A Model-Based Approach to Language Integration

Federico Tomassetti*, Antonio Vetro'*, Marco Torchiano*, Markus Voelter† and Bernd Kolb‡
*Department of Computer Engineering and Control Automation
Politecnico di Torino, Torino, Italy
Email: [federico.tomassetti|antonio.vetro|marco.torchiano]@polito.it
†independent/itemis,
Email: voelter@acm.org,
‡itemis,
Email: kolb@itemis.de

***Abstract*—The interactions of several languages within a software system pose a number of problems. There is several anecdotal and empirical evidence supporting such concerns. This paper presents a solution to achieve proper language integration in the context of language workbenches and with limited effort. A simple example is presented to show how cross-language constraints can be addressed and the quality of the support attainable, which covers error-checking and refactoring. A research agenda is then presented, to support future work in the area of language integration, taking advantage of modern language workbenches features.**

***Index Terms*—language interactions; language integration; model driven development; projectional editors.**

I. Introduction

Multi-language systems development represents one of the crucial challenges in model software development [1]. In fact nowadays not only the size of software systems makes them complex, but also the large number of artifacts and the coexistence of distinct though interacting languages. As a matter of fact, the top 50 projects among the most active ones indexed by the *Ohloh* OSS directory[1] are composed, on average, by 16 distinct languages, ranging from a minimum of 3 (openSSH) to a maximum of 71 (Debian GNU/Linux).

The possibility of having different languages that interact and cooperate to deliver software functionalities adds flexibility and capabilities to software development. In fact the limitations of a language can be compensated with the capabilities offered by others. However, interaction of languages without proper integration and tool support might be a source for problems. As a matter of fact, we should consider that current tools typically check only the consistency within a set of artifacts written in the same language. For example, editors check that the methods invoked by an expression in Java code actually exist in the codebase, either in the same file or in another Java file. However, they are not able to control whether a piece of XML code used for configuration refers to a really existing Java class, because they are not aware of the cross-language semantics.

Problems due to language interactions have been in some cases addressed with ad-hoc solutions involving the development of specific supporting tools or plugins for different development platforms. This is the case, for instance, of the Spring tool suite which consists of a series of plugins for Eclipse offering support for the cross-references between Spring configuration files and Java code. Similar plugins are also available to handle references between Android XML configuration files and Java code.

However, in general verifying the consistency across the language boundaries is not possible because tools are not aware of the cross-language semantics.

These ad-hoc approaches are a symptom of the need for assuring some level of consistency of the global system, also across language boundaries. The major limit of those approaches is that they address the problem of supporting cross language references in an ad-hoc way, i.e. for a particular relation involving a specific pair of neighbor languages, which may result expensive and incomplete.

We advocate a mechanisms that offers tool support without involving the creation of specific editors for any peculiar use of a language, e.g. using XML for configuring a specific aspect of a given framework.

This paper first shows the relevance of these issues (section II), then provides some evidence concerning the problem deriving from language interactions (section III). After that it outlines a possible solution (section IV) and describes related work (section V). Eventually research agenda is also provided to guide future works on this topic (section VI).

II. Prevalence of Language Interactions

Some of the authors of this paper recently conducted an investigation on language interactions. They carried on a case study [2] on the Hadoop project, to understand the magnitude of the phenomenon and identify possible implications. We started our investigation observing the commits in the version repository of Hadoop, driven by the following approximation: if a commit concerns files of different languages we assume that those files are related. For instance, considering a commit that fixes a bug and contains an HTML file and a CSS file: probably both files were changed in order to fix the bug.

We called cross language commits (CLC) those commits containing files of different languages. In particular we define *Cross Language Ratio* (CLR) as the ratio of CLC among all commits. In addition to the project CLR, we tried to

[1] http://www.ohloh.net/

TABLE I
MOST INTERACTING LANGUAGES IN A SAMPLE OF FIVE APACHE PROJECTS

File Extension	Hadoop	Derby	Forrest	Harmony
All	53%	64%	66%	77%
C	96%	-	-	91%
sh	87%	92%	71%	74%
properties	72%	91%	77%	91%
XML	71%	88%	72%	94%
java	59%	62%	64%	77%
xsl	84%	100%	82%	99%
HTML	100%	87%	91%	99%

understand which were the most interacting languages by measuring the CLR for each language, that is computed considering only the commits involving that language.

