Towards Multilingual Programming Environments

Tijs van der Storm, Jurgen Vinju

CWI, Amsterdam

Abstract

Software projects consist of different kinds of artifacts: build files, configuration files, markup files, source code in different software languages, and so on. At the same time, however, most integrated development environments (IDEs) are focused on a single (programming) language. Even if a programming environment supports multiple languages (e.g., Eclipse), IDE features such as cross-referencing, refactoring, or debugging, do not often cross language boundaries. What would it mean for programming environment to be truly multilingual? In this short paper we sketch a vision of a system that integrates IDE support across language boundaries. We propose to build this system on a foundation of unified source code models and metaprogramming. Nevertheless, a number of important and hard research questions still need to be addressed.

Keywords: Programming environments, language interoperability, metaprogramming

Programming environments are an important thread through Paul's research career. From the early papers on the ASF+SDF Meta-Environment to the Rascal programming language of today, a constant focus of Paul's work has been to improve the life of the programmer with better tools, better designs, and better languages. In the meantime the software landscape has only become more complex, more heterogeneous and more multi-faceted. This short paper envisions a programming environment that both embraces and unifies this multiplicity using one of Paul's favorite topics: metaprogramming. Thanks Paul, for our professional careers and for your friendship. We hope you enjoy reading this paper.

1. Introduction

Most software projects consist of many kinds of different artifacts, in different languages. For instance, a typical Java Web application project might contain Java source files, templates (e.g., JSP), Javascript source files, SQL schema definitions, ORM mapping files, and HTML templates. All these artifacts are related to each other. They may refer to each other through special names. For instance, class names coincide with database tables and HTML templates refer to tag libraries. Generally IDEs do not take the inter-operation of these languages into account; the reference links are not explicitly modeled, and thus not actionable, but they do exist in the code base. As a result, supporting features such as cross-referencing, refactoring, and debugging do not cross linguistic boundaries. This leads to inaccurate support which eventually leads to bugs that must be resolved later.

Another source of language multiplicity is the meta information that is necessary to build, configure and deploy projects: build files (e.g., Ant, Maven), configuration files (e.g., Spring XML files). These languages seem secondary, but they have big impact on the semantics of the eventually running program, and as such may contain bugs. More problematically, language support for the other languages (such as Java) does not model most of the information in this meta data, introducing inaccuracy in IDE features such as quick

URL: storm@cwi.nl (Tijs van der Storm), jurgenv@cwi.nl (Jurgen Vinju)

lookup, flow graphs and autocomplete. The more frameworks that offer reuse and high levels of abstraction are introduced, the harder it becomes to provide meaningful IDE features.

Language referencing and meta information are just two examples of relations between languages. Other relations between languages include containment, where one language is embedded in another (e.g., SQL in COBOL code), — or derivation where one language is compiled into another language (e.g., code generation of a DSL). We expect truly multilingual IDEs to take all of these relations into account.

Common IDEs like Eclipse provide IDE support for some of these languages and some of their combinations. They are, however, primarily targeted at a single programming language and there exists ample opportunity for further integration. For instance, Eclipse is mainly an IDE for Java. The plugin system of Eclipse allows users to get IDE support for other languages, such as JavaScript, XML, SQL, etc. However, such plugins are mostly isolated from each other: integration across language boundaries is limited. Often, each language lives in its own silo. As a result, the programmer constantly has to switch perspectives and mentally keep the different artifacts in sync.

In this paper we analyze this problem in some depth and propose an approach to overcome these limitations. The key aspect of our design is to consider the multiplicity of languages as a federation of languages. Only the combination of all artifacts leads to the software system captured by the project, — in a sense there is only one big, composite language. Understanding a software project thus means understanding how these languages work together. Consequently, we hope, truly multilingual IDE support will be instrumental in improving understanding and thus in helping to construct and maintain high quality software.

2. Towards monolingual programming environments, redux

Heering and Klint wrote "Towards monolingual programming environments" in 1985 [8], warning us for the complexity of the exploding number of languages in programming environments and proposing to fully reverse this development into a single language with a single and consistent programming and debugging environment. Today we are faced with what we were warned for: hundreds of independent languages for programming, scripting, configuring, defining, and debugging software. Moreover, due to the availability of memory and disk space, we now have all these languages installed and active within the same computer system. For the sake of argument, let us assume this complex reality was introduced for all good reasons.

