

IT University of Copenhagen

PhD Thesis

Multi-language Development Environments


—

Design Space, Models, Prototypes,
Experiences

Rolf-Helge Pfeiffer
March 29, 2013

Supervisor
Andrzej Wąsowski

IT University of Copenhagen
Software and Systems Section
Rued Langgaards Vej 7
2300 Copenhagen S



Abstract

Non-trivial software systems are constructed out of many artifacts expressed in multiple modeling and programming languages describing different system aspects on different levels of abstraction. I call such systems multi-language software systems. Even though artifacts constituting multi-language software systems are heavily interrelated, existing development environments do not sufficiently support developers of such systems. In particular, handling relations between heterogeneous artifacts is not supported at development time.

This thesis *a)* studies the characteristics of contemporary multi-language software systems, *b)* it investigates features needed in software development tools and environments to support or enhance multi-language software system development, and *c)* it establishes required knowledge and building blocks for creation of multi-language development environments.

I address these research goals by applying tool prototyping, technical experiments, user experiments, surveys, and literature survey as methodological tools.

The main results of this thesis are *a)* a taxonomy for construction, comparison, and characterization of multi-language development environments, *b)* the identification of navigation, static checking, visualization and refactoring of cross-language relations as four elementary cross-language support mechanisms, *c)* the experimental evaluation of multi-language software system development with support of cross-language support mechanisms, *d)* the characterization of two industrial strength open-source systems as multi-language software systems by technical experiments, *e)* a characterization of the state-of-the-art in development of multi-language software systems and in multi-language development environments, and lastly *f)* a set of open-source prototype tools, which implement different language representations and relation models and which implement different facets of multi-language development environments.

My main conclusions are *a)* contemporary software systems are indeed multi-language software systems, *b)* relations between heterogeneous development artifacts are ubiquitous and troublesome in multi-language software systems; they pose a real problem to development and evolution of multi-language software systems, *c)* users highly appreciated cross-language support mechanisms of multi-language development environments, *d)* generic multi-language development environments clearly enhance the state of the art in tooling for language integration, and *e)* multi-language development environments can be constructed by cautiously deciding on a language representation for heterogeneous development artifacts, a model for relations between heterogeneous development artifacts, and considering typical properties of heterogeneous development artifact relations.

Acknowledgements

There are many people, who have their share in that I deliver this thesis. This is of course my family. I thank my brother Hendrik for always being interested in my opinions, having an open ear, and for sharing my enthusiasm about the subject. I also thank my mother Gabriele for supporting me in going my way.

Furthermore, I would express my thanks to some of the new friends I found here in Copenhagen. I thank Christoph Froeschel for helping me to better understand the Danes and for the interesting discussions. Mads Johansen always selected the right music to the coffee in the morning. Especially Paper **B** would not have been possible without the relaxed atmosphere in our sunny living room.

My old friends Vinzenz Hilbert and Konrad Hotzel enriched my thoughts in long discussions about resonance, eigenfrequency, and atomic structures. I am very thankful for these discussions and I am still thinking about this. . .

I offer my thanks to all the nice and interesting people whom I got to know during my project. Jan Reimann, Uwe Aßmann, Jendrik Johannes, Sven Karol, Mirko Seifert, Christian Wende, Julia Schröter, Claas Wilke, Jan Polowinski, and everyone at the Software Development Group at TU Dresden. I thank you all for the friendly reception, the discussions, and our reading group during my stay abroad.

Here at IT University of Copenhagen, I thank all the members of the Software Development Group. For all the discussions, feedback, and insight I thank especially my supervisor Andrzej Wąsowski, Kasper Østerbye, Peter Sestoft, Joe Kiniry, Philippe Bonnet, David Christiansen, Hannes Mehnert, Andrea Campagna, Paolo Tell, Rosalba Giuffrida, Josu Martinez, Kevin Tierney, Dario Pacino, Alberto Delgado-Ortegon, and Fabrizio Biondi. I am deeply thankful for the good time I had with you as colleagues.

Additionally, I would like to thank the Danish people for making it possible that I conduct my research at IT University in Copenhagen. It is a great place.

Finally, where would I have been without music? I thank the samba schools Bafo do Mundo and Samba Universo for getting me into the groove and all the joyful Sundays. Furthermore, I thank my double bass teacher Mathias Wedeken for guiding me in unknown territory and for making me see patterns.

Contents

1	Introduction	11
1.1	Preface	11
1.1.1	List of Papers	11
1.1.2	Tools Developed in this Project	12
1.1.3	Additional Contributions	13
1.2	Outline	14
2	Motivation	15
2.1	Contemporary Software Systems – Multi-language Software Systems are Real	15
2.2	Software System Development – The Confusion of Languages	17
2.2.1	JTrac – A Java Web-application for Issue-tracking . .	20
2.2.2	OFBiz – The Apache Open for Business Project	21
2.3	Contemporary Development Environments – Taming the Confusion of Languages	23
3	Terminology	27
3.1	Characteristics of Development Artifacts	27
3.2	The Internal Structure of Software Systems	33
3.3	Relations between Development Artifacts	33
3.3.1	Relation Models – Explicit Relation Representation . .	37
3.4	Software Development Tools	41
4	The Design Space of Multi-language Development Environments	43
4.1	Language Representation	43
4.1.1	Lexical Language Representation	43
4.1.2	Syntactic per Language Representation	44

4.1.3	Syntactic per Language Group Representation	45
4.1.4	Syntactic Universal Representation	48
4.2	Relation Models	48
4.2.1	Explicit Relation Model	49
4.2.2	Tags	51
4.2.3	Interfaces	52
4.2.4	Search-based Relation Model	53
4.3	Relation Types	54
4.4	Inference of Relations between Development Artifact	56
4.4.1	Inference by Program Instrumentation	56
4.4.2	Inference out of Development Artifacts	57
4.5	Cross-language Support Mechanisms	58
4.6	Overview and Comparison of Related Work	60
5	Problem Definition	63
5.1	Research Questions	64
5.2	Theses	65
6	Solution Overview	67
6.1	Methodology	67
6.2	Summary & Contributions per Paper	68
6.2.1	An Aspect-based Traceability Mechanism for Domain Specific Languages – ECMFA-TW’10 (Paper A)	68
6.2.2	Taming the Confusion of Languages – ECMFA’11 (Pa- per B)	70
6.2.3	Tengi Interfaces for Tracing between Heterogeneous Components – GTTSE’11 (Paper C)	72
6.2.4	TexMo: A Multi-Language Development Environment – ECMFA’12 (Paper D)	73
6.2.5	Cross-Language Support Mechanisms Significantly Aid Software Development – MODELS’12 (Paper E)	76
6.2.6	The Design Space of Multi-language Development En- vironments – SoSyM’13 (Paper F)	77
6.2.7	Language-Independent Traceability with Lässig – Un- der Submission (Paper G)	80
6.3	Contributions in a Nutshell	82
7	Discussion, Conclusion, and Future Work	85
7.1	Discussion & Conclusions	85
7.1.1	Thesis T1 – Multi-language Software Systems	85
7.1.2	Thesis T2 – Developer Support	87

7.1.3	Thesis T3 – Tool Builder Support	89
7.2	Contribution to Community’s Research Agendas	93
7.2.1	On the Unification Power of Models [24]	93
7.2.2	A Model-based Approach to Language Integration [125]	94
7.3	Future Work	95
8	An Aspect-based Traceability Mechanism for Domain Specific Languages – ECMFA-TW’10 (Paper A)	111
9	Taming the Confusion of Languages – ECMFA’11 (Paper B)	125
10	Tengi Interfaces for Tracing between Heterogeneous Components – GTTSE’11 (Paper C)	143
11	TexMo: A Multi-language Development Environment – ECMFA’12 (Paper D)	159
12	Cross-language Support Mechanisms Significantly Aid Software Development – MODELS’12 (Paper E)	177
13	The Design Space of Multi-language Development Environments – SoSyM’13 (Paper F)	195
14	Language-independent Traceability with Lässig – Under Submission (Paper G)	245
	Appendices	263
A	Variability Mechanisms in Software Ecosystems: Closed versus Open Platforms – Under Submission	265
B	Multi-language Software Systems on GitHub	277



Figure 1: *Confusion of Tongues*,
Gustave Doré, 1865
([http://en.wikipedia.org/wiki/
Confusion_of_tongues](http://en.wikipedia.org/wiki/Confusion_of_tongues))

This thesis is based on a collection of papers. The following chapters motivate my work, present the used terminology, summarize the state of the art, state the research problems, and present a summary of my contributions. The core part of this document, the research papers themselves, are included in Chapter 8 to Chapter 14.

1.1 Preface

This section provides an overview of the research papers and research tools which I wrote and developed during my PhD project. I also summarize additional works which I have co-authored. They are either of lesser significance for this thesis or in early stages of research. Thus, they are not discussed in detail in the subsequent chapters. For all other contributions I provide links to online resources and references to their occurrence in this document. Note, all published research papers are peer-reviewed.

1.1.1 List of Papers

- Paper A** Rolf-Helge Pfeiffer and Andrzej Wąsowski: “*An Aspect-based Traceability Mechanism for Domain Specific Languages*” Published in: ECMFA-TW ’10 Proceedings of the 6th ECMFA Traceability Workshop, doi:10.1145/1814392.1814399 (Chapter 8)
- Paper B** Rolf-Helge Pfeiffer and Andrzej Wąsowski: “*Taming the Confusion of Languages*” Published in: ECMFA’11 Proceedings of the 7th European Conference on Modelling Foundations and Applications, doi:10.1007/978-3-642-21470-7_22 (Chapter 9)
- Paper C** Rolf-Helge Pfeiffer and Andrzej Wąsowski: “*Tengi Interfaces for Tracing between Heterogeneous Components*”, Published in: GTTSE’11 Proceedings of the 4th International Summer School on Generative and Transformational Techniques in Software Engineering, doi:10.1007/978-3-642-35992-7_12 (Chapter 10)
- Paper D** Rolf-Helge Pfeiffer and Andrzej Wąsowski: “*TexMo: A Multi-language Development Environment*”, Published in: ECMFA’12 Proceed-

ings of the 8th European Conference on Modelling Foundations and Applications, doi:10.1007/978-3-642-31491-9_15 (Chapter 11)

Paper E Rolf-Helge Pfeiffer and Andrzej Wąsowski: “*Cross-Language Support Mechanisms Significantly Aid Software Development*”, Published in MODELS’12 Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems, doi:10.1007/978-3-642-33666-9_12 (Chapter 12)

Paper F Rolf-Helge Pfeiffer and Andrzej Wąsowski: “*The Design Space of Multi-Language Development Environments*”, Published in the Journal of Software and Systems Modelling, doi:10.1007/s10270-013-0376-y (Chapter 13)

Paper G Rolf-Helge Pfeiffer, Jan Reimann, and Andrzej Wąsowski: “*Language-Independent Traceability with Lässig*”, Under submission (Chapter 14)

1.1.2 Tools Developed in this Project

Tengja is a tool for automatic generation of trace links between visual models (Eclipse GMF) and their serialization syntax. The tool is based on aspect-oriented observation of the serialization of visual models. The tool is available online¹ and it is described in detail in Paper A (Chapter 8).

GenDeMoG the Generic Dependency Model Generator, is a tool for pattern-based inference of an explicit relation model out of source code, for example, in XML-based domain-specific languages and Java. GenDeMoG is available online² together with an experimental model of Java 5³ and models for OFBiz’ DSLs⁴. The tool is described in detail in Paper B (Chapter 9).

Tengi is a domain-specific language and framework to interrelate heterogeneous textual and visual languages via interfaces. It is available online⁵ and Paper C (Chapter 10) describes it in detail.

TexMo is a prototype of a multi-language development environment, which relies on a universal language representation and an explicit relation model to interrelate heterogeneous development artifacts. It is available online⁶. The tool is described in detail in Paper D (Chapter 11).

Coral is a prototype extending the Eclipse IDE into a multi-language development environment. It relies on per language representations and a search-based relation model to interrelate heterogeneous development artifacts. Coral includes a domain-specific language to declare cross-language relations as constraints and a tool to infer cross-language relation constraint libraries from heterogeneous source code in textual languages. The tool is available online⁷ and Paper F (Chapter 13) describes it in detail.