The rationale of computing this metric is to get a proxy of the level of interaction of files written in a given language. Assuming the bi-univocal correspondence language-extension, with this proxy it is possible to understand for each extension whether the majority of its files interact with files of other extensions.

In the Hadoop repository we observed CLR=53%. i.e. 53 out 100 commits in the repository were cross language. Further analyses (not yet published) on three more Apache projects (Derby, Forrest, Harmony) confirmed the initial observation: they show CLR respectively 64%, 66% and 77%.

Focusing on the specific languages we showed that the most interacting languages (among the ones with more files) of the Hadoop projects were C, sh, XML, Java and, considering also non programming languages, .properties files. CLR ranged from 59% (Java) to 96%(C) and even 100% (HTML, but with a lower number of files than C). Table I shows the CLR computed for common extensions in Hadoop and in three more projects, Derby, Forrest and Harmony: with the exclusion of C and sh, we observe very similar or higher CLR in the other projects. For Derby the most interacting extension is xsl, in Forrest is HTML while in Harmony both xsl and HTML.

These figures confirm the observation that the different languages used in software projects are not sealed off from each other, but they interact. In the next section we will discuss the problematic implications of such interactions, providing both anecdotal and empirical evidence.

III. PROBLEMS GIVEN BY LANGUAGE INTERACTIONS

Combining different languages inside one system can lead to possible inconsistencies across the language boundaries.

In order to prove that interacting languages can be a source of problems in software projects, we are going to briefly provide and discuss examples of well-known cases (III-A), and to summarize the empirical evidence found in the previous work and in followup analyses (III-B).

A. Anectodal evidence

In the literature, as well as in every developer's experience, there are several common examples of language interactions and related problems. Here we report a few specimens.

a) Web applications: Web applications are developed using a plethora of languages. A typical web application uses a general purpose language for server-side processing, SQL to access the database, some template language (e.g., JSP or Facelets[2]) to generate the pages, HTML in the form of entire pages or snippets to be combined, Javascript for client-side elaborations, and CSS to control the appearance of the page.

All these languages cooperate to produce a working software system. References between artifacts written in different languages are therefore very common. Let us consider an HTML tag (e.g. `<input>`), with a specific *class* or an *id*. It can be referred to by:

- **Javascript:** for example, a function written in Javascript could verify the correctness of the input inserted by the user, if the tag is a field, or it can be used to react to some event generated by the user,
- **Server side language:** it could need to process the result of a submit including the value of that tag, if it is a field,
- **Template language:** it could need to generate javascript which refers to that particular id, or a css configuring that particular class,
- **CSS:** a CSS rule can be written to customize the appearance of a tag with that id or class.

A simple typing error in the name of the class or the id of the tag would be unnoticed by CSS (the rule will just not apply to the element), Javascript would tend to fail silently while it could cause a run-time failure on the server side.

b) XML configuration of Java applications: Java applications rely frequently on configuration written in XML files, for example to configure Dependency Injection[3] or a Web Application framework[4].

These configuration files are used to achieve flexibility in the application. They commonly drive the instantiation and binding of classes, therefore they contain references to Java classes, expressed as strings corresponding to the name of the classes referred. The referred classes are expected to be present in the system, to extend a particular class, and to implement a given interface or possess some methods: for example a getter or setter for a particular property. When these conditions are not met the system can incur in a run-time error. There could be more complex constraints between the different elements referred, for example if a class is referred in some point, it should be initialized in another section of the file.

c) C and the preprocessor: Most of the files written in C (as well as files written in Objective-C or C++) host directives which are interpreted by the preprocessor and are expressed in its own peculiar language, which operates at the token level. The preprocessor directives constitute a language *per se* that

[2]http://facelets.java.net/

[3]See for example Spring, http://www.springsource.org/ .

[4]See for example Apache Struts, http://struts.apache.org/ .

could be used also in different contexts, i.e. outside C files. The interactions between these two languages – C and the language of the preprocessor – are source of different types of errors, which can be difficult to find and are detected only under particular configurations [3]. In this case a language is embedded into the other. This is not the only case, consider for example the usage of SQL in different host languages.

For example a macro could obfuscate a C type or a C identifier, it could assume a value which makes the compilation fails or it can cause more subtle errors. Consider the example presented in Listing 1: here a function-like macro is used to calculate the square of a given number. It works as expected when it receives a number literal, but when it receives an expression having side-effects (like `a++`), it calculates the incorrect result.

```
#ifdef INLINE
#define square(x) ((x)*(x))
#else
#define square _square
#endif

int foo(int a){
  return square(a++);
}
```

Listing 1. An example of problematic language interaction between C and the preprocessor

It is worth noting that while the preprocessor is frequently associated with C, it is completely unaware of the C semantics (except for the concept of comments and literals, which should be recognized to correctly identify preprocessor directives).