In this paper we propose to view the de facto multiplicity of languages that a programmer is subjected to as a single, federated language. This federation of languages encompasses all kinds of "source code in the broad sense" [18]. What is the syntax of this language? What is its semantics? How do we model name resolution, declarations, uses, control flow, data flow, and types for this language? Given answers to these questions, we will have a principled method of modeling cross language semantic dependency. On top of such generic models, advanced IDE features such as refactoring tools, debugging, hover help, reference hyperlinks and auto-complete may be constructed. We propose a high level design and a research agenda towards integrated, multi-lingual development environments.

2.1. One IDE to rule them all

Here we describe the high-level requirements for enabling the syntactic and semantic integration of IDE support for multiple software languages. This design acts as a frame of reference for identifying the open problems that we discuss in Section 3.

Uniform representation. At a very basic level language artifacts are a form of structured data. To allow this data to be processed in a typed and uniform way, requires three meta-level services. First, the structure of the data should be modeled in a uniform way, for instance using meta modeling, data description, or schema language. Second, the data itself should have a uniform, typed in-memory representation. Finally, a typed serialization/deserialization service is needed to load the language artifacts to/from disk.

Uniform identification. Artifacts and sub-entities of those artifacts often have an identity to be able to refer to them. When a project consists of multiple languages, there are multiple meta models at play. Through the uniform representation the artifacts can be processed in a uniform way. However, each language might provide specific ways of identifying entities. To refer to elements from different languages in a uniform way we need a generic identification mechanism that works for all of them.

Modeling relations. There are many examples of possible relations between artifacts and relations between sub-artifact elements: containment, call graphs, use-define relations, import relations, control-flow and data flow graphs, inheritance relations etc. It should be possible to super-impose such relations on top of the uniform representation of the artifacts. The generic identification mechanism plays a key role here.

Hooks into the user interface. The analyses and transformations realized on top of the uniform representations of artifacts and the relations between them should be made available to the programmer through user interface affordances. The key requirement, however, is that the interface is language parametric. It should be possible to, for instance, provide syntax coloring for a language without having to customize the IDE per language. The same holds for invoking refactorings, hyperlinking artifacts, creating outlines etc.

2.2. From reverse engineering to forward engineering

The idea of generic language interoperability, composition or integration is not new. Especially in the fields of re-engineering and reverse engineering it was recognized earlier that software projects have to deal with language multiplicity. To be able to understand a software system, one must understand all of its components and both their implicit and explicit dependencies. At first, only uniform fact exchange formats [22, 12], explicitly modeled relations [11] and programmable visualizations [23] existed to conquer this challenge. More depth to this was added by explicitly analyzing other forms of source code, such as databases schemas [3], and configuration files.

In particular, Yazdanshenas and Moonen used the concept of system dependence graphs (SDG), to model cross language dependency and make these dependencies actionable [38, 37]. Perin extended the Moose system for architectural reconstruction supporting cross language modeling [24]. The MoDisco Eclipse plugin[1] is a reverse engineering workbench which supports multiple languages and cross language dependencies as well. This is based on the Knowledge Discovery Meta-Model (KDM) model[2], a language independent meta model for knowledge discovery which is specifically designed to cross language boundaries. Finally, the SemmleCode environment employ a generic DataLog based model to express and query cross language relations [34]. .

We propose to bring as much as possible of these reverse engineering tools to the IDE for forward engineering. In particular generic models such as KDM and system dependence graphs are a fundamental tool in this respect. This introduces challenges in terms of scale (in terms of the kinds and amount of languages supported), performance (IDE responsiveness) and accuracy (correctness). Statically analyzing scripting languages [14, 13, 9] is particularly hard, but necessary for this vision to work. Also, advanced IDE tools require more in-depth analysis than architecture reconstruction tools do. Tools such as refactoring and debugging have intrinsic knowledge about static and dynamic semantics that may be abstracted from in the reverse engineering context, but not in these contexts.

2.3. Integrating languages

Below we list four important alternative perspectives for bringing different languages closer together in the forward engineering context:

- Linguistic middleware: this is the approach taken by the ASF+SDF Meta-Environment [32] where the Toolbus architecture [4] was used to coordinate the different components of IDEs and languages: parsers, compilers, editors, debuggers [30], – all these components were connected to a central bus which routed data and events to and from heterogeneous components.