1. <http://www.itu.dk/~ropf/download/tengja.zip>

2. <http://www.itu.dk/~ropf/download/dk.itu.sdg.tengsl.depgen.ofbiz.generator3.zip>

3. <http://www.itu.dk/~ropf/download/JaMoITU.zip>

4. http://www.itu.dk/~ropf/download/OFBiz9_04_DSLs.zip

5. <http://www.itu.dk/~ropf/download/tengi.zip>

6. <http://www.itu.dk/~ropf/download/texmo.zip>

7. <http://www.itu.dk/~ropf/coral.html>

Lässig is a tool prototype for language independent traceability. It adds traceability to all programs compiled to and executed on the Java Virtual Machine. The tool generates aspects instrumenting the JVM out of meta-models and automatically populates a trace model. It is available online⁸. The tool is described in detail in Paper G (Chapter 14).

1.1.3 Additional Contributions

Software Ecosystems Paper

Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wąsowski, and Steven She: “*Variability Mechanisms in Software Ecosystems: Closed versus Open Platforms*”, Under submission (Appendix A)

In the paper we study and compare mechanisms for handling variability in five successful software ecosystems. We investigate closed platforms like eCos and Linux in which variability is managed centrally and open platforms like Debian, Eclipse, and Android, which foster distributed free markets of assets for customization. We investigate the underlying mechanisms that sustain success and growth of these two classes of ecosystems. Our findings are systematized into a framework for comparison and design of software systems with a variability mechanism.

I contributed the information about the Eclipse ecosystem, for which I constructed a tool inferring the interaction of Eclipse bundles out of existing Eclipse installations.

NLP Model Parser

The NLP model parser⁹ is a tool prototype, a set of Eclipse plugins, combining parsing of Microsoft Word¹⁰ files into Eclipse Modeling Framework (EMF) models, with natural language processing provided by the Stanford Parser¹¹. Our NLP model parser allows to parse Word files containing English text into models containing structural information of text documents, e.g., information about headlines, paragraphs, etc., and linguistic information about the document’s contents, i.e., information about the types of words, such as, nouns, verbs, adjectives, and their references.

The NLP model parser is constructed in collaboration with Jan Reimann (TU Dresden). After integrating our NLP model parser in Coral or a similar model-driven tool chain, we plan an evaluation, in which the NLP model parser is applied to development of a multi-language software system specifying its requirements in Word documents. The goal is to measure the amount and quality of automatically established relations between a software system’s requirements, and models and source code implementing the system. We plan to compare the results to other solutions in this field.

8. <http://www.itu.dk/~ropf/laessig.zip>

9. <https://github.com/DevBoost/EMFText-Zoo/tree/master/BreedingStation/NLP>

10. <http://office.microsoft.com/en-us/word/>

11. <http://www-nlp.stanford.edu/software/lex-parser.shtml>

1.2 Outline

The remainder of the document is structured as follows. Chapter 2 motivates my research by providing analogies to examples outside the field of software engineering. Chapter 3 introduces the reader to the terminology used in this thesis. Also, it harmonizes notions that might differ between some of the publications (Papers **A** to **G**). Related research is presented and discussed in Chapter 4. Subsequently, Chapter 5 formalizes the informal problem description of the first two chapters. It states theses, goals, and research questions. Chapter 6 introduces the research methods used for the research in Papers **A** to **G**. Additionally, each paper is summarized, its contributions with respect to the goals and research questions from Chapter 5 are discussed, and the applied methodology is stated. Conclusions and future work are discussed in Chapter 7. All research papers are collected in Chapters 8 to 14 and in Appendix A.

My work is driven by the desire to enhance the state of the art in environments and tools for software development. Therefore, this thesis investigates how to provide better support to develop large multi-language software systems. I am motivated by my experience as software developer and the observation that the tools, in particular the development environments, which we currently employ for development of software systems do not appropriately support developers. I believe that we develop large multi-language software systems with tools, which in their architecture and the offered features only aid development of single language software systems in the style of the seventies.

This work focuses on researching *a)* characteristics of multi-language software systems, *b)* enhanced support for developers of multi-language software systems, and *c)* technological foundations and techniques that allow for enhanced development support. The following sections detail the motivation further along these lines. I will use illustrative examples characterizing contemporary software systems, how developers work on them, and what enhanced development tools should offer to developers.

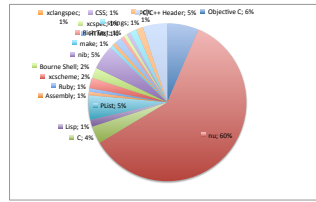
2.1 Contemporary Software Systems – Multi-language Software Systems are Real

Contemporary software systems are large and heterogeneous. They are constructed out of a multitude of artifacts in a multitude of different languages. Development artifacts are all files that are created or edited during development of a software system. For example, files containing source code in different languages, configuration files, system documentation, etc. are all development artifacts. Artifacts are expressed in languages ranging from natural languages over domain-specific languages (DSLs) to general-purpose languages (GPLs). In short, contemporary software systems are multi-language software system.

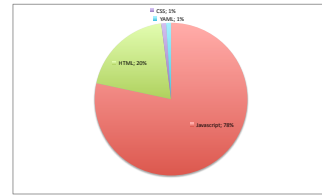
For example, around one third of developers using the Eclipse IDE work with C/C++, JavaScript, and PHP besides Java and a fifth of them use Python besides Java [2]. PHP developers regularly use one to two languages besides PHP [1].

Figure 2.1 summarizes the language composition of the twelve most *interesting* projects on GitHub¹. A larger version of Figure 2.1 with better readable

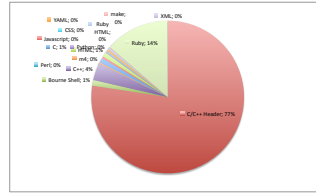
1. <https://github.com> counted on January 16th 2013



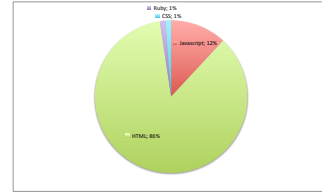
(a) *Nu* <https://github.com/timburks/nu>



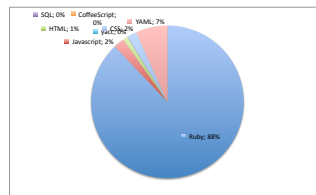
(b) *Prototype* <https://github.com/sstephenson/prototype>



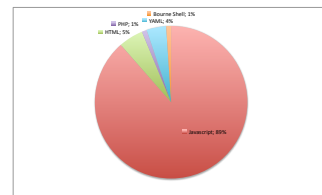
(c) *Passenger* <https://github.com/FooBarWidget/passenger>



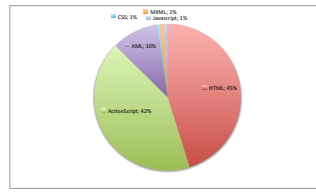
(d) *Scriptaculous* <https://github.com/madrobby/scriptaculous>



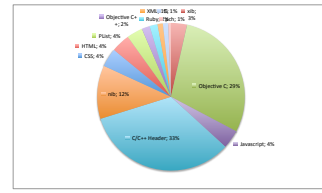
(e) *Rails* <https://github.com/rails/rails>



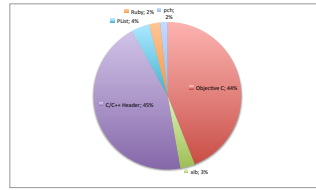
(f) *mootools-core* <https://github.com/mootools/mootools-core>



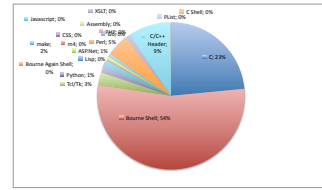
(g) *Restfulx* <https://github.com/dima/restfulx>



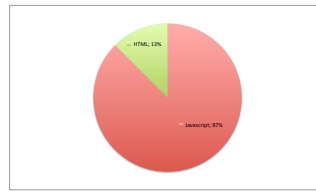
(h) *GitX* <https://github.com/pieter/gitx>



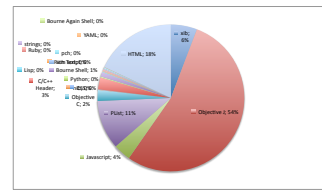
(i) *asi-http-request* <https://github.com/pokeb/asi-http-request>



(j) *Git* <https://github.com/git/git>



(k) *Raphael* <https://github.com/DmitryBaranovskiy/raphael>



(l) *Cappuccino* <https://github.com/cappuccino/cappuccino>

Figure 2.1: The twelve most interesting projects on GitHub and their language composition

language names can be found in Figures B.1 to B.3 in Appendix B. Note, *interesting* is a category on the GitHub computed out of user preferences and interests. The projects contain everything from two to at least 19 languages.

The language compositions are counted with the *cloc* tool². The numbers of reported languages are under approximations of the actual amount of languages, as *cloc* only counts languages declared in a configuration file. Obviously, all these systems are multi-language systems.

Complex large enterprise systems consist of even more languages. For the following four Enterprise Resource Planning (ERP) and e-commerce systems, I count all languages in their code bases. The code base of *OFBiz*³, an industrial quality open-source ERP system, contains more than 30 languages including GPLs, several XML-based DSLs, configuration files, properties files, and build scripts. *ADempiere*⁴, another industrial quality ERP system, uses 19 languages. The e-commerce systems *Magento*⁵ and *X-Cart*⁶ utilize 12 and 10 languages respectively.

Software systems constructed utilizing the model-driven development paradigm are likely to consist of even more languages. They additionally rely on languages for model management, e.g., metamodeling (UML, Ecore, KM3, etc.) model transformation (QVT, ATL, ETL, Xtend, etc.), code generation (Acceleo, XPand, etc.), and model validation (OCL, EVL, etc.)⁷.

Another well-known example of a large multi-language software system is the Linux kernel. Despite the common belief that it is a C project, the Linux kernel consists of more than 20 languages⁸. Additionally, it contains a large variability model in the KConfig language. The model is used for configuration and it is strongly interrelated with the kernel source code [84].

Not only that we construct software systems out of a multitude of languages. All these languages have varying characteristics. They vary in the kinds of concrete syntax (textual or visual), in the level of abstraction they aim to represent, and certain languages are tied to certain development phases. For example, natural languages and visual languages are usually utilized in early development phases, whereas later development phases rely more on textual programming languages.

2.2 Software System Development – The Confusion of Languages

Despite that development artifacts may have different characteristics with respect to language, abstraction, development phase, etc. they all together constitute a single whole. Each artifact provides a different view on a software system. To form a single coherent system, development artifacts are composed, which introduces various kinds of relations between fragments of artifact or between entire artifacts. Artifacts or their fragments either directly refer to each other or they refer to the same aspect of a system. Some of these relations may be explicit. For example, source code in a programming language usually contains explicit references to other software components in the same language, see for example the class references in Listing 2.3. Other relations may be implicit. For example, visual models and code generated from them are both

2. <http://cloc.sourceforge.net>

3. <http://ofbiz.apache.org>

4. <http://www.adempiere.com>

5. <http://www.magentocommerce.com>

6. <http://www.x-cart.com>

7. See <http://uml.org>, <http://eclipse.org/modeling/emf>, <http://wiki.eclipse.org/KM3>, <http://www.omg.org/spec/QVT>, <http://www.eclipse.org/atl>, <http://www.eclipse.org/epsilon/doc/etl>, <http://www.eclipse.org/xtend>, <http://www.eclipse.org/acceleo>, <http://wiki.eclipse.org/XPand>, <http://www.omg.org/spec/OCL>, <http://www.eclipse.org/epsilon/doc/evl> respectively.

8. Cloc count of linux-3.7.9 <http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.7.9.tar.bz2>



Figure 2.2: A head and a foot of a person



Figure 2.3: A modified version of the contents of Figure 2.2

descriptions of the same system aspect at different levels of abstraction, but their detailed relation remains hidden in a code generator. Some artifact relations can even remain completely undocumented, stored only in human memory. For instance, requirements documents are sometimes directly translated to source code without recording any traces between the corresponding artifacts.