Typically the references across language boundaries are implemented by using a common identifier. Other possible ways of combining languages are described by Völter [4] (at least for DSLs).

Unfortunately tools supporting any language are unaware of the type, the characteristics or even the existence of elements with that particular identifier among artifacts written in another language. Therefore the coherence of the whole system depends on human code inspections or verifications at runtime, when a failure, if noticed, can start an investigation of the problem.

This happens because the rules controlling language interactions are not explicitly formalized. Moreover typically modern IDEs support different languages through independent editors, which are hosted on a common platform (e.g., Eclipse). What is missing is a shared meta-model, permitting to express cross language concerns, and an environment supporting the expression of these concepts, and the enforcement of this constraints. Language workbenches offer that.

B. Empirical evidence: side effects of languages interaction

In the previous preliminary work mentioned in Section I [2], some of the authors of this paper have examined the role of language interactions on defect proneness. Starting from the definition of CLR (i.e., the ratio of cross language commits in all commits), they classified modules of Hadoop in Cross Language Modules (CLM), i.e. files with CLR $\geq$ 50%, and Intra Language Modules (ILM), i.e. files with CLR $<$ 50%.

Considering the five most interacting extensions in Hadoop, the authors observed that three extensions (XML, Properties and C) had CLM with statistically significant higher defect proneness, while two extensions (Java and sh) exhibit the opposite relation. Breaking down the analysis on specific pairs of extensions, we observed that:

- four extension pairs had CLM more defect prone then ILM (C-Java, C-XML, Properties-C, sh-C);
- five extension pairs had ILM more defect prone then CLM (C-Properties, C-sh, Java-XML, Properties-XML, XML-Java);
- one extension pair had exactly same defect proneness (Properties-Java).

Subsequent analyses on Forrest and Harmony projects revealed that Java (in Forrest), cpp and XML (in Harmony) were the languages whose Cross Language Modules were more defect prone than Intra Language Modules.

Although these observations do not provide univocal answers, they support the theory that particular interactions between languages could be problematic. We will discuss in the next section our proposed solution.

IV. Language Integration in Language Workbenches

Herein we present a preliminary approach to obtain seamless language integration with full tool support in the context of language workbenches. Our reference implementation uses the Jetbrains MetaProgramming System[5] (MPS) but it is not limited to it: it could be implemented also for the Eclipse Modeling Platform [5] or other Language workbenches (e.g., Spoofax [6]) as long as the Language Workbench considered supports the languages of interest.

MPS is a projectional editor: it means that the abstract information underlying the model (something similar to the Abstract Syntax Tree) is persisted independently from the concrete syntax of the language. This is radically different from what happens with text editors. There are many benefits with this choice, but a very important one is that no parsing is necessary. In this way languages can be freely evolved and combined without the risk of obtaining an ambiguous grammar. The models are then projected, hence they are represented in a form suitable for understanding and editing by the user. Typically these projections are textual but they could also be graphical. Different artifacts can be later generated from the models: for example compiled java classes or the concrete XML files to be distributed within the compiled system.

We chose MPS for the completeness of the tool and because some of the authors acquired experience with this environment in the context of the mbeddr project[6]. Moreover MPS is distributed with language plugins which permit to to operate with Java and XML out of the box.

[5]http://www.jetbrains.com/mps/
[6]http://mbeddr.com

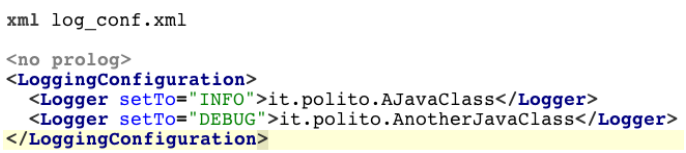

Fig. 1. Editing the XML configuration file inside MPS without any extension.

```
<LoggingConfiguration>
  <Logger setTo="INFO">it.polito.AJavaClass</Logger>
  <Logger setTo="DEBUG">it.polito.AnotherJavaClass</Logger>
</LoggingConfiguration>
```

Fig. 2. The XML file generated opned in a text editor.