[1] `http://www.eclipse.org/MoDisco`
[2] `http://www.omg.org/spec/KDM/1.3/`

- Language as library: in this case languages are integrated at the syntactic level by embedding one language into another. The integration of C into Java to bridge the JNI interface is an example [10]. More generically, SugarJ allows the embedding of custom defined DSLs into a host language (such as Java) [5]. This leads to an approach where languages act as libraries [29]. The perspective described in "Towards monolingual programming environments" [8], represents an extreme design point where a single language offers the best tools for all the requirements a programmer may have.

- Projectional editing: another approach is taken by language workbenches based on structure editors, such as MPS [35] and the Intentional Workbench [26]. In this case languages are integrated by requiring the programmer to edit the uniform representation of programs directly through textual or graphical projections in the GUI. As a result, all language artifacts *a priori* are represented using the same data structures.

- Model driven engineering: in language workbenches [16, 6, 32] generic and reusable meta models (such as grammars and class diagrams) exist primarily for reuse purposes, fed by the commonality between all languages. A meta-modeling format such as ECore [3] may also facilitate cross language integration. A generic debugging interface, such as TIDE [30], facilitates the construction of debuggers for new languages, and may be extended to facilitate debugging between languages as well. It has already been used to debug between different levels of abstraction (between interpreting and interpreted language).

What is problematic with these efforts to integrate languages from our perspective, however, is that they assume a future world in which all software will be written in a more integrated fashion, or a world in which all existing software will be transformed into the new integrated perspective.

In our proposal the multiplicity of artifacts remains as it is. We take the reverse engineering stance towards the languages, and the forward engineering stance towards their IDE: to make all their syntactic and semantic inter-relations manifest.

2.4. Enter Rascal: metaprogramming to bring languages together

The Rascal metaprogramming language [19, 20] was specifically designed to create source code models and map source code to these models. Three use cases, namely reverse engineering, refactoring tools and engineering of domain specific languages, have been the inspiration of its design. Rascal is becoming a language which natively supports the modeling, analysis, transformation and generation of source code artifacts in an integrated manner. Below we describe how some of the feature of Rascal address the requirements of Section 2.1:

Algebraic Data types (ADTs) and typed parse trees. Software language artifacts can be generically represented using ADTs and type parse trees. ADTs can be used to describe all kinds of tree structures, such XML documents, Abstract syntax trees, the nesting structure of text documents etc. As a result these data types provide a uniform representation for source code artifacts. The same models can also be used for abstract (symbolic) execution, representing for example type constraints and logical expressions. Rascal's built-in context-free grammars and Box pretty-printing language [31] allow complex programming language artifacts to loaded for and rendered to text in a declarative fashion. Moreover, a number of standard library modules provide access to XML, YAML and JSON files.

Source locations. Rascal features source locations as a built-in data type (`loc`). They can be used as unique identifiers for artifacts and fragments of artifacts. For example: `|file:///tmp/HelloWorld.java|(0,100)`, points to the characters ranging from index 0 to 100 in the HelloWorld.java file in the `/tmp` directory on the current machine. New URI schemes can be introduced for the `loc` type to define qualified names for a language. Eventually each user defined scheme resolves to a physical URI. For example: the get method in the List interface of the Java Collections API is identified as `|java+method:///java/util/List/get|`. Here

[3] `http://www.eclipse.org/emf/`

we use the scheme separator `+` to define first the language and then the type of semantic artifact that is identified. The language specific identification schemes help to make model extraction and the models themselves stable under code modification (akin to modular and incremental compilation which is facilitated by a separate linking phase).

Maps, sets, and relations. Relations between entities can be conveniently expressed using built-in map, set and relation data types. For instance, containment is modeled in Rascal using a relation: `rel[loc outer, loc innner]`. Allowing quick lookup, transitive closure and other queries directly from its expression language. Call graphs are modeled in Rascal using a relation too: `rel[loc caller, loc callee]`. Note how this representation allows calls between different languages via the qualified name representations. Declarations are modeled in Rascal using a `set[loc]`. Note that we might index such relations per module, resulting in `map[loc module, set[loc] declarations]` to represent modularized extracted fact. This could help making analyses more incremental.

