graph.

1) The Clojure Programming Language: Clojure is a functional programming language that runs on the Java Virtual Machine (JVM). Syntactically, Clojure is a Lisp dialect. Since Clojure runs on the JVM, developers can directly use libraries available for the programming language Java.

Since Clojure and Java share the same platform (cf. Figure 1b), developers do not need additional means to invoke Clojure source code from Java and vice versa. However, invoking Clojure functions in Java without additional means works only for compiled Clojure source code. Additionally, the Clojure library provides an FFI (cf. Figure 1a), which allows developers to dynamically load and invoke Clojure source code directly from Java source code.

For dynamically invoking Clojure functions from Java source code, developers must provide the namespace and the name of the function to be called. For instance, to call the function `add2` in namespace `i.o.c.Test` (see Figure 6), developers use the class `RT` shown in Figure 7. Since version 1.6, the preferred way of calling a Clojure function is to use class `Clojure` that returns an instance of class `IFn`. Nevertheless, the basic principle has not changed.

```
1 (ns o.i.c.Test)
2
3 (defn add2 [x]
4   (+ x 2))
```

Figure 6. Definition of a namespace and a function in Clojure.

```
1 Var f = RT.var("o.i.c.Test", "add2");
2
3 f.invoke(2);
```

Figure 7. Invocation of Clojure function in Java.

2) The Clojure-Java-Interface-Checker Library: We used the structure-graph library to implement the clojure-java-interface-checker [23]. The clojure-java-interface-checker is a library that checks the dynamic invocation of Clojure functions in Java source code. To this end, the library creates a graph for the function invocations in the Java source code and for the actual function definitions in the Clojure source code. The graph contains a tree for each namespace defined in the Clojure source code. Each namespace has a child node for each function defined in that namespace. Additionally, each node representing a function has a child node for each function parameter. Having a node for each parameter allows to check that the function invocation in the Java source code contains the correct number of parameters. Figure 8 shows the generalized structure of a tree for a Clojure namespace.

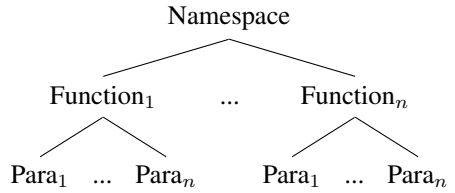


Figure 8. Clojure Function Graph.

For the function definition in Figure 6 and the function invocation in Figure 7, the library creates two different trees (see Figure 9): The only difference between the two trees is that in Java, we have no information about the called parameter but its position. Therefore, in Figure 9b the parameter `x` is represented by the parameter's position `0`. Accordingly, before we can compare these graphs, we need to replace the parameter name in Figure 9a by its position in the function definition.

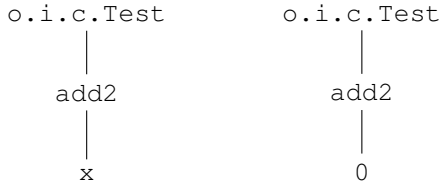
V. Discussion

In Section III, we discussed an algorithm for checking the interaction between a host and guest language. We presented two tools that show the practicability of our approach. In the following, we discuss the theoretical performance with respect to the complexity of checking the Conditions (1) and (2) from Section III-B and the generality of our approach. Furthermore, we describe the necessary implementation effort and other areas of application for our approach.

A. Performance

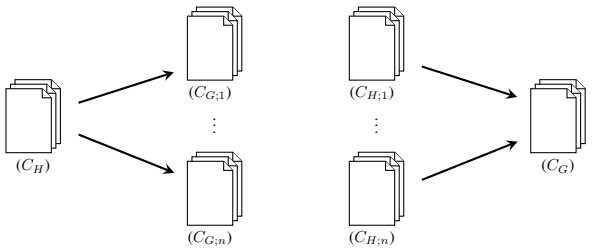
For Condition (1), we check that for each node in $R_{C_H \to C_G}$ a node exists in R_{C_G}. In a tree structure, each node has a unique path, thus, we need to check at most h_G nodes in R_{C_G} for each node in $R_{C_H \to C_G}$ where h_G is the height of R_{C_G}. Hence, we get each missing node in $O(n_H h_G)$ where n_H is the number of nodes in $R_{C_H \to C_G}$.