For the sake of simplicity and because of space constraints, we will present our approach using a working example of language interactions involving the most common pattern: references across two different languages.

Let's consider a simple logging framework which obtains the configuration from an XML file. The configuration file specifies for each class the level of verbosity of the associated instance of the logger. This mechanism is for example used by Log4J[7].

We start by creating some Java classes and an XML model for the configuration of the framework. Both Java classes and the XML model are edited inside MPS.

MPS is capable, without any extension, of editing an XML model and inserting the name of the Java class we want to configure as simple text, as shown in Figure 1. The XML file generated is shown in Figure 2.

This system is brittle: if the user inserts a typo, or the referred class is deleted, renamed or moved to another package the system will incur in an error which the IDE is not able to detect. Moreover the user has to type long class names (including the name of the package, to be univocal), which is both error-prone and time consuming.

Our approach consists in creating particular elements to hold references from one language inside artefacts of other languages. In this case we created a particular element which permits to represent at the appropriate semantic level a reference to a Java class inside an XML document. Using the terminology of MPS, we created a new Concept named *JavaClassRefAsXmlContent*. This concept:

- extends *XmlContent* (which represents the content of XML tags). In this way it is possible to insert instances of JavaClassRefAsXmlContent in all the places where instances of XmlContent are allowed,
- has a reference to the Concept *ClassConcept* (which represents Java classes). We named this reference "class",
- has specific scoping rules for the reference "class",
- when generation is invoked, has its instances substituted by the full name of the referred class. The full name is obtained concatenating the name of the package containing the class with the name of the class itself.

[7]http://logging.apache.org/log4j/

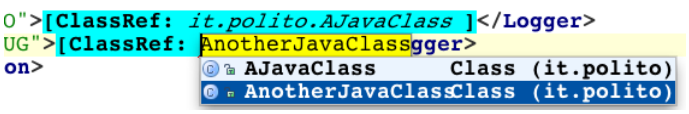

Fig. 3. The system showing autocompletion.

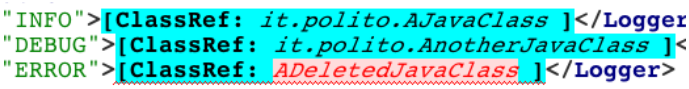

Fig. 4. The system showing a broken reference.

The implementation of this mechanism required only a few minutes.

MPS provides automatically autocompletion for the new Concept, considering the scoping rules we specified. The result obtained is visible in Figure 3. This mechanism in addition to offer autocompletion out of the box (therefore saving the user from writing long names, an error-prone activity), guarantees automatically consistency: renaming or moving a class the reference is automatically updated and the correct value is inserted during the generation phase. If the class is deleted the reference is recognized to be broken and the editor presents an error message, as shown in Figure 4. It is possible also to navigate the reference, hence from the XML file it is possible to click on the name of the referred Java class and open in in the editor.

This simple implementation offers a description of the level of language integration which is possible to achieve inside language workbenches with a very small effort.

While the simple example we presented deal with cross-language references, we plan to investigate also other kinds of interactions in the future.

Other mechanisms like the use of annotations of the use of custom persistence are also possible: they are not discussed in this article but they represent a future work.

V. Related Work

In the literature for language integration it is possible to identify two main threads: (i) approaches applicable to language families which permit to have a strong control on the definition of the languages and (ii) approaches with more general applicability.

A. Approaches working on family of languages

Upon observing that the the amalgam of languages used in a single web application project are typically poorly integrated [7], Groenewegen et al. propose the adoption of an unique language to model all the different concerns of web applications: WebDSL. They discuss the integration of an access control policy inside WebDSL. They propose to express these policies separately and then weave them inside WebDSL. In this way WebDSL remains unaware of the access control policies, while the language used to describe access control policies is created embedding knowledge of WebDSL. From this prospective we could consider this language as being part of the "family"

of WebDSL. The approach of creating family of DSLs with built-in language integration is common (another example is Epsilon [8]). Language integration across languages of the same family, built with this particular goal in mind, is easier, w.r.t. integration among two languages not created specifically to be integrated.

Barja et al. [9] present a system based on the integration of a logic query language with an imperative programming language in the context of an object-oriented data model. They first discuss various possible approaches for the integration of the two languages and the implementation of one of them. Also in case the languages were already developed with the goal of language integration, they therefore constitute a family of DSLs.