Metaprogramming. Rascal's design is inspired by the Extract-Analyze-SYnthesize paradigm[4] (EASY) [20]. Many of the use cases for Rascal follow the structure of extracting source code facts, analyzing them and synthesizing a result. A typical example is typechecking: extract a set of constraints, solve them, and finally, if there are errors synthesize a set of error messages to be shown in the IDE. Another example is transforming a language artifact to a tree structure that can be displayed as an outline view in the IDE. Rascal's built-in features for parser generation, pattern matching, tree traversal, relational calculus, and string templates are used across each of the EASY steps.

In summary, Rascal can support a simple and uniform mechanism for extracting and representing basic language independent source code models, both syntactic and semantic. This removes the need for a combinatorial number of language interactions. Instead of modeling each of the exponential number of interactions between two languages one-by-one, we map each language into the common representation (such as KDM, SDGs) one-by-one using high-level metaprogramming techniques. Technological infrastructure alone, however, is only the first step to achieve multilingual IDEs. Below we review the research questions raised our proposal.

3. Open challenges, questions and opportunities

Commercial forces. One could argue that one of the causes of the current language explosion problem is the commercial interest of individual companies: branding and lock-in. At the same time, this effect hampers the creation of the multilingual IDE as well. One solution may be to very radically separate the GUI and interaction facilities in the IDE from the extraction, modeling, analysis and transformation facilities, such that language components may be shared across different IDE frameworks (e.g. Visual Studio, Eclipse, IntelliJ IDEA). In today's IDEs a tight integration between these concerns has been observed [2]. Perhaps the service-oriented paradigm may offer solutions that are economically viable as well: language-as-a-service?

Bridging technical spaces. The artifacts used in software projects range from traditional source code, markup documents, requirements documents, spreadsheets, serialized data, configuration files, log files, templates, UML models, API documentation, HTML5 documents, Office documents, etc. As mentioned earlier, there exists a multiplicity of relations among many of these languages. However, in typical software projects these relations are implicit (by convention). The challenge is then to faithfully model the semantics of such bridges in the language integrated IDE.

For instance, an ORM mapping tool relates Java source code to SQL table definitions. Such tools encode complex mappings between two technical spaces: OO classes and Entity-Relation model. To support IDE interactions that take both spaces into account then boils down to model the semantics that is encoded in the ORM tool. This in itself represents a quite daunting task. It is unclear how generic metaprogramming

[4]EASY is an extension of the "extract-query-present" pattern (see, e.g., [36]), by including source-to-source transformation and staged models

support could make the construction of such bridging models easier. There seems to be no general answer to this problem: each interaction will come with specific constraints and limitations. Nevertheless, exploring reusable and extensible meta-level analyses or transformations is a challenging goal of the research we propose (see, e.g., [21] for a similar position).

Staged and embedded languages. A particular complex kind of language inter-operability is when languages are nested in each other or on top of each other. Some languages are compiled into other languages (staged), or embedded fragments are assimilated into a larger host language [1]. The C language with its pre-processor is a prime example of a staged language, making IDE services such a refactoring crumble [27, 15].

Although the result of embedded language designs are often pleasing on the language syntax level and simplify basic IDE support such as parsing, highlighting and referencing, the more advanced IDE tools such as tracing and debugging are very hard to get right since the language distinctions have disappeared at run-time and sometimes even at compile-time. The problem of embedded languages has been attacked for specific host languages [25], and code generation systems, but there still remains a whole set of legacy staged and embedded languages where meta information about the original embedding must still be reified somehow to provide useful language-level information to the programmer.

An integrated IDE must model the fact that some code has been generated and that some code will be interpreted at run-time. In fact, even the simple feature of an IDE preventing the user from editing generated code is currently missing in most IDEs.

Semantic models. The basic IDE features are mostly of a syntactical nature: "Jump to definition", syntax highlighting, code folding and outlines. The more interesting features however, require in-depth static and sometimes even dynamic analysis. Refactoring tools are perhaps the most challenging, since they come with the hard promise of run-time semantics preservation. What it means for refactorings to be cross-language is witnessed by some examples in Eclipse. The rename refactoring is propagated by the JDT in javadoc source code comments as well as OSGI configuration files. For more advanced refactorings however, language inter-operability is more interesting. Here are a few basic examples. "Change method signature" should not be allowed to add parameters to constructors of classes that will be instantiated by an Eclipse extension point. "Introduce type parameter" on an abstract class that has been generated from a BNF grammar can not be executed unless the supporting BNF formalism supports type parameter on non-terminals as well, and this probably has an influence on the upper bound for the new parameter ("T extends ASTNode").