From a developer's perspective, software systems are huge compositions or conglomerates of development artifacts. But only a fragment of the artifacts are in focus of developers during modification, customization, or evolution of a software system. Due to the nature of tools used for software development, developers usually have a quite narrow and task centric view on a system under construction. To appropriately support development of multi-language software systems a more holistic view on the entire system, or at least a more holistic view of interrelated development artifacts, is desirable.

I illustrate the developer perspective and the missing holistic view on a constructed system with an analogy to modern art. Figure 2.2 shows the head and a foot of a person. In a software system these would correspond to two development artifacts in different languages. A developer needs to change one of them to perform a certain task. Due to an incomplete view of the system, she might not know about the relation of the displayed head and foot. Is there a relation at all? Do both body parts belong to the same person? Is it a person performing a somersault? Consequently, the developer could modify the system as depicted in Figure 2.3, where the foot is flipped horizontally.

With the support of current development tools she cannot judge easily the feasibility, the validity, or the impact of the applied modification with respect to the entire system. This is similar to changing a fragment of a Java class or a fragment of a model in a software system without complete understanding of the artifacts and their context. See sections 2.2.1 and 2.2.2, which detail the problem of interrelated development artifacts on two exemplary multi-language software systems.

A more holistic view enables a developer to judge the illegitimate modification. It informs a developer about the kind and existence of a relation between head and foot. See Figure 2.4 to the right. Obviously, head and foot are in relation, they are parts of the same person, and flipping the foot would "break" the person.

Similarly to the example, environments for development of multi-language software system, lack comprehensive support that provide developers with a holistic view of a constructed system allowing to judge the effects of applied modifications. Furthermore, they lack support to guide developers to modify development artifacts while ensuring the well-formedness of the entire system

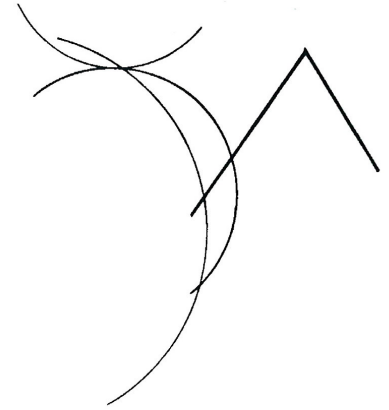


Figure 2.4: Wassily Kandinsky,
Dance Curves: The Dances of
Palucca 1926 [18]

at development time.

Relations between heterogeneous development artifacts are numerous and they are problematic for software developers:

- Since cross-language relations are usually implicit, they require substantial insider knowledge for a developer to correctly perform simple development tasks, such as development of new features or modification of exiting systems. Substantial insider knowledge means for example, knowledge of architecture, diverse languages, application frameworks, etc. Furthermore, it requires knowledge of those language constructs that are potentially referring to other constructs or that are referred by constructs in other languages.
- Errors caused by broken relations between development artifacts are most often only exposed at runtime. Detection of any errors requires thorough testing of the modified code, while at least errors caused by broken relations between static artifacts, could be revealed at development time.

While constructing software systems, developers continuously have to reason about such relations and to navigate along relations between heterogeneous development artifacts. This calls for investigating language oblivious tools, which support developers of multi-language software systems better than current tools. In particular, visualization, static checking, and navigation of relations across languages and development artifacts could already be valuable mechanisms supporting developers as indicated by the example above.

Provision of appropriate tools for multi-language software system development does not only render the work of single developers more convenient. Instead, there is a global concern to consider. Maintenance and enhancement of software systems is expensive and time consuming. Between 85% to 90% of project budgets go to legacy system operation and maintenance [37]. Lientz et. al. [83] state that 75% to 80% of system and programming resources are used for enhancement and maintenance, where alone understanding of the system stands for 50% to 90% percent of these costs [117]. Provision of multi-language

Listing 2.1: A fragment of a properties file (left) and a fragment of Java login logic (right)

```

1 login.title = JTrac Login
2 login.home = Home
3 login.loginName = Login Name / email ID
4 login.password = Password
5 login.rememberMe = remember me
6 login.submit = Submit
7 login.error = Bad Credentials

```

```

1 private class LoginForm extends StatelessForm {
2     private String loginName;
3     private String password;
4     public String getLoginName() {
5         return loginName;
6     }
7     public String getPassword() {
8         return password;
9     }

```

Listing 2.2: A fragment of the HTML code describing JTrac's login page

```

1 <table class="jtrac">
2 <tr>
3 <td class="label"><wicket:message key="login.loginName" /></td>
4 <td colspan="2"><input wicket:id="loginName" size="35"/></td>
5 </tr>
6 <tr>
7 <td class="label"><wicket:message key="login.password" /></td>
8 <td colspan="2"><input type="password" wicket:id="password" size="20"/></td>
9 <td align="right">
10 <input type="submit" wicket:message="value login.submit" />
11 </td>
12 </tr>

```

Figure 2.5: Three source code fragments of JTrac's login page

software system development tools could contribute to reduce costs of system understanding, maintenance, and enhancement.

However, the following sections and my research demonstrates, that designing such tools is challenging. The challenge lies in the tension between the generic and the specific. Heterogeneous artifacts, and even more so relations between them, are often domain-specific. That is, they are intrinsically hard to support with generic tools.

The following two sections introduce the two multi-language software systems JTrac and OFBiz. In these sections, the illustrative example of interrelated development artifacts (Figure 2.4) is carried over to the realm of software engineering highlighting some of the problems when developing multi-language software systems.

2.2.1 JTrac – A Java Web-application for Issue-tracking

JTrac is an open-source web-based bug-tracking system. JTrac's code base consists of 374 files. The majority of files, 291, contain source code in Java (141), HTML (65), property files (32), XML (16), JavaScript (8), and 29 other source code files such as Shell scripts, XSLT transformations, etc. The remaining 83 files are images and a single jar file. JTrac is clearly a medium-size multi-language software system.

Similar to many other web-applications, JTrac implements the Model-View-Controller (MVC) pattern. This is achieved using popular frameworks, such as the persistence framework *Hibernate*⁹ for OR-Mapping and web-development framework *Wicket*¹⁰ to couple views and controller code.

To illustrate the complexity of development of multi-language software system from a developer's perspective, let's consider an example extracted from JTrac. JTrac's login page (Figure 2.5) is implemented in three source code artifacts in three different languages. The login page itself is described in HTML

9. <http://hibernate.org>

10. <https://cwiki.apache.org/WICKET>

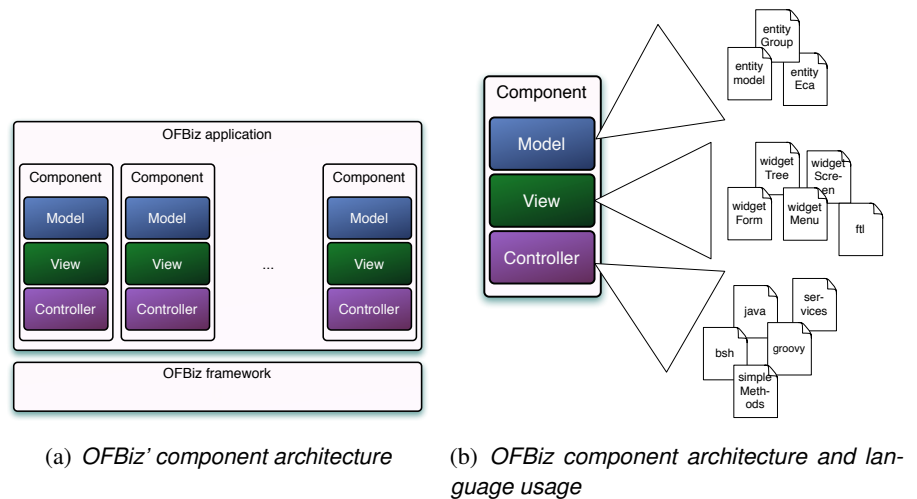


Figure 2.6: OFBiz' architecture in a nutshell

(Listing 2.2), displayed messages are given in a properties file (Listing 2.2.1 to the left), and the logic evaluating a login is described in Java (Listing 2.2.1 to the right). The HTML code specifies the structure of the login page and its contents, i.e., the order or basic layout of the input fields for login and password. Since JTrac is built using the web-development framework Wicket, the HTML code contains wicket identifiers, which serve as anchors for string generation or behavior triggering, see lines 3, 4, 7, 8, and 10 in Listing 2.2. The properties provide certain messages for the login page. For instance, the property on line 3 in Listing 2.2.1 provides the message string for line 3 of the HTML code. The Java code (Listing 2.2.1 to the right) implements authentication logic. To focus on the example, the actual authentication logic is not displayed. In order, to correctly invoke the Java code, the corresponding accessor methods (lines 4 and 7), and mutator methods (not shown), must use a concatenation of get or set and the capitalized name of the wicket identifiers on lines 4 and 8 in Listing 2.2 respectively.

Now, imagine a developer renaming the string literal `login.loginName` on line 3 in the HTML code to `login.loginID`. Obviously, the relation between the properties file (line 3) and the HTML file is now broken. In effect, the message asking for a login name is not displayed correctly anymore. The mistake is only visible at runtime. Observe that such small quiet changes of behavior can easily be missed by testers. Similarly, renaming the string literal `loginName` on line 4 in Listing 2.2 to `loginID` breaks a relation to the accessor `getLoginName` in the Java file. The effect of this change is even more serious. JTrac crashes with an error page.

Obviously, a more holistic view on JTrac incorporating the cross-language relations¹¹ (blue lines in Figure 2.5) could support developers in understanding the impact of their actions when working on interrelated artifacts.

2.2.2 OFBiz – The Apache Open for Business Project

Open For Business (OFBiz) is an industrial-strength open-source enterprise automation software project. Typical use cases for OFBiz are implementation of ERP systems and online shops.

OFBiz is a large multi-language software system. Its code base is approximately

11. The explicit cross-language relations in the example of interrelated artifacts in JTrac, the blue lines in Figure 2.5, correspond to the dance curves in Figure 2.4.

```

1 <entity entity --name='FinAccount' " package--name="org.ofbiz.accounting.finaccount"
2   title ="Financial Account Entity"> <field name='finAccountld' " type="id--ne"></field>
3   <field name="finAccountTypeld" type="id"></field>
4   <field name="statusld" type="id"></field>
5   ...
6 </entity>

1 public static boolean validatePin(GenericDelegator delegator, String finAccountld, String pinNumber) {
2   GenericValue finAccount = null;
3   try {
4     finAccount = delegator.findByPrimaryKey("FinAccount ", UtilMisc.toMap('finAccountld ", finAccountld))
5   } ...

```

Figure 2.7: Two cross-language cross-component relations in OFBiz

300MByte large. It contains development artifacts in more than 30 DSLs and GPLs. Further, it consists of nearly 5000 development artifacts. Unlike JTrac, OFBiz is a component-based software system. Its basic distribution contains currently around 30 coarse-grained components. OFBiz itself provides a framework for running OFBiz applications. For convenience, I concentrate on OFBiz applications and consider such applications as separate software systems. Figure 2.6(a) depicts the component-based architecture of OFBiz applications. Each OFBiz component follows the MVC design pattern. Figure 2.6(b) sketches how different languages are used in the MVC architecture. The language names are not detailed any further, see Paper B for more details. Here, the only important fact is, that OFBiz applications are implemented using multiple different languages.

Figure 2.7 shows an example of two relations between a development artifact containing Java code validating a personal identification number and the data model described in OFBiz' XML-based DSL for data model. Both artifacts reside in different components. OFBiz development artifacts are heavily interrelated by relations linking string literals (see Paper B). For example, the highlighted arguments of a method in the Java code (Figure 2.7, line 4) refer to the corresponding names in the data model (lines 1 and 2). Such relations between development artifacts may cross language boundaries and the boundaries of the MVC architecture. Even the boundaries of OFBiz components may be crossed by such relations. This is illustrated in Figure 2.8. The red lines denote relations between development artifacts across the different MVC architecture levels, within and across components. The artifacts are not explicitly depicted. Obviously, it exists an implicit relation graph interrelating development artifacts. Such implicit relations are problematic as they are not described explicitly. Again, imagine a developer renaming `finAccountld` into `finAccountIdentifier`. At

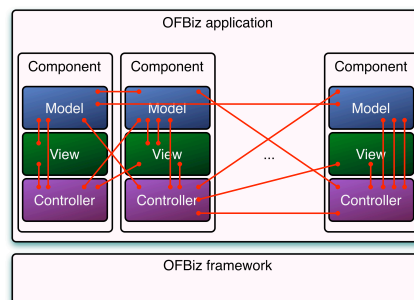


Figure 2.8: OFBiz component architecture and implicit relations (red lines)

development time, there is no feedback informing the developer that not only a cross-language relation between artifacts is broken but also a relation crossing component boundaries is broken. Clearly, such implicit relations hinder customization and evolution of multi-language software systems and they hinder replacement of components in component-based systems.