Tolvanen et al. [10] describe their experience in integrating Domain Specific Modeling (DSM) languages. They do not consider integration with general purpose languages because they intend to take advantage from the fact that companies have full control of the individual DSM languages developed for internal used and how they can be integrated. This is fundamentally different from the general case of having to integrate arbitrary languages, without the possibility of modifying their definition. They discuss the case of integration based on string matching and the possibility of direct reference. They consider the possibility to combine both approaches. The second one relies on the particular technology used to realize the DSM, the MetaEdit+ system[8].

B. General approaches

Mayer et Schroeder [11] name the problems of references across artifacts written in different languages as "semantic cross-language links". Being these links out of scope of the individual programming language, they are ignored by most language-specific tools and are often checked only at runtime. They propose to express explicitly constraints for these links and present three possible approaches to do that: at the source code level, using language-specific meta-models, and using language-spanning meta-models. Of these approaches they chose the second, while we advocate the third, which permits to reuse a common API and, in the case of language workbench, is already available without the necessity of developing it. Their approach is named XLL and it permits to automatically identify semantic cross-language links, correct or broken, and support the user in the activities of program understanding, analysis and refactoring. XLL requires to develop for each language considered:

- a meta-model specific for each language considered. The meta-model should contain all the information needed to individuate possible link instances, i.e. it should not necessarily represent the entire language.
- a mechanism to list the artifacts of the languages and instantiate for each of them a model pertaining to the language specific meta-model,
- an adapter to locate bindings to the refactoring capabilities of the IDE where the implementation of the approach is realized.

The approach requires then to describe possible type of links, describing for each link the nature of the element involved in the link.

Using information obtained from models of the artifacts and description of the types of links, the system is able to calculate successful and unsuccessful links. Continuous recalculation of the state of links is performed in background. The authors present a protypal implementation for the Eclipse platform and discuss two kinds of relations. The main advantage of this approach, is the possibility to be implemented as a plugin for the IDE of choice. On the other hand this approach requires a considerable effort to develop language specific adapters, and it is limited in refactoring capabilities. The authors state that their mechanism of refactoring: i) could fail under certain condition, ii) it is limited by the availability of refactoring bindings in the IDE of choice (which the authors report to be available for Java and in some form for Ruby, on the platform considered for the implementation), and iii) consider only the rename refactoring. Moreover it is not able to capture manual name changing which are not performed through explicit refactorings. Navigability seems to be limited to the artifict, not to the specific element involved in the link, which our approach permits. Both ours and their approaches support program analysis, but while their require the development of language specific artifacts, our require almost a null effort and provide a far better tool support in respect to navigability and refactoring.

Pfeiffer realized a system called TexMo [12] which permits to express references between artifacts written in different languages, but not to express other kind of constraints. It is realized as an Eclipse plugin and it is intended to be used instead of the original editors provided inside Eclipse. It uses a syntactic universal representation of all the languages supported. I.e., for each language has to be provided an adapter generating models (instances of the universal metamodel) from the concrete artifacts (e.g., java or xml files). Our approach do not require to recreate editors but instead permit to simply enrich the industrial-strength editors already available in MPS. It does not resoirt on a limited universal metamodel, but instead use the MPS representation of the language, which permits to consider every aspect of the language.

Pfeiffer et al. [13] used TexMo in a controlled experiments with 22 subjects to demonstrate the effects of tool support for cross-language references. They provided to the subjects two different instances of TexMo: one with the support for cross-language references enabled and one with it disabled. Results show a significative improvemente in the ability to correctly locate the source of errors (which could be unnoticed or lead to run-time failure, when such tools support is not available). An important result is in the different way errors are located: while developers having tool-support for cross-language references locate correctly the source of errors (i.e., the broken reference), other developers barely find the effect of error, but are not able

[8]MetaEdit+ Workbench 4.5 SR1 User's Guide, http://www.metacase.com/support/45/manuals/

to understand the reason of the error, at least not in the short time allotted for each task (10 minutes).

VI. Conclusions and Research Agenda

Almost every non trivial system is developed using a set of languages. While using the proper language for each task helps the productivity, the side effect is to incur in problematic language interactions which can lead to inconsistency and errors which are not recognized at development time and cause errors during the execution.

We are convinced that Language workbenches offer a solution to these problems, making possible to specify and enforce constraints. Tools can be built with a minimal effort, which provide a complete support and help developing consistent systems.

While this approach seems very promising there are different aspects which require more work. Therefore we would like to conclude this paper presenting a list of aspects which we think deserve further investigation.