Supporting the time dimension. Today we are helped enormously by the tight integration of the IDE with source repository support (git, svn). Commonly this versioning support lives mostly on the file or syntactical level. At the same time, useful historical analyses on the abstract semantical level are being developed, such as clone evolution [28]. The problem with abstract models and language inter-dependencies in the IDE is that they too must provide historic information in order to provide meaningful support [7]. This problem is exacerbated by the continuous evolution of the respective languages and frameworks, such that one cross-language inter-dependency may change over time not only because the software has changed, but also because the semantics of the supporting languages and their inter-operation has changed (see below).

Analyzing dynamic languages. The popularity of dynamically typed programming languages for software construction, like Ruby, Python, Javascript or PHP has been steadily increasing recently, but the use of scripting languages to glue systems together has always been popular. Dynamical language provide particular challenges for multilingual IDEs since static analysis of such languages is known to be very hard [9]. On the other hand, the multilingual perspective also provides opportunities in that different artifacts (other than the source code itself) may supplant some of the missing information that makes these analyses hard in the first place. The analysis of configuration files, coding idioms, or framework specific conventions has the potential to make static analyses of dynamic languages more feasible.

Modeling a moving target. Finally, we arrive at the challenge of dealing with language evolution. Programming languages such as C#, C++ and Java evolve quickly, but modeling languages evolve even faster and configuration languages are often not even versioned because each version of a system comes with a new

dialect of the configuration language. Building simple tools such as outlines suffer from such evolution, but advanced tools such as refactoring tools, debuggers simply break on it. We need to re-think the design of advanced IDE tools to a priori allow for language evolution, supporting backward compatibility and forward extensibility at the same time [35]. As witnessed by the lack of support in refactoring tools for the newest versions of Java, advanced IDE features are hard to keep up-to-date even for a single language.

The universal model trap. Although not a challenge in itself, we wish to point out a pitfall in the solution direction we pointed out. Although shared universal models seem the way out of a combinatorial explosion of language inter-operation, they can also introduce dangerous inaccuracy or even errors.

Consider a "universal abstract syntax tree format", where an if-then-else node type or binary boolean expression node types may exists which may be shared among many different languages. Even the simplest metrics tools, such as computing the Cyclomatic Complexity of a method body, will produce wrong results if the differences between the semantics of the languages are ignored simply because they have the same representation in the abstract model. What if one language short-circuits && and the other does not?

Another example: import statements induce transitive dependency in one language and non-transitive dependency in another, or compile time dependency in one language and also run-time dependency in another. Simply modeling dependency as an abstraction that has the same meaning across different languages is bound to lead to confusion. The design challenge is thus to introduce as much reuse as possible without ignoring the devilish details of the static and dynamic semantics of each different language.

Responsiveness of IDE services. Let us be brief about our final and biggest challenge. In order to achieve IDE responsiveness, the kind of algorithms necessary to produce accurate cross language analysis results are often infeasible. The answers may lie in applying modular processing [17], incremental algorithms [33], and concurrency. We do have some work for all these idle cores.

4. Concluding remarks

Understanding tomorrow's software is about understanding software projects from a holistic standpoint. We identified a challenge: the construction of an IDE that understands the heterogeneous reality of software projects as they are constructed via many interdependent and interoperating languages. On the one hand, our proposal aims to solve these problems by providing common representations and interoperability for selected abstract semantic models (references, call graphs, etc.). One the other hand, we still have to face the daunting task of modeling the moving target of myriads of software languages and come up with scalable analysis techniques.

Acknowledgements

We thank Leon Moonen and Arie van Deursen for providing helpful feedback.

References

[1] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA*, pages 365–383. ACM, 2004.

[2] P. Charles, R. Fuhrer, S. Sutton, Jr., E. Duesterwald, and J. Vinju. Accelerating the creation of customized, language-specific ides in eclipse. In *OOPSLA*, pages 191–206. ACM, 2009.