Development tools should support developers of multi-language software system in their work by provision of a more holistic view on OFBiz, i.e., incorporating cross-language relations (blue lines in Figure 2.7) appropriately. As the example of JTrac and OFBiz development illustrate, enhanced development tools should generically support multi-language software system development as the problem of interrelated artifacts is independent of any concrete domain. Only the type of relations depends on the domain.

2.3 Contemporary Development Environments – Taming the Confusion of Languages

In this section I motivate my research from a tool builder perspective, using illustrative examples.

As described above, contemporary development environments do not appropriately support development of multi-language software systems. From a tool builder perspective, the reason for this lack of support, is the way integrated development environments (IDEs) represent heterogeneous development artifacts. Usually, IDEs are parametrized with plugins of language development tools to support new languages. These plugins contain separate editors, parsers, and optional compilers, debuggers, etc. This is similar to word processors, which are parametrized with dictionaries and grammar checking rules for various natural languages. The illustrative analogy of IDEs for development of software systems and word processors for writing of documents in natural language is utilized in the following. Contemporary IDEs are comparable to word processors. The latter usually support writers with spell checking and grammar checking when writing text in a single natural language. For example, consider the following two sentences¹²:

John left. He said he was ill.

Figure 2.9: Two interrelated English sentences

These sentences are in correct English. Since there is no violation of orthographical and grammatical rules, neither a word processor's spell checker nor grammar checker detect any errors. In focus of this thesis in the analogy from a tool builder's view, is: How do the spell checker and the grammar checker work? What concepts do they need to implement?

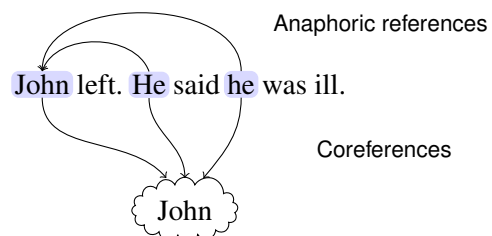


Figure 2.10: Two correctly interrelated English sentences and the type of relations

As illustrated before, an important concept for a spell and grammar checker

12. The example is adapted from plato.stanford.edu/entries/anaphora on anaphora.

are relations. In particular, relations between multiple sentences. Figure 2.10, depicts the relations between the two sentences. There are two types of relations between the noun “John” and the two pronouns “he” (highlighted words in Figure 2.10). First, these three words are coreferences to the same *thing*, the concept of a person or the precise person *John*. Additionally, the pronouns “he” are anaphoric references. They are back-references to the noun “John”. Word processors integrating modern natural language processing facilities [19] can automatically establish and check coreferences and anaphoric references.

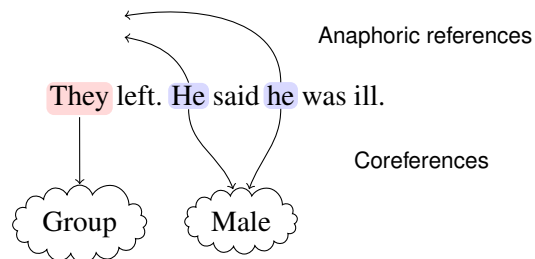


Figure 2.11: Two incorrectly related English sentences and the type of relations

What word processors usually cannot check is evolution of interrelated sentences. Think of modifying the noun “John” to the pronoun “They”, as in Figure 2.11. Both sentences on their own are still correct. But the previously established relations are broken, since the pronouns “They” and “he” refer to different concepts and the anaphoric references do not point to any noun anymore. A word processor could mark the sentences and let the writer know that two previously related sentences are now unrelated.

Listing 2.3: Two correctly interrelated Java classes

```

1 package humans;
2 import java.util.List;
3
4 public class Person {
5     public String name;
6     ...
7 }
8
9 public class Group {
10     private List<Person> members = new ArrayList<Person>();
11     ...
12 }

```

As this thesis is about software development, the natural language examples are used for illustration only. In the realm of software development using contemporary IDEs and development tools, the previous natural language examples can be transferred to the example of two interrelated Java classes `Person` and `Group`, see Listing 2.3.

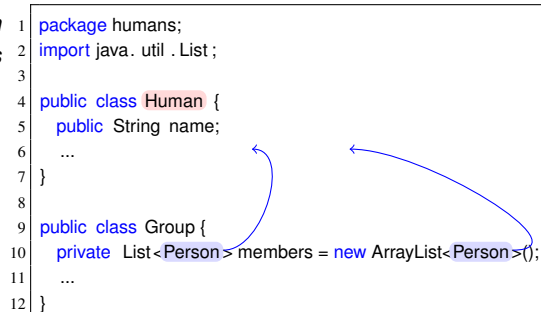
This example illustrates two correctly interrelated Java classes. Each class is the equivalent of an English sentence in a software development scenario. Obviously, the attribute `members` in class `Group` relies on two correct references to the class `Person` (line 10). Similar to modification of “John” to “They” in Figure 2.11, the relations are broken when the class `Person` is renamed, for example, into `Human`, see Listing 2.4. Contemporary IDEs statically check such relations by relying on abstract syntax trees and classpath information. Broken relations are reported via error messages to developers. Contemporary IDEs in combination with language development tools are good in supporting developers in such single language settings. That is, there is not much to

Listing 2.4: Two Java classes with broken relations

```

1 package humans;
2 import java.util . List ;
3
4 public class Human {
5     public String name;
6     ...
7 }
8
9 public class Group {
10     private List<Person> members = new ArrayList<Person>();
11     ...
12 }

```



research.

But what about if we switch to a polyglot setting? In the analogy to word processors, it is about writing documents with interrelated sentences in various languages. Figure 2.12 depicts the same example as in Figure 2.10, but now the second sentence is in Danish (literal translation of “He said he was ill.”).

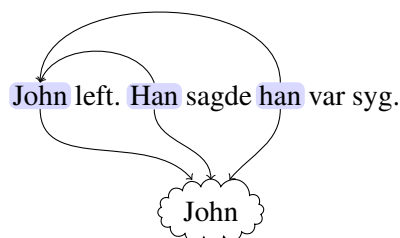


Figure 2.12: Two correctly interrelated sentences in English and Danish

Obviously, the two sentences contain the same relations apparent to a reader literate in both languages. Unfortunately, contemporary word processors do not provide grammar checking across language boundaries. Thus, modification of the noun “John” to “They” breaks the relations just as in Figure 2.11, but no grammar checker can provide any useful feedback to the writer. This example transferred to the domain of software development highlights the lack of multi-language support in contemporary IDEs and development tools. Figure 2.13 illustrates another excerpt of a Java class in JTrac describing the logic of a login page (Listing 2.5) and an excerpt of an HTML file describing the login page itself (Listing 2.6). There are two relations between the Java and the HTML code between the marked fragments. I do not describe them in more detail here, see Paper F a detailed description. I call such relations in software development *cross-language relations*, see the following chapter (Section 3) for more detailed definitions of such relations.

As demonstrated before, the cross-language relations between Java and HTML code are broken when renaming either relation end to something other than title or form respectively. Generally, contemporary IDEs do not support developers in such a scenario. They do not statically check for correct cross-language relations and they do not let developers navigate such relations. The problem in general is: contemporary IDEs do not integrate various languages, their editors, and tools with each other. From a tool builder perspective, the motivating question is now, how to represent the interrelated languages and the relations themselves to be able to build tools, which support developers in multi-language settings?

Listing 2.5: An excerpt of a Java class describing a login page of a web-application

```

1 public class LoginPage {
2     private static final Logger logger = ...
3
4     public LoginPage() {
5         setVersioned(false);
6         add(new IndividualHeadPanel().setRenderBodyOnly(true));
7         add(new Label("title ", getLocalizer().getString("login.title", null)));
8         add(new LoginForm("form "));
9         String jtracVersion = JtracApplication.get().getJtrac().getReleaseVersion();
10        add(new Label("version", jtracVersion));
11    }
12    ...
13 }

```

Listing 2.6: HTML code excerpt describing a login page

```

1 <html>
2 <head>
3   < title wicket:id="title "></title>
4   <link rel="stylesheet" type="text/css" href="resources/jtrac.css"/>
5   <link rel="shortcut icon" type="image/x-icon" href="favicon.ico"/>
6 </head>
7 <body>
8   ...
9   <form wicket:id="form " class="content">
10     ...
11   </form>
12   ...
13 </body>
14 </html>

```

Figure 2.13: Interrelated multi-language source code in Java and in HTML

*“Oh, look what the good lord for us has done
each day he gives us the rising sun.
You don’t have to find yourself sunlight
or the moon and stars that guides your way at night.
So, the best things in life ’s for free,
I say the best things in life ’s for free.”*

The Heptones, The Best Things in Life

The following sections introduce necessary terms and concepts to understand and relate the contributions of the research papers (Papers **A** to **G**). Furthermore, the given terms harmonize notions across the publications. This chapter contains already parts of the contributions of this dissertation that need to be introduced here to allow for discussion of the papers themselves. For example, Section 3.3 and Section 3.4 summarize contents of Paper **F** to introduce terminology required in the following chapters.

3.1 Characteristics of Development Artifacts

Generally, a software system consists of a number of separate programs, configuration files, documentation, etc. [113]. All of these are usually stored in various files. I uniformly call such files **(software) development artifacts** or just **artifacts**.

Definition 1 (Software Development Artifact)

Software development artifacts *are all files, which are created, edited, or modified by humans or machines with the purpose to develop, customize, or modify a software system. Such files may contain source code, models, plain text, images, etc. A collection of development artifacts is **heterogeneous** if they are instances of different languages (see Definition 2).*

The term **mogram** [75] denotes a similar, but a little more restricted notion of development artifacts. Mograms are just models and programs with no further distinction [76]. Especially, in Papers **D** to **F**, I use the term mogram.

In my thesis I use a very broad definition of language. I consider any development artifact, such as text files, program sources, models, etc. as sentences of languages.

Definition 2 (Language)

*A **language** is a set of sentences. Each **sentence** is a collection of symbols, where **symbols** are usually alphanumerical characters.*

Sentences can be fragmented. **Fragments** are just sequences of symbols in a sentence.

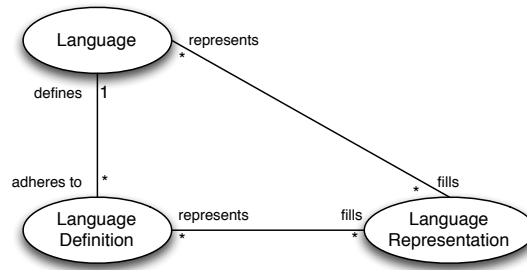


Figure 3.1: The concepts of language, language definition, language representation, and their relations

Definition 3 (Language Definition)

*A **language definition** is a formal way to specify which sentences belong to a language.*

Usually, language definitions are given by formal grammars, which explicitly specify which sentences belong to a language. However, I consider any computer program that parses development artifacts as language definitions, since such programs implicitly specify the set of sentences belonging to a language.

In this thesis I work with abstractions of languages as I want to work with development artifacts in different languages generically. So, the central concept to tackle the research questions stated in Section 5 is **abstraction** of development artifacts and languages to more abstract representations. In fact, my thesis is strongly influenced by the credo “*Everything is a model*” [24] when representing languages and development artifacts in multi-language software systems.