For Condition (2), we check that for each mandatory node in R_M a node exists in $R_{C_H \to C_G}$. Again, we need to check at most h_H nodes in $R_{C_H \to C_G}$ for each node in R_M where h_H is the height of $R_{C_H \to C_G}$. Hence, we get each missing mandatory node in $O(n_M h_H)$ where n_M is the number of nodes in R_M. Since the maximum height of the trees is constant, we get a complexity of $O(n_H + n_M)$ for checking all conditions.



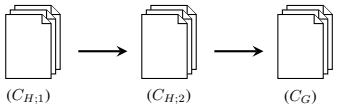
(a) R_{C_G} of Figure 6. (b) $R_{C_H \to C_G}$ of Figure 7.

Figure 9. Function graphs.



(a) Single host and multiple guest programming languages.

(b) Multiple host and single guest programming languages.



(c) A host is also a guest programming language.

Figure 10. MLSA setups.

B. Generality of the Approach

Until now, we discussed an MLSA consisting of one host and one guest programming language. However, MLSAs can consist of more than two programming languages. Thus, we need to discuss our approach in the context of generalized MLSA setups to justify the generality of our approach.

1) Multiple Guest and Single Host Programming Languages: The single host programming language uses different interfaces to interact with multiple guest languages (cf. Figure 10a). To check the referential integrity of the language interaction, we need to compute and compare $R_{C_{G;n}}$ and $R_{C_H \to C_{G;n}}$ with n being the nth guest language of the MLSA.

2) Single Guest and Multiple Host Programming Languages: The host programming languages use the same elements of the guest programming language for language interaction (cf. Figure 10b). Thus, to check the referential integrity of the language interaction, we need to compute and compare $R_{C_{H;n} \to C_G}$ and R_{C_G} with n being the nth host language of the MLSA.

3) A Host is also a Guest Programming Language: Given code of three programming languages C_G, $C_{H;1}$, $C_{H;2}$ involved in an MLSA where C_G is accessed by $C_{H;2}$ and $C_{H;2}$ is accessed by $C_{H;1}$ (cf. Figure 10c). Hence, this MLSA contains code of two guest programming languages (C_G and $C_{H;2}$) and code of two host programming languages ($C_{H;1}$ and $C_{H;2}$). In other words, we have multiple guest and multiple host programming languages. However, having multiple guest and multiple host programming languages corresponds to a combination of the preceding cases.

Apart from the three MLSA setups shown in Figure 10a to 10c, to the best of our knowledge, no other generalized MLSA setup exists. Since our approach supports all three MLSA setups, we conclude that our approach supports the refactoring of arbitrary MLSAs.

C. Implementation Effort

Our approach requires developers to provide language-specific components: parsers and the evaluation logic. Parsers extract the source code of the interacting host and guest languages and create the graphs for $R_{C_{G;n}}$ and $R_{C_H \to C_{G;n}}$. For the creation of graphs, SSC includes the Java graph library JGraphT [24]. The evaluation logic interprets the results of SSC and gives language-specific feedback to the user.

D. Fields of Application

Apart from refactoring, another use case for our approach is content assistance. That is, we can use R_{C_G} to provide developers with a list of elements with which the developers can interact with. However, especially in respect to legacy or undocumented source code, we have to note that our approach does not consider the guest language's semantics. Thus, developers still need to know the behavior of C_G.

VI. Related Work

TexMo [25] and XLL [7][26] are tools for linking and refactoring MLSAs. TexMo uses GenDeMoG [27] which implements a dependency graph and XLL implements a linking model to link artifacts of interacting languages. Links are resolved by dependency patterns in GenDeMoG and binding resolvers in XLL. Thus, in TexMo and XLL, the artifacts involved in language interaction between two languages are hidden in dependency patterns or binding resolvers. However, based on our experience with other language combinations [13], we developed our graph-based approach that makes the structures involved in language interaction transparent. We argue that this transparency is crucial because language interaction is not only affected by renames of interacting elements. Yet, TexMo and XLL solely support rename refactorings.