Empirical assessment of the problems related to language interactions: to the best of our knowledge, we are not aware of any empirical study considering the effects of languages interactions, with the exclusion of a work [2] from some of the authors of this paper. The metrics provided in that preliminary study however need further validation: in breadth – more projects should be analyzed – and in depth – careful analysis of the detected interaction–. Probably better proxies to measure the level of interaction of a languages are needed. In addition to that, a more qualitative analysis on the projects already analyzed might offer both a validation of the metrics and the creation of a first catalog of the most frequent problems when languages interact. Finally, we believe it is important to keep on studying the effects on defect proneness caused by languages interactions, and to extend the investigation on the on the effect on productivity and code maintainability;

Categorize language interaction mechanisms: in the literature only cross references implemented through common identifiers are discussed. Other possible interactions are not described at all. An ontology of the kinds of interactions would be a relevant contribution. A good starting point would be the categorisation of language modularisation for DSLs as presented by Völter [4].

Techniques to express cross language constraints: while the simple example we showed in section IV permits to understand the potential of using Language workbenches for language integration, complete approaches have to be developed and validated in practice, possibly in different domains and on projects of different sizes;

Queries involving multiple languages: a more mature language integration would permit to query the system under development, for example about all the references to a particular element, including cross-language references. The possibilities of global queries as opposed to single-language queries have yet to be explored,

Custom persistency: the example we presented was based on the editing of XML and Java models, stored in the MPS proprietary format. One of the recent features of MPS is the possibility to use custom format for persistency (the same feature is present as well in other language workbenches). It would be possible therefore to edit XML files in MPS, while storing them as XML files, instead of using the MPS persistency format. We believe that the diverse persistency mechanisms should be investigated and assessed.

Acknowledgments

We would like to thank the DAAD that partially financed this work. Mbeddr has been supported by the German BMBF, FKZ 01/S11014.

References

[1] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *Principles of Software Evolution, Eighth International Workshop on*, sept. 2005, pp. 13 – 22.

[2] A. Vetro', F. Tomassetti, M. Torchiano, and M. Morisio, "Language interaction and quality issues: an exploratory study," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ser. ESEM '12. New York, NY, USA: ACM, 2012, pp. 319–322. [Online]. Available: http://doi.acm.org/10.1145/2372251.2372309

[3] K. Nie and L. Zhang, "On the relationship between preprocessor-based software variability and software defects," in *High-Assurance Systems Engineering (HASE), 13th Int. Symp. on*, 2011, pp. 178 –179.

[4] M. Völter, "Language and ide development, modularization and composition with mps," in *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2011*, ser. LNCS. Springer, 2011.

[5] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*. Pearson Education, 2003.

[6] L. C. Kats and E. Visser, "The spoofax language workbench," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. SPLASH '10. New York, NY, USA: ACM, 2010, pp. 237–238. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869592

[7] D. Groenewegen and E. Visser, "Declarative access control for webdsl: Combining language integration and separation of concerns," in *Web Engineering, 2008. ICWE '08. Eighth International Conference on*, july 2008, pp. 175 –188.

[8] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The epsilon object language (eol)," in *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*, ser. ECMDA-FA'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 128–142.

[9] M. L. Barja, N. W. Paton, A. A. A. Fernandes, M. H. Williams, and A. Dinn, "An effective deductive object-oriented database through language integration," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 463–474. [Online]. Available: http://dl.acm.org/citation.cfm?id=645920.672969

[10] J.-P. Tolvanen and S. Kelly, "Integrating models with domain-specific modeling languages," in *Proceedings of the 10th Workshop on Domain-Specific Modeling*, ser. DSM '10. New York, NY, USA: ACM, 2010, pp. 10:1–10:6. [Online]. Available: http://doi.acm.org/10.1145/2060329.2060354

[11] P. Mayer and A. Schroeder, "Cross-language code analysis and refactoring," in *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, sept. 2012, pp. 94 –103.

[12] R.-H. Pfeiffer and A. Wasowski, "Texmo: a multi-language development environment," in *Proceedings of the 8th European conference on Modelling Foundations and Applications*, ser. ECMFA'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 178–193.

[13] R.-H. Pfeiffer and A. Wasowski, "Cross-language support mechanisms significantly aid software development," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds. Springer Berlin Heidelberg, 2012, vol. 7590, pp. 168–184.