Definition 4 (Language Representation)

*A **language representation** is a data structure specifying the set of abstract concepts of languages and their relations.*

In general, I distinguish two groups of language representations. These are **lexical** and **syntactic language representations**. A **lexical language representation**, represents any development artifact of any language as a stream of characters. Whereas, **syntactic language representation**, relies on data structure like trees and graphs to describe concepts and their relations. Often, metamodels are used for specification of syntactic language representations.

Obviously, the concepts *language*, *language definition*, and *language representation* are not independent of each other. Figure 3.1 illustrates the relation of the concepts, saying that each language has multiple language definitions and multiple language representations. On the other hand, any language definition defines exactly one language but any language representation potentially represents many languages. Note, Figure 3.1 does not depict a metamodel, but just an illustration for ontological disambiguation.

The relation between language definition and language representation is also described by others [5, 76]. In both works, the authors state that grammars and metamodels can be considered as equivalent concepts as they can be mapped to each other. Actually, the authors describe, that out of grammar rules metamodel concepts can be generated, usually resulting in quite concrete language representations. Furthermore, out of metamodel concepts, grammar rules can be generated under incorporation of further information, for example, the used symbols. The last constraint demonstrates, that language representations are a

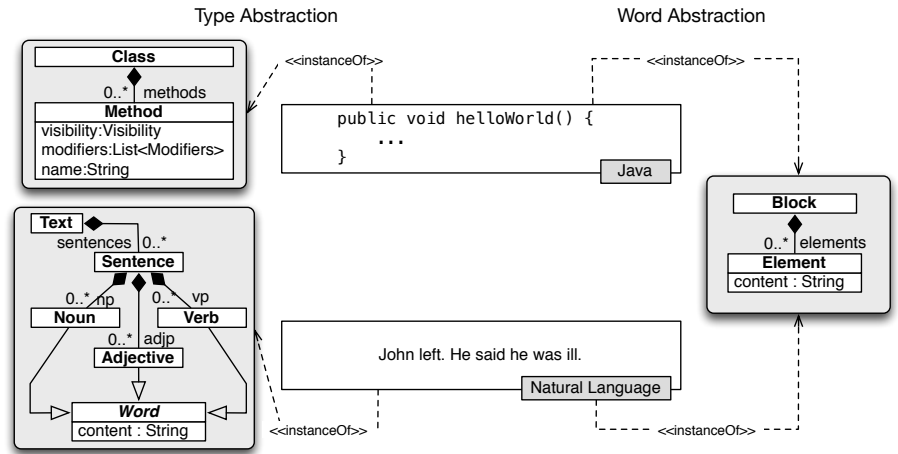


Figure 3.2: Type abstraction and word abstraction, two orthogonal abstraction mechanisms

“weaker” concept compared to language definitions. Due to information loss in the process of abstraction, language representations cannot always generate valid sentences in the language they represent. Language definitions are needed to populate a language representation.

Even though very interesting, the question if there exist languages without description and representations is not discussed here. It is out of scope of this dissertation.

Obviously, there exist many languages. On computers interaction with development artifacts is based on tools, such as editors and the like. All these tools encode language definitions rendering any artifact on a computer a sentence of a language. Additionally, depending on the tool processing an artifact, development artifacts can be instances of many languages. For example, a Java 5 program is also a Java 6 program. Independently of tools, development artifacts can also be represented in many ways. For example, a development artifact containing a program in Java 5 can be represented as instance of the MoDisco Java 5 model [28], as instance of the JaMoPP Java 5 model [60], or as instance of my Java 5 model (Paper B). All three models are different representations of the same language.

In general, there exist even more languages. In computer sciences it is often distinguished between **natural** and **formal languages**. **Natural languages** are languages like English, Danish, etc. They are used by people to communicate and they evolve, by people informally agreeing on a languages structure. Precisely, natural languages are not described formally. In contrast, **formal languages**, are artificial languages defined by formal language definitions and language representations, such as Java 5 in the example above.

Only formal languages are of interest in this thesis. Thus, in the remainder, I use the terms *language* and *formal language* synonymously. However, even natural languages, can be represented formally by sufficiently abstract language representations. This is illustrated in the following example.

Consider the example of Java and natural language in Figure 3.2 (centered). A Java method declaration has a visibility, modifiers, a name, and it resides in a class (not illustrated in concrete syntax). This domain knowledge is captured in the example Java language representation (top left). A natural language text however consists of sentences, which in turn consists of nouns, verbs, and adjectives. The simple English language representation in Figure 3.2 (bottom left) keeps this domain knowledge. On the other hand, Java method

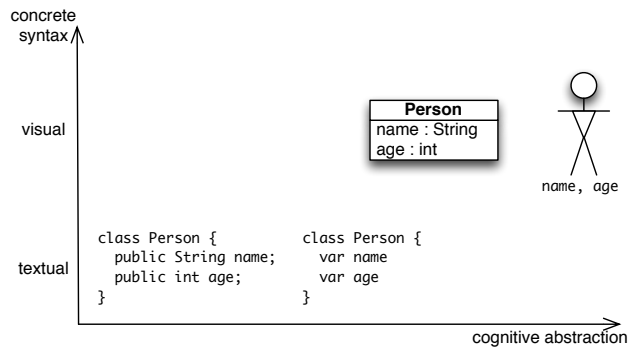


Figure 3.3: The same language concept in different concrete syntaxes mapped to their level of cognitive abstraction (Figure extended from [65])

declarations and the natural language sentences can be represented by a common model capturing the structure of both artifacts but neglecting their language specific differences, see Figure 3.2 to the right. This example illustrates, that development artifacts may be instances of many languages and that there may be different language representations for them. Furthermore, it demonstrates that two orthogonal abstraction mechanisms exist. First, *type abstraction* [130], also referred as *ontological metamodeling* [15] or *logical metamodeling* [14], and second, *word abstraction* [130], also referred as *linguistic metamodeling* [15] or *physical metamodeling* [14]. Type abstraction is a unifying abstraction which describes domain concepts along with their properties, whereas word abstraction is a simplifying abstraction, describing structures of sentences or structures of sequences of symbols. According to Colburn [31], the fundamental difference of both abstraction types lies, in relying on *content* or on *form* for abstraction. Any of the two abstractions can be applied at the same time to create any type of language representation. Obviously, in type abstraction a Java method and natural language sentences are distinguishable by their corresponding language concepts, whereas in the more generic word abstraction this information is lost. Abstraction of arbitrary languages into language representations is a key concept applied in this thesis. Abstraction is a powerful tool as it allows to build generic tools integrating diverse languages with each other.

The previous example illustrates an *informal* criterion for language characterization. The **abstraction level** of a language. The level of abstraction informally groups languages into **more concrete** and **more abstract languages**. In a common understanding, more concrete languages encode more information and more details compared to more abstract languages. For example, in Figure 3.2 the block language (right) is more abstract compared to the two languages on the left.

Another example for languages on different levels of abstraction are UML class diagrams and Java. UML class diagrams are more abstract than classes in Java. The abstraction levels of UML and Java illustrate, that in software development more abstract languages are usually used in early development phases, whereas more concrete languages are usually used in later phases. Furthermore, in model-driven development, transformation chains formed by code generators and code transformations, most often transform artifacts in more abstract languages into artifacts in more concrete languages.

```

1  /**
2   * Sets the passed Model the specified slot. Removes the model currently set to the specified slot.
3   *
4   * @param slotName
5   *   The slot's name.
6   * @param value
7   *   object to store in the slot (set <code>null</code> to remove the slot)
8   */
9  void set(String slotName, Object value);

```

Figure 3.4: Javadoc, an embedded DSL. Source: the interface `WorkflowContext.java` in package `org.eclipse.emf.mwe.core` of the Eclipse bundle with the same name

In addition to level of abstraction, languages can be categorized¹ by their appearance, i.e., by their concrete syntax.

Definition 5 (Concrete Syntax Types)

A language with **visual concrete syntax** uses visual symbols, such as shapes, lines, etc. to represent language concepts. Additionally, alpha numeric characters may be used. A language with **textual concrete syntax** solely uses alphanumerical characters to represent language concepts.

For brevity, I call languages according to their concrete syntax either visual or textual languages. Visual languages occur frequently in software engineering. They are usually used for high-level problem description and clarification of structures and behaviour. Often, visual languages are not formally defined. For example, until recently the Unified Modeling Language (UML) was defined in natural language documents only. Other languages are defined formally, especially those implemented in tools running on computers.

Especially in Paper A, I claim that visual languages do not actually exist on computers. Even if tools present development artifacts in visual concrete syntax, these artifacts are always persisted in textual concrete syntax. That is, visual concrete syntaxes are only visualizations, i.e., rendered representations, of textual languages. Consequently, also for visual languages my definitions of *language* (Definition 2) and *development artifact* (Definition 1) apply.

Interestingly, a form of abstraction, **cognitive abstraction**, is applied in creation of concrete syntaxes too. For example, Figure 3.3 illustrates the same language concept, a person on one level of abstraction, in four different concrete syntaxes. Illustrated on bottom of Figure 3.3 are two textual concrete syntaxes for the concept *person* in concrete syntax of Java and Scala respectively. On top of the same figure, are two visual concrete syntaxes in UML class diagram syntax and a custom visual syntax. It seems, that in software engineering, more abstract languages have concrete syntaxes which are more cognitively abstract. However, cognitive abstraction is an informal characteristic of concrete syntaxes and strongly individual.

Formal languages can be further divided into domain-specific languages (DSL) and general-purpose languages (GPL).

Definition 6 (Language Types)

Domain-specific languages (DSL) are languages tailored to a particular domain. DSLs are often not Turing complete. That is, they may not be usable

1. The categorization is somewhat informal and may be debated philosophically. Think for example, of ancient Egyptian hieroglyphics. Are hieroglyphs characters in visual or textual languages? Similarly, Chinese characters (Han characters) or Japanese characters (Katakana characters), are on the border between visual and textual languages, from a European point of view. However, I do not detail the discussion here.

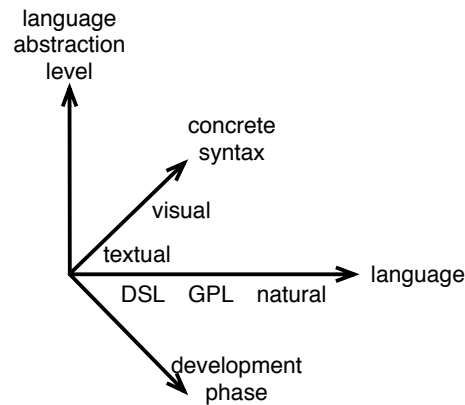


Figure 3.5: Space spanned by properties of development artifacts

to describe general computation problems, as they most typically lack control structures like loops, conditions, etc. In contrast to DSLs, **General-purpose languages (GPL)** are Turing complete languages, that can describe general computation problems.

Note, most GPLs have textual concrete syntax. GPLs with visual concrete syntax exist but they are quite rare. For example, LabView’s G language² is a GPL with visual concrete syntax.

Languages can be *mixed* together by *embedding* one language into another. For example, Javadoc is an embedded DSL in Java. Listing 3.4 shows an example of Javadoc code on a method declaration within an interface. For example, @param is a keyword of the DSL stating method parameters. Javadoc is an embedded DSL since it extends means of the Java Language, namely comments and annotations, for its definition. Interestingly, in the example, Javadoc contains fragments of HTML code; <code> and </code> are HTML tags. Here, HTML is a DSL embedded into Javadoc, which in turn is embedded into Java.

Domain-specific languages can be categorized into **internal** and **external** DSLs [115, 49]. *Internal DSLs* are languages which are embedded into a host language. They reuse mechanisms of the host language for its definition. The terms internal and embedded DSL are often used synonymously. *External DSLs* are languages that are specified without a host language. Separate dedicated tools, e.g., parsers, interpreters, etc., are used to process them. For this thesis, the categorization of languages into internal and external DSLs and the categorization in DSLs and GPL is not central. The tools and research papers forming this thesis rely only on the fact development artifacts can be represented by dedicated language representations. The language representations for development artifacts possibly containing internal DSL code, is likely independent of the host language anyways. Thus, the language representation is responsible for representing fragments of an internal DSL to a sufficient extend. The latter is elaborated further, especially in Paper F in the discussion about language representations.