Language composition comprises approaches to extend a programming language's syntax and semantics [28]. For instance, *SugarJ* [29] and *TSL Wyvern* [30] allow developers to embed different languages as first-class citizens in Java and Wyvern [31], respectively. As a first-class citizen, an embedded language's syntax is validated at compile-time. Compile-time validation allows developers to fix syntax errors in the code of the embedded language before run-time. Our approach complements language composition because, additionally, it allows to check the interaction introduced by embedded languages. For instance, embedding SQL by language composition can only ensure syntactical correctness, but not that SQL statements reference elements that are available in the database schema. Thus, language composition cannot ensure language interaction in general.

UMLDiff detects differences between two UML class models and reports added, removed, renamed, and moved UML elements [32]. UMLDiff is part of the Eclipse plug-in *JDEvAn* [33]. JDEvAn allows to retrieve UML representations from Java source code but is extensible to retrieve UML representations from other languages. JDEvAn and UMLDiff may be used to retrieve information about the changes that led to a broken language interaction and, thus, complement our check of the referential integrity for language interaction.

Orthographic Software Modeling (OSM) introduces views as first-class entities in software development [34]. In OSM, all information about a software application is represented in a single underlying model (SUM). All other models, such as UML or source code, are generated from the SUM. Since all changes to a view must be propagated to the SUM, all views are kept consistent automatically. For instance, in a database application using JPA, changing the relational model results in the adaption of the relational schema as well as the

 ISBN: 978-1-61208-458-9

JPA entities. However, the OSM approach requires a SUM and a developer, called *methodologist*, who implements and maintains the SUM. In general, we cannot presume these requirements to be fulfilled. Furthermore, the methodologist needs to model language interaction in the SUM, which requires intimate knowledge of implementation details.

VII. Conclusion and Future Work

Language interaction in a multi-language software application (MLSA) can be diverse and, thus, complicates the refactoring of involved languages. In our approach, we use graphs of trees to represent syntax elements of different programming languages that are involved in language interaction. Based on these graphs, we can check if the source code of one language correctly interacts with the source code of another language.

Based on our approach, we presented two tools, sql-schema-comparer and clojure-java-interface-checker, which check the interaction within a database application and between source code of the programming languages Java and Clojure, respectively. We presented performance considerations which suggest that graphs of trees are a viable basis for checking the interaction of source code of different languages. We also discussed transferability of our approach to arbitrary MLSA setups.

In our future work, we want to integrate the sql-schema-comparer and clojure-java-interface-verifier in the Eclipse IDE [35] to simplify their usage for daily software development. Furthermore, we want to re-use graphs of trees extracted from the guest language for detecting source-code modifications, such as refactorings, that led to a broken language interaction. We assume that the information about source-code modifications can support developers in fixing language interaction.

Acknowledgments

The work of Reimar Schröter is funded by BMBF, grant number 01IS14017B.

References

[1] B. Kullbach, A. Winter, P. Dahm, and J. Ebert, "Program Comprehension in Multi-Language Systems," Working Conference on Reverse Engineering, 1998, pp. 135–143.

[2] T. C. Jones, Estimating Software Costs. Hightstown, NJ, USA: McGraw-Hill, Inc., 1998.

[3] M. Grechanik, D. Batory, and D. E. Perry, "Design of Large-Scale Polylingual Systems," International Conference on Software Engineering, 2004, pp. 357–366.

[4] D. Strein, H. Kratz, and W. Lowe, "Cross-Language Program Analysis and Refactoring," IEEE International Workshop on Source Code Analysis and Manipulation, 2006, pp. 207–216.

[5] P. K. Linos, W. Lucas, S. Myers, and E. Maier, "A Metrics Tool for Multi-Language Software," International Conference on Software Engineering and Applications, 2006, pp. 324–329.

[6] N. Chen and R. Johnson, "Toward Refactoring in a Polyglot World: Extending Automated Refactoring Support across Java and XML," Workshop on Refactoring Tools, 2008, pp. 1–4.

[7] P. Mayer and A. Schroeder, "Cross-Language Code Analysis and Refactoring," International Working Conference on Source Code Analysis and Manipulation, 2012, pp. 94–103.

[8] N. Ford, The Productive Programmer. O'Reilly, 2008.

[9] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, USA, 1992.

[10] M. Fowler, Refactoring: Improving the Design of existing Code. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] H. Li and S. Thompson, "Tool Support for Refactoring Functional Programs," ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, 2008, pp. 199–203.