[3] A. Cleve, J. Henrard, and J.-L. Hainaut. Data reverse engineering using system dependency graphs. In *13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 157–166. IEEE Computer Society, 2006.

[4] H. de Jong and P. Klint. Toolbus: the next generation. In *FMCO*, volume 2852 of *LNCS*, pages 220–241. Springer, 2003.

[5] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA*, pages 391–406. ACM, 2011.

[6] M. Eysholdt and H. Behrens. Xtext: Implement your language faster than the quick and dirty way. In *SPLASH Companion*, pages 307–309. ACM, 2010.

[7] V. U. Gomez. *Supporting Integration Activities in Object-Oriented Applications.* PhD thesis, Vrije Universiteit Brussel, 2012.

[8] J. Heering and P. Klint. Towards monolingual programming environments. *ACM Trans. Program. Lang. Syst.*, 7(2):183–213, Apr. 1985.
[9] M. Hills, P. Klint, and J. J. Vinju. An empirical study of php feature usage: a static analysis perspective. In *ISSTA*, pages 325–335, 2013.
[10] M. Hirzel and R. Grimm. Jeannie: granting java native interface developers their wishes. *SIGPLAN Not.*, 42(10):19–38, Oct. 2007.
[11] R. C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *WCRE*, page 210. IEEE Society, 1998.
[12] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a standard exchange format. In *WCRE*, page 162. IEEE, 2000.
[13] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the `eval` that men do. In *ISSTA*, pages 34–44, 2012.
[14] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *SIGSOFT FSE*, pages 59–69, 2011.
[15] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 805–824, New York, NY, 10 2011.
[16] L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463. ACM, 2010.
[17] P. Klint, A. Kooiker, and J. Vinju. Language parametric module management for IDEs. *ENTCS*, 203(2):3–19, 2008.
[18] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM TOSEM*, 14(3):331–380, July 2005.
[19] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. *SCAM*, pages 168–177, 2009.
[20] P. Klint, T. van der Storm, and J. Vinju. EASY meta-programming with Rascal. In *GTTSE III*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.
[21] R. Lämmel and J. Heering. Generic software transformations (extended abstract). Workshop on Software Transformation Systems, 2004.
[22] H. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *ICSE*, pages 80–86. IEEE, 1988.
[23] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of moose: an agile reengineering environment. In *ESEC/FSE-13*, pages 1–10. ACM, 2005.
[24] F. Perin. *Reverse Engineering Heterogeneous Applications*. PhD thesis, Universität Bern, 2012.
[25] L. Renggli. *Dynamic Language Embedding With Homogeneous Tool Support*. Phd thesis, University of Bern, Oct. 2010.
[26] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *OOPSLA*, pages 451–464. ACM, 2006.
[27] D. Spinellis. Cscout: A refactoring browser for c. *Science of Computer Programming*, 75(4):216 – 231, 2010. ¡ce:title¿Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008)¡/ce:title¿.
[28] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Softw. Engg.*, 15(1):1–34, Feb. 2010.
[29] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *PLDI*, pages 132–141. ACM, 2011.
[30] M. van den Brand, B. Cornelissen, P. Olivier, and J. Vinju. TIDE: A generic debugging framework - tool demonstration. *ENTCS*, 141(4):161–165, 2005.
[31] M. van den Brand, A. Kooiker, and J. Vinju. A language independent framework for context-sensitive formatting. In *CSMR*, pages 103–112. IEEE, 2006.
[32] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *CC*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.
[33] E. van der Meulen. *Incremental Rewriting*. PhD thesis, University of Amsterdam, 1994.
[34] M. Verbaere, E. Hajiyev, and O. De Moor. Improve software quality with semmlecode: an eclipse plugin for semantic code search. In *OOPSLA Companion*, pages 880–881, 2007.
[35] M. Voelter and V. Pech. Language modularity with the MPS language workbench. In *ICSE*, pages 1449–1450. IEEE, 2012.
[36] K. Wong. *Untangling safety-critical source code*. PhD thesis, University of British Columbia, 2005.
[37] A. Yazdanshenas and L. Moonen. Crossing the boundaries while analyzing heterogeneous component-based software systems. In *ICSM*, pages 193–202, 2011.
[38] A. Yazdanshenas and L. Moonen. Tracking and visualizing information flow in component-based systems. In *ICPC*, pages 143–152, 2012.