The information given in this section allows to define a four-dimensional space, of development artifacts, see Figure 3.5. This space is spanned by the dimensions for level of language abstraction, language type, concrete syntax, and development phase. Development artifacts are points in this space corresponding to their properties and their usage in a software development process.

2. www.ni.com/labview

First, development artifacts contain sources in a certain language ranging from domain-specific over general-purpose to natural languages. All these languages may differ in their concrete syntax, i.e., they are visual or textual languages. All development artifacts describe system aspects on a different level of abstraction, placing them on different levels of language abstraction. Lastly, they may be tied to different development phases.

3.2 The Internal Structure of Software Systems

Modern software systems grow larger and larger. They deal with more and more complex tasks and they often reuse legacy systems, frameworks, etc. *Ultra-Large-Scale Software Systems* [41] or *Systems of Systems* are predicted to appear in future.

I define the size of a software system by the number of development artifacts present at development time. **Medium-size** software systems are composed out of multiple hundreds of development artifacts. **Large-scale** software systems are composed out of multiple thousands of development artifacts. In this thesis I address development of medium-size and large-scale multi-language software systems.

The times of developing large software systems using a single or only few languages are long over. Regardless of the used language and the level of abstraction, contemporary systems are constructed out of development artifacts containing requirements documents, documentation documents, models, source code, etc. Furthermore, in multi-language software systems problems are usually solved by deploying appropriate languages for certain tasks. For example, contemporary systems utilize different languages for database definition and database access, for specification of business logic and for definition of views on data. As they grow over time, legacy systems often evolve into large-scale multi-language software system. They get adapted to new use cases, new technological environments, or are simply customized by wrapping ever-new layers of development artifacts in new languages around a legacy base.

Definition 7 (Multi-language Software Systems)

Multi-language software systems are software systems, which are composed out of development artifacts in various languages. Each language may be used for a special purpose and in a different phase of development.

Software systems can be defined more formally with respect to the space for development artifacts described in Section 3.1. Software systems are planes in the four-dimensional space (Figure 3.5), given by language, language abstraction, concrete syntax, and development phase, since only one language is used for their development. Thus, multi-language software systems are hypercubes in the same four-dimensional space. They are defined by development artifacts and their relations, which are all laid out in this space.

3.3 Relations between Development Artifacts

As demonstrated by the examples of JTrac and OFBiz in Section 2.2, software systems are not only collections of development artifacts. Instead, they are collections of *interrelated* development artifacts. Correct relations are key to

run a software system with the expected behavior.

Definition 8 (Relation)

*Two fragments of distinct development artifacts are in **relation** to each other as soon as an **operation** during development either requires the presence of both fragments or the operation produces one fragment by relying on the other.*

In the definition above I refer to operations performed manually by humans, automatically by machines, or semi-automatically by both of them. For example, a developer needs a UML use case diagram and an API documentation to produce a certain piece of source code. The UML use case diagram and the API documentation are in relation to each other, since the developers performed an operation reusing both artifacts. Furthermore, the source code, the UML use case and the API documentation are in relation to each other as the latter are used to produce the resulting source code.

My definition of relations between development artifacts is somewhat similar to the one given by Salay et al. [109]. They define relations between models based on model semantics. Two models are in relation if the interpretation of one model constrains the possible interpretations of the other one. In my definition these interpretations are externalized into the operations requiring the presence of artifacts or their fragments.

Relations between development artifacts can be formal or informal, depending on availability of the operation causing relations between development artifacts.

Definition 9 (Formality of Relations)

Formal Relations are relations between development artifacts which are caused by a formal execution of the operation causing the relations. So, either a machine in combination with a program performing the operation is available or the operation is executed by humans in a documented process.

Informal Relations are non-formal relations. That is, the operation interrelating development artifacts is not formally available. It only exists in human interpretation.

Informal relations are weak, as the perception of when artifacts are in relation is biased by individuals inspecting the corresponding development artifacts.

On top of formality, relations between development artifacts can be explicit or implicit.

Definition 10 (Explicit and Implicit Relations)

Explicit relations are relations described explicitly either within a development artifact or in a separate development artifact. **Implicit relations** are non-explicit relations. They are established by execution of computer programs or by manual operation application, but they are not directly accessible to developers or other tools.

For example, consider a model transformation generating Java classes out of a UML class diagram. The relations between the UML classes and Java classes are formal implicit relations, as no artifact explicitly represents relations between corresponding classes. However, the relations are formally specified by the transformation itself but they are hidden in the transformation code. Assume, the transformation is not available anymore. Then the UML and Java classes

are in informal implicit relation. Of course the artifacts are still in relation but the relations are now subject to human interpretation. On the other hand think of the same model transformation additionally generating a relation model linking each UML class with the corresponding Java class. The relations in the generated relation model are formal explicit relations, when both transformation and relation model are available. The UML and the Java classes are in informal explicit relation when the UML class diagram the generated source code and the relation model are readily available but the transformation itself is not available anymore. Implicit relations are not important to execute a software system. But the availability of explicit relations is important for developers working on or evolving a software system.

Relations between development artifacts have orthogonal characteristics depending on the abstraction level of the used languages. Development artifacts are in a **view relation** if they describe the same aspect of a software system from a different perspective at the same level of abstraction. Whereas, development artifacts are in **refinement relation** if they are concerned about the same system aspect on different levels of abstraction. Recall, development artifact relations are caused by operations applied to them. For example, model transformations are automatic operations setting development artifacts in relation. The property of orthogonality is also observed by Mens et al. [88] in their taxonomy of model transformations. Amongst others, the authors identify horizontal and vertical model transformations. Both transformations are characterized by the level of abstraction of the transformed models. Models on the same level of abstraction are transformed by horizontal transformations and models on different levels of abstraction are transformed by vertical transformations.

For example, UML class diagrams and UML activity diagrams may be in view relation, as they may describe the same aspect of a system from a different perspective. They are a structural and a behavioral description of a system aspect³. View relations are horizontal relations with respect to the development space (Figure 3.5). During the development process, development artifacts usually refine each other. For example, a requirements document in natural language is refined to a design document, containing UML class diagrams. A design document in turn is refined to source code in a GPL. Here, refinement itself is the operation setting the artifacts into a refinement relation. Thus, refinement relations are vertical relations with respect to language abstraction. Refinements of development artifacts can be generated manually by humans or automatically by transformation and generation programs. Manual refinements are problematic in software development, since the relations from artifacts on high abstraction level to concrete artifacts are left to human interpretation. Such relations are especially prone to complete misunderstanding.

Chronologically ordered relations between development artifacts are called **traces**.

Definition 11 (Traceability)

Traceability refers to the capability for tracing artifacts chronologically along a set of chained operations. [100]

Note, traces are enriched relations. They can be formal or informal too.

The characteristics of relations between development artifact discussed so far,

3. Paper C provides an example for a view relation between a model and a UML activity diagram.

reflect the big picture of development of software systems. However, all of these relations are caused by fragments of development artifacts which are in relation. Precisely, fragments of development artifact are in relation to each other, due to some operation during software development, which requires the presence of the fragments or produces one out of the other. I refer to the fragments linked by relations as relation ends. I observe the following three fundamental types for relations. They are described in more detail in Paper D, Paper E, and Paper F.

Definition 12 (Basic Relation Types)

*A relation between two fragments f and g in distinct development artifacts is a **fixed relation**, if $f = g$. It is a **string-transformation relation**, if the two fragments are similar, i.e., if there exists a transformation T , so that $f = T(g)$ and T is not the identity function. It is a **free relation**, if the two fragments are diverse, i.e., if the relation is neither a fixed nor a string-transformation relation.*

Note, this does not mean that all identical fragments of various development artifacts are necessarily in relation to each other. Only if an operation during software development requires the presence of the fragments or produces one out of the other, they are in relation.

Fixed relations occur frequently in practice. For example, a link in HTML requires an anchor name and a link tag name to be identical. Otherwise, the link is broken. HTML links are an example of fixed relations within the same language. Recall Figure 2.13, it shows an example of two fixed relations across language boundaries between the identical fragments title and form respectively.

String-transformation relations are also quite common. For example, the Wicket framework requires certain identifiers in HTML files to have an accessor and a mutator method (get- and a set-method) in a corresponding Java class, see Paper E for a more detailed description. In Figure 2.5, the Wicket identifier loginName (Listing 2.2 line 4) requires a method with the name getLoginName and setLoginName in the corresponding Java class (Listing 2.1, lines 4 and 7). Depending on the direction, a string-transformation relation either attaches or removes get/set and capitalizes or decapitalizes loginName.

Fixed and string-transformation relations can be broken by modifying one relation end without modifying the opposite relation end accordingly. I sometimes refer to *broken relations* as *dangling references*.

On top of these basic relation types that reflect physical properties of artifact fragments, there may be many more relation types. I consider them domain-specific and discuss other possible relation types in related work (Section 4.3) and in Paper F. However, domain-specific relation types are not in focus of my thesis as I am interested in fundamental generic support of multi-language software system development.

Correctly interrelated development artifacts, i.e., relations that are not broken, are necessary to let computers execute software systems as expected. That is, compilers and interpreters need correctly interrelated development artifacts to run a program. In generative and transformative settings more abstract development artifacts need to be correctly interrelated since code generators and transformers serve as compilers in such settings. Frameworks and other programs present at runtime can impose further constraints on how development artifact need to be interrelated, see Paper E.

As indicated above, relations between development artifacts may or may not

Listing 3.1: An explicit relation model corresponding to Figure 2.13

```

1 RelationModel {
2   Artifact "/" jtrac /src/main/java/info / jtrac /wicket/LoginPage.html" {
3     keys A, B;
4   }
5   Artifact "/" jtrac /src/main/java/info / jtrac /wicket/LoginPage.java" {
6     references C, D;
7   }
8
9   Key "A" </ jtrac /src/main/java/info / jtrac /wicket/LoginPage.html> {
10    [" // @document/@webpagebody/@head/@items.0/@parameter.0/@value"]
11  }
12  Key "B" </ jtrac /src/main/java/info / jtrac /wicket/LoginPage.html> {
13    [" // @document/@webpagebody/@body/@tag.2/@parameter.1/@value"]
14  }
15
16  Reference "C" </jtrac /src/main/java/info / jtrac /wicket/LoginPage.java> {
17    [" // @class.0/constructor.0/@expressionstatement.2/@newconstructorcall/@expression/@stringreference"]
18  }
19  Reference "D" </jtrac /src/main/java/info / jtrac /wicket/LoginPage.java> {
20    [" // @class.0/constructor.0/@expressionstatement.3/@newconstructorcall/@expression/@stringreference"]
21  }
22
23  Relation A <- C [FIXED]
24  Relation B <- D [FIXED]
25 }

```

cross language boundaries. That gives rise to the following two relation categories:

Definition 13 (Language Relation Categories)

Intra-language relations are relations between development artifacts of the same language. *Cross-language relations (CLR)* are relations between development artifacts of the different languages.

As if the four-dimensional space for development artifacts were not enough, the previous section detailed, that all artifacts in a software system do not exist on their own. Instead, they are interrelated. The stress field between development artifacts in all their variety and different relations amongst them creates the *Confusion of Languages* in software development. Multi-language Software System developers suffer from the confusion of languages similarly to biblical notions illustrated in Figure 1. Provision of appropriate tools for development of multi-language software systems will improve their situation. So in this thesis, I focus on how tools can support multi-language software system development with arbitrary languages and various relations between development artifacts.

3.3.1 Relation Models – Explicit Relation Representation

As discussed previously, relations between development artifacts are usually not described explicitly. This is problematic as developers are not aware of potential relations. Furthermore, tools supporting development of multi-language software systems benefit from explicit relations, as they can exploit them to provide feedback to developers. For example, in form of static checking and visualization of relations between development artifacts.

In this section, I introduce four techniques to explicitly represent relations between artifacts (Definition 10). The four techniques are a result of a literature survey in Papers **D** and **F**. Their identification and the given examples are already contributions of my thesis. However, they are introduced here to facilitate comprehension of Chapter 6.

Listing 3.2: An excerpt of a Java class with link tags

```

1 public class LoginPage {
2     private static final Logger logger = ...
3
4     public LoginPage() {
5         setVersioned(false);
6         add(new IndividualHeadPanel().setRenderBodyOnly(true));
7         add(new Label(@link(in("../LoginPage.html"), target(wicket:title)),
8             getLocalizer().getString("login.title", null)));
9         add(new LoginForm(@link(in("../LoginPage.html"), target(wicket:form))));
10        String jtracVersion = JtracApplication.get().getJtrac().getReleaseVersion();
11        add(new Label("version", jtracVersion));
12    }
13    ...
14 }

```

Listing 3.3: An excerpt of HTML code with relation anchor tags

```

1 <html>
2 <head>
3     <title @anchor(wicket:title)></title>
4     <link rel="stylesheet" type="text/css" href="resources/jtrac.css"/>
5     <link rel="shortcut icon" type="image/x-icon" href="favicon.ico"/>
6 </head>
7 <body>
8     ...
9     <form @anchor(wicket:form) class="content">
10        ...
11    </form>
12    ...
13 </body>
14 </html>

```

Figure 3.6: Multi-language source code in Java and HTML (corresponding to Figure 2.13) interrelated via tags

Definition 14 (Explicit Relation Model)

An **explicit relation model** is an artifact, which contains explicit links interrelating fragments of various development artifacts.

An explicit relation model can be seen as a graph, whose edges are the relations and whose vertices are the relation ends in development artifacts. Actually, an explicit relation model resembles a lot the illustration in Figure 2.13. The blue lines are an explicit relation model with two cross-language relations connecting the highlighted relation ends. Listing 3.1 illustrates a possible explicit relation model in a textual concrete syntax. The relation model is described in more detail in Paper D. It contains two fixed relations (lines 23 and 24) between two relation ends (references and keys) respectively. Note, the relation ends in this model are named (A, B and C, D) and they refer to the development artifacts containing them (lines 9 to 11, 12 to 14, 16 to 18, and 19 to 21).

Paper F and Section 4.2.1 discuss, that explicit relation models are, especially in model-driven development, often utilized to represent relations between model elements.

Alternatively, explicit relation models can be represented by tags, similar to HTML link tags. In HTML, link tags can be used to specify relations between fragments of other HTML documents or entire artifacts. Such kind of tags are conceivable for non-hypertext systems too.

Definition 15 (Tags)

A **tag-based relation model** marks interrelated fragments directly within heterogeneous development artifacts. Relations are expressed by link tags which refer to anchor tags.

Listing 3.6 illustrates an exemplary explicit relation model based on tags. The

Listing 3.4: A Tengi interface corresponding to LoginPage.java

```
1 Tengi LoginLogic ENTITY "LoginPage.java" [
2   IN: { loginTitleHTML, loginFormHTML }; CONSTRAINT: loginTitleHTML & loginFormHTML;
3   OUT: { loginTitleJava, loginFormJava }; CONSTRAINT: loginTitleJava & loginFormJava;
4 ]
5 LOCATOR loginTitleJava IN "LoginPage.java" OFFSET 198 LENGTH 5;
6 LOCATOR loginFormJava design IN "LoginPage.html" OFFSET 278 LENGTH 4;
7 }
```

Listing 3.5: A Tengi interface corresponding to LoginPage.html

```
1 Tengi LoginView ENTITY "LoginPage.html" [
2   IN: { loginTitleJava, loginFormJava }; CONSTRAINT: loginTitleJava & loginFormJava;
3   OUT: { loginTitleHTML, loginFormHTML }; CONSTRAINT: loginTitleHTML & loginFormHTML;
4 ]
5 LOCATOR loginTitleHTML IN "LoginPage.html" OFFSET 27 LENGTH 17;
6 LOCATOR loginFormHTML design IN "LoginPage.html" OFFSET 244 LENGTH 16;
7 }
```

Figure 3.7: Interfaces interrelating the Java and in HTML code from Figure 2.13

example is based on Figure 2.13. Obviously, the artifacts are modified to store anchor tags (@anchor) in HTML sources and link tags (@link) in the Java sources. Link tags specify relations to the corresponding opposite relation ends marked with anchor tags.

Fragments of development artifacts can also be interrelated, by explicitly specifying relation ends and their relations in interfaces. Imagine interfaces as tagged relation ends, as in tag-based relation models, which decoupled from the corresponding development artifacts.

Definition 16 (Interfaces)

Interface-based relation models explicitly define fragments and their relations in interfaces. Interfaces are separate artifacts accompanying interrelated development artifacts.

Listing 3.7 illustrates two interfaces for the interrelated Java and HTML sources of Figure 2.13. The interfaces are expressed in the Tengi interface DSL, see Paper C. Tengi interfaces define relation ends in corresponding development artifacts (ENTITY) as ports (LOCATOR). Out-ports (OUT) specify which relation ends are provided to the environment and in-ports (IN) specify which relation ends are required from the environment. Constraints (CONSTRAINT) specify how development artifacts are in relation to each other.

Unlike the three relation models presented so far, which directly refer to relation ends, relations can also be specified based on search queries. Search queries refer relation ends indirectly. They need to be evaluated before relations between concrete relation ends are established.

Definition 17 (Search-based Relation Models)

Search-based relation models represent relations between fragments of development artifacts via queries locating fragments and constraints between the query results, describing the relations themselves. Only after query and constraint evaluation, relation instances are established.

Listing 3.6 illustrates a search-based relation model. It is expressed in the Coral DSL (Paper F), which allows for specification of constraints for cross-language relations.

The listing illustrates a relation between a string reference in Java and a parameter in HTML (lines 9 to 15). The actual constraint is implemented in Groovy.

Listing 3.6: A search-based relation model interrelating the Java and HTML sources from Figure 2.13

```

1 java {
2   StringReference is org.emfext.language.java.references.impl.StringReferenceImpl;
3 }
4
5 html {
6   StringValParameter is html.impl.StringValParameterImpl;
7 }
8
9 fixed : StringReference::value in java <—> StringValParameter::value in html with
   wicketIDsInJavaConstructors
10 is info display "Wicket IDs in Java constructor call ." implementation "
11   def constructorCallContainer = leftHand.getConstructorContainer()
12   if (rightHand.name.equals("wicket:id") && constructorCallContainer instanceof NewConstructorCall &&
       leftHand.value.equals(rightHand.value)) {
13     return true
14   }
15   return false"

```

It says that a string reference in Java and a parameter in HTML are in relation as soon as their values are identical and the string reference in Java appears in a constructor call. Obviously, in search-based relation models, relations between artifacts are specified on metalevel. Evaluation of the cross-language relation constraint (line 9) establishes the two relations illustrated in Figure 2.13.

Languages for Specification of Relation Ends. Different languages can be utilized to identify relation ends. Based on the examples for the four different relation models discussed above, I observe three different kinds of such languages.

Physical Navigation The interface-based relation model in Figure 3.7 specifies relation ends by locating fragments via an offset and length in a stream of characters. In that case development artifacts are in a lexical language representation.

Path Navigation The explicit relation model in Listing 3.1 utilizes uniform resource identifiers (URIs) to specify relation ends in development artifacts. Development artifacts with syntactic language representations allow to specify of relation ends by path expressions navigating the data structure of the language representation.

Query Evaluation The search-based relation model in Listing 3.6 specifies relations via queries and constraints. The queries and constraints require syntactic language representations, as they encode relations on top of structural and type information of the language representation.

Obviously, the language for specification of relation ends is influenced by the language representation. Please refer to Paper F for more detailed information on language representations and representation of relations between development artifacts.

Inference of Relation Models. Relations and relation models do not necessarily need to be specified manually. Instead, they can be inferred automatically or semi-automatically. The inference may either rely on the static properties of a system, i.e., its development artifacts, or on its dynamic behavior. By querying the development artifacts in a code base together with knowledge about language constructs causing relations between artifacts, relation models can be

inferred out of artifacts themselves. Especially Papers **B** and **F** demonstrate the application of inference of relation models out of development artifacts. Furthermore, if relations are first present at runtime, e.g., trace links, they can be inferred by observing the programs processing development artifacts. That is, relation models can be *inferred by instrumentation of programs*. Papers **A** and **G** apply inference by instrumentation.

3.4 Software Development Tools

Contemporary software systems are most often constructed with the help of one or more software development tools. Software development tools are anything from simple text editors to full-fledged IDEs, which are utilized to edit, modify, or develop artifacts in software systems.

Definition 18 (Integrated Development Environment)

IDEs are software development tools, which integrate development artifact editors with other tools, such as, debuggers, build tools, etc.

One of the fundamental problems addressed in this thesis is that existing IDEs do not directly support development of multi-language software systems. That is, they do not generically integrate editors and software development tools across language boundaries with each other.

Usually, existing software development tools support developers with certain mechanisms to effectively perform their tasks. For example, contemporary IDEs statically check source code and report the results to developers, IDEs let developers navigate source code, or they visualize certain aspects of software systems.

Definition 19 (Support Mechanisms)

Language support mechanisms are all mechanisms provided by software development tools, which support developers constructing software systems in a single language. Cross-language support mechanisms are all language support mechanisms, which are accessible across language boundaries.

Paper **F** and Chapter 4 survey various IDEs, programming editors, and literature to collect a set of contemporary language support mechanisms.

Definition 20 (Multi-language Development Environment (MLDE))

MLDEs are IDEs implementing cross-language support (CLS) mechanisms. That is, MLDEs integrate editors and other language specific tools across language boundaries with each other.

*“Cry tough, don’t you know you’re getting old.
Cry tough, don’t you don’t you know you’re getting slower.
How can a man be tough, tougher than the world tougher than the world?
For if he’s rough, he’s against the world he’s against the world.”*

Alton Ellis, Cry Tough

4

The Design Space of Multi-language Development Environments

In this chapter I survey the state of the art in development of multi-language development environments and in support for development of multi-language software system. There is only few research explicitly targeting multi-language development environments. Therefore, I summarize related literature populating the taxonomy of design choices of multi-language development environments (Paper **F** and Figure 6.5). The sections of this chapter are aligned along the design choices of the taxonomy, such as language representation for development artifacts (Section 4.1), relation model types (Section 4.2), types of relations between artifacts (Section 4.3), inference of relation models (Section 4.4), and cross-language support mechanisms (Section 4.5). In each section I also summarize my contributions (Papers **A** to **G**) with respect to the corresponding design decision or with respect to their novelty resulting of the combination of design choices and their application.

The main contribution of my dissertation with respect to this chapter is the establishment of the design space for multi-language development environments in a taxonomy (see Figure 6.5 and Papers **D** and **F**) and the systematized overview of the related work according to the taxonomy (Section 4.6).

4.1 Language Representation

The taxonomy in Paper **F** contains four language representations falling into two major categories. First, lexical language representations (Section 4.1.1) and second, syntactic language representations (Section 4.1.2 to Section 4.1.4). The following four sections detail the language representations and present corresponding related work.

4.1.1 Lexical Language Representation

A lexical language representation, represents any development artifact of any language as a stream of characters.

Plain text editors usually keep a lexical representation of the edited development artifacts. For example, *SED* [87] and *Emacs* [116] (without language modes enabled) keep a lexical language representation and let users modify a stream of characters. In [122] the notion of a text editor is formalized. Sufrin et al. [122] formally define commands for text editing on top of characters and on top of words and lines. That is, editing commands are formalized on physical properties of a development artifact, on a lexical language representation.

Especially Paper A and Paper C rely on a lexical language representation to locate fragments of development artifacts serving as relation ends. Lexical language representations are useful as their simple structure allows to easily track modifications of development artifacts and thereby adapting relation ends accordingly.

4.1.2 Syntactic per Language Representation

A syntactic per language representation, represents a single language, which is already defined by another mechanism such as a formal specification, a parser, a metamodel, etc. using data structures like trees or graphs.

Using models to represent source code is getting more and more popular. Modern IDEs, such as, *Eclipse*, *IntelliJ*, *VisualStudio*, *NetBeans*, etc. implement separate editors with separate isolated abstract syntax representations for every supported language. A typical IDE provides separate Java, HTML, and XML editors. Most, IDE editors maintain an abstract syntax tree (AST) in memory and automatically synchronize it with modifications applied to concrete syntax. That is, concrete and abstract syntax are in (a projectional) relation. The ASTs are exploited to facilitate source code navigation, refactoring, static checking, etc. on single languages separately.

Language workbenches [45, 46]¹ use models to represent languages and abstract syntax trees. I consider models and abstract syntax trees synonyms, as they are both abstract, per language representations. Per language representation via models is a concept facilitated by emergence of language workbenches. Language workbenches such as *EMFText* [59, 13] and *Xtext* [35, 39] rely on EMF models for language representation. Additionally, for each metamodel representing a language, concrete syntax mapping rules are specified. Parsers, editors, etc. are automatically generated out of the combination of metamodels and concrete syntax mapping rules. The effectiveness of using models as language representations is documented by the large amount of available languages. Wide over 100 are listed in the syntax zoos of the two frameworks *EMFText*² and *Xtext*³. *Spoofax* [74] does not exploit EMF for definition of languages. Instead, it utilizes the syntax definition formalism SDF. Similarly, the *Meta-Programming System MPS* [33, 129], relies on a proprietary language, the Structure Language, to define a language's abstract syntax. However, the concept of using models as per language representations is common to all language workbenches. Please refer to Merkle et al. [89] for an elaborate comparison of language workbenches.

Often, frameworks for refactoring of legacy code exploit per language representations based on models. For example, the *MoDisco* [28] project is a model-driven framework for software modernization and evolution. It represents Java, JSP, and XML source code as EMF models. Each language is represented by

1. Also see www.languageworkbenches.net for the annual language workbench competition.

2. http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo

3. https://bugs.eclipse.org/bugs/show_bug.cgi?id=328477

its own distinct model. The models abstract a legacy software system into models to enable automatic transformation into a new modernized representation. Similarly, the reverse engineering framework *BlueAge* [17] represents legacy COBOL source code as EMF models, so that model transformations can be employed to modernize legacy COBOL systems. The same principle of an abstract model-based representation of a programming language is implemented in *JaMoPP* [60]. *JaMoPP* is an EMF per language representation of Java5. Similarly, *Featherweight Java* [22, 23], is a model-based implementation of a subset of the Java language, which is often utilized to research new language concepts. *JavaML* [16] is a pre-EMF per language representation of Java using XML for a structured representation of Java source code. On the other hand, *SmartEMF* [64] represents XML-based DSLs via EMF models and maps them to a Prolog knowledge base, where the EMF models realize a per language representation.

Especially applications, which integrate tools to allow editing of interrelated heterogeneous artifacts, rely on a per language representation manifested by the respective tools. For example, Meyers [90] discusses how to integrate tools into *multi-view development systems*. In general, integration of heterogeneous development artifacts, i.e., language integration, can be considered a problem of tool integration, since artifacts cannot be edited, modified, compiled, etc. without tools allowing to perform these actions. Meyers describes, amongst others, basic tool integration on file system level. There, each tool keeps a separate internal data representation. That is, each tool is a manifestation of a per language representation. Similarly, the prototype *ToolNet* [7, 48] integrates heterogeneous development artifacts by integrating tools. The authors of *ToolNet* propose a kind of message bus on which registered tools exchange actions applied to development artifacts. By these actions development artifacts which are modified in different tools are interrelated. Each tool registering to the bus manifests a per language representation.

There is more research presenting syntactic per language representations. They are listed in the following in Section 4.1.3 in case the per language representations are utilized to generate a per language group representation.

Contribution: Especially the prototypes GenDeMoG and Coral (Papers **B** and **F**) but also Tengja and Lässig (Papers **A** and **G**) are based on per language representations. In Paper **B**, I utilize per language representations to enable automatic inference of an explicit relation model containing cross-component relations of development artifacts in component-based applications. Similarly, Coral (Paper **F**) requires per language representations to infer relations between development artifacts. In both cases, the choice of per language representations enables the technical experiments investigating relations in multi-language software systems. A reason for the choice of per language representations is, that often they can be easily obtained. For example, in Paper **B** the per language representations of OFBiz' XML-based DSLs are automatically generated out of readily available XML schemas. Other reasons for the choice of per language representations, in particular with respect to feasible features in multi-language development environments, are discussed in Paper **F**.

4.1.3 Syntactic per Language Group Representation

A syntactic per language group representation, represents a group of languages defined by multiple language definitions or represented by multiple per language

representations using data structures like trees or graphs.

The taxonomy in Paper F proposes a syntactic language representation per group. The reason to integrate various languages into a single representation is either to effectively handle languages which are often used in combination (e.g., [53]) or to represent languages, which share a common characteristic (e.g., [68]). The following research utilizes per language group representations without explicitly naming it or without explicitly pointing to it as a major feature.

Some IDEs chose to represent languages as groups. For example, Strein et al. [121, 120] realize that most IDEs do not support analysis of multi-language software systems and that they do not allow to refactor source code across language boundaries. In order to address these issues, the authors present *X-Develop*, an IDE with specific support for development of multi-language software systems. X-Develop integrates heterogeneous programming languages by representing source code uniformly as models. In fact, the metamodel for programming language representation is a syntactic per language group representation integrating various object-oriented and mark-up languages. It is extensible to adapt the X-Develop to new development scenarios, for example development with non-object-oriented programming languages. One reason for a language group representation is to provide refactoring across various programming languages. Unfortunately, X-Develop seems to be no longer available. Jarzabek [70] describes specification and generation of multi-language development environments using interrelated attribute grammars in combination with attributes holding semantic expressions. The language definitions, in form of grammars, together with the relation specifications, in form of attributes, are used to generate a per language group representation in a database and editors working on top of this representation.

Gómez et al. [55] present a tool, which allows to work with visual models with evolving metamodels. The authors utilize their *Generic Intermediate Metamodel (GIMM)* to check models against evolving metamodels and to provide support to developers to facilitate adjusting the models appropriately. GIMM is a language group representation for heterogeneous models that share some basic characteristics, e.g., the models must consist of elements with typed attributes.

The *GUPRO* tool [81], integrates multiple per language representations step-wise into a per language group representation to implement a basic IDE. GUPRO provides a graph-based query mechanism to locate and navigate cross-language relations between heterogeneous development artifacts. GUPRO is intended to especially support the process of understanding multi-language software systems and relations across the languages COBOL, Job Control Language (JCL), and Communicating Sequential Processes (CSP).

LiMonE [111] is an editor for literate programming. Relations between natural language documents and UML models are expressed as constraints in Object Constraint Language (OCL). The constraints are statically checked to support users, for example, when documenting UML models or when creating models out of natural language specifications. Similar to *SmartEMF* [64], LiMonE compiles the heterogeneous development artifacts (natural language and UML) together with the OCL constraints to a Prolog knowledge base and to Prolog rules. To support static checking of cross-language constraints, Prolog rules access and evaluate the knowledge base. In general, Prolog knowledge bases can be considered as a per language group representation. Interestingly, integration

of new languages into LiMonE is challenging as for any new language a new compiler generating Prolog code needs to be manually implemented.

A similar concept can be found in [44]. Ford coins the term *polyglot programming* for development of applications utilizing more special-purpose languages in combination with general-purpose languages. In an interview in [43], he says: “*I define a polyglot solution as languages that run on the same runtime. . . Polyglot solutions for me produce the same bytecode.*”. This denotation suggests a unifying language as criteria for polyglot programming⁴. Here bytecode can be, similar to the Prolog knowledge bases, considered as per language group representation as it integrates heterogeneous languages into a common representation. Apart from that, Ford [44] also emphasizes, that development of multi-language software systems is a matter of fact in industry. Web-applications are named as prominent instances of multi-language software systems and the Java solution stack exemplifies the cause of multi-language software systems. Each framework, such as Hibernate, Spring, etc. comes with their own often XML-based DSLs. According to the author, developers using such frameworks know easily more than ten languages.

Languages on their own can also be represented as groups. That is, without an IDE or similar tool fulfilling a special purpose. For example, Holst [68] presents *Meta*, a language and environment for specification of language families. Languages with similar characteristics are grouped into language families. For example, *Meta(Oopl)* is the language unifying all object-oriented programming languages. and *Meta(Doc)* is the language unifying typesetting languages, such as LaTeX and HTML. Obviously, each *Meta(X)* is a per language group representation as it integrates all languages sharing the characteristic X.

More often, languages which are used together in the same context are represented as groups. For example, the *WebDSL* [53] framework can be utilized for implementation of web-applications using a set of abstract interrelated DSLs. The framework comprises DSLs for specification of data models and webpages. All heterogeneous development artifacts in the corresponding DSLs are represented by a per language group representation, the WebDSL integrating heterogeneous languages, which are used in combination.

Wende [132] presents a solution for language composition based on configuration. It relies on the idea that languages are constructed out of building blocks called components. Such components can be reused and composed with other into languages or larger reusable components. For example, many programming languages implement arithmetic and Boolean expressions. Both are elementary languages which may be reused in other programming languages. Compositions of languages are language group representations, since many per language representations are integrated into a single representation. *Mélusine* [38], a DSL composition framework, follows a different pattern for composition of executable DSLs. It generates a virtual machine (VM) for each DSL and composes DSLs by composing the corresponding VMs. Those DSLs that can be executed on composed VMs are again instances of a per language

4. I do not reuse the term *polyglot programming*. First, it emphasizes programming over the entire process of development of multi-language software systems, and second, the term is defined using *bytecode*, a concrete language, as common language representation. Instead, I refer to *multi-language software system development* as soon as more than one language is utilized for development of a software system. Furthermore, I advocate for the use of abstract languages for language representation. Especially in Paper D and Paper F, I demonstrate that it is likely more convenient to integrate languages by sufficiently abstract generic representations compared to provision of compilers to a concrete language for arbitrary languages.

group representation.

Unlike the two previous solutions for language composition [132, 38], Renggli et al. [106, 105] do not compose language components into a language group representation. Instead, they extend a host language (Smalltalk) with first-class DSL concepts. The various extensions together with the host language form a language group representation.

Island grammars [123] allow to parse development artifacts containing fragments with source code in multiple languages. For example, the authors refer to ASP webserver pages consisting of interleaved source code in HTML, JavaScript, and Visual Basic. Parsing such development artifacts is challenging as fragments in different languages may be hard to distinguish. The authors rely on island grammars, which specify fragments of languages separately in a single grammar and also specify how heterogeneous fragments are distinguishable. The ASTs resulting from parsing with island grammars are instances of per language group representations.

4.1.4 Syntactic Universal Representation

A syntactic universal language representation, represents any language defined by any language definition or represented by any language representation using data structures like trees or graphs.

Research on truly universal language representations is quite scarce as mostly language group representations are suggestive of being universal representations. However, as mentioned previously Meyers [90] discusses tool integration schemes to generate multi-view development systems. He discusses the possibility and desirability of a *canonical* representation, which he describes as some fundamental data structure maintaining the core data common to all development artifacts. Such a data representation for tool integration corresponds to a universal language representation. However, the author argues that it is quite hard to come up with an appropriate universal language representation and that no representation for any arbitrary language has been identified so far.

Contribution: In Papers **D** and **F**, I present with TexMo a universal language representation. The devil’s advocate argues that TexMo’s universal language representation is only a per language group representation for arbitrary textual languages as it uniformly represents text as paragraphs, words, etc. However, remember that I argue in Section 3.1 and in Paper **A** that on computers only textual languages exist. In TexMo, the universal language representation is leveraged to interrelate heterogeneous development artifacts via an explicit relation model. A universal language representation in combination with a relation model allows to implement navigation, static checking, refactoring, etc., i.e., implementation of CLS mechanisms, only once for all languages. Except of TexMo, I am not aware of any IDE implementing a universal language representation.

4.2 Relation Models

A rationale for discussing relation models is given in [88]. The authors present a taxonomy of model transformation. They define support for traceability and change propagation across models as a success criteria for transformation languages and tools. Their notion of transformation tool corresponds to a basic

multi-language development environment. To enable traceability and change propagation relations between fragments of development artifacts need to be represented, stored, or made accessible. The relation models and occurrences in related work are discussed in the following four sections corresponding to the definitions in Section 3.3.1 and the design decisions in the taxonomy in Paper F.

4.2.1 Explicit Relation Model

Explicit relation models seem to be the most natural relation representation from a developer's perspective. Alone