[12] S. Ambler, Agile Database Techniques: Effective Strategies for the Agile Software Developer. John Wiley & Sons, Inc., 2003.

[13] H. Schink, M. Kuhlemann, G. Saake, and R. Lämmel, "Hurdles in Multi-Language Refactoring of Hibernate Applications," International Conference on Software and Database Technologies, 2011, pp. 129–134.

[14] Clojure homepage. [Online]. Available: http://www.clojure.org/ (visited on Dec. 3, 2015).

[15] M. Furr and J. S. Foster, "Checking Type Safety of Foreign Function Calls," ACM Transactions on Programming Languages and Systems, vol. 30, no. 4, 2008, pp. 1–63.

[16] J. Hamilton, "Language Integration in the Common Language Runtime," ACM SIGPLAN Notices, vol. 38, no. 2, 2003, p. 19.

[17] M. Mernik, J. Heering, and A. M. Sloane, "When and how to Develop Domain-Specific Languages," ACM Computing Surveys, vol. 37, no. 4, 2005, pp. 316–344.

[18] L. Tratt, "Compile-Time Meta-Programming in a Dynamically Typed OO Language," Symposium on Dynamic Languages, 2005, pp. 49–63.

[19] S. Günther, "Multi-DSL Applications with Ruby," IEEE Software, vol. 27, no. 5, 2010, pp. 25–30.

[20] J. Jones, "Abstract Syntax Tree Implementation Idioms," Conference on Pattern Languages of Programs, 2003, pp. 1–10.

[21] Structure graph. [Online]. Available: https://github.com/hschink/structure-graph (visited on Dec. 3, 2015).

[22] H. Schink, "sql-schema-comparer: Support of Multi-Language Refactoring with Relational Databases," International Working Conference on Source Code Analysis and Manipulation, 2013, pp. 164–169.

[23] Clojure java interface checker. [Online]. Available: https://github.com/hschink/clojure-java-interface-checker (visited on Dec. 3, 2015).

[24] Jgrapht. [Online]. Available: http://jgrapht.org/ (visited on Dec. 3, 2015).

[25] R. H. Pfeiffer and A. Wąsowski, "TexMo: A Multi-Language Development Environment," European Conference Modelling Foundations and Applications, 2012, pp. 178–193.

[26] P. Mayer and A. Schroeder, "Automated Multi-Language Artifact Binding and Rename Refactoring between Java and DSLs used by Java Frameworks," European Conference Object-Oriented Programming, 2014, pp. 437–462.

[27] R. H. Pfeiffer and A. Wąsowski, "Taming the Confusion of Languages," European Conference on Modelling Foundations and Applications, 2011, pp. 312–328.

[28] S. Erdweg, P. G. Giarrusso, and T. Rendel, "Language Composition Untangled," Workshop on Language Descriptions, Tools, and Applications, 2012, pp. 1–8.

[29] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann, "SugarJ: Library-Based Syntactic Language Extensibility," ACM SIGPLAN Notices, 2011, pp. 391–406.

[30] C. Omar, D. Kurilova, L. Nistor, and B. Chung, "Safely Composable Type-Specific Languages," European Conference on Object-Oriented Programming, 2014, pp. 105–130.

[31] L. Nistor, D. Kurilova, and S. Balzer, "Wyvern: A Simple, Typed, and Pure Object-Oriented Language," Workshop on MechAnisms for SPEcialization, Generalization and inHerItance, 2013, pp. 9–16.

[32] Z. Xing and E. Stroulia, "UMLDiff: An Algorithm for Object-Oriented Design Differencing," IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 54–65.

[33] Z. Xing and E. Stroulia, "The JDEvAn Tool Suite in Support of Object-Oriented Evolutionary Development," Companion of the International Conference on Software Engineering, 2008, p. 951.

[34] C. Atkinson, D. Stoll, and P. Bostan, "Orthographic Software Modeling: A Practical Approach to View-Based Development," in Evaluation of Novel Approaches to Software Engineering, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2010, vol. 69, pp. 206–219.

[35] Eclipse. [Online]. Available: http://www.eclipse.org/ (visited on Dec. 3, 2015).

 ISBN: 978-1-61208-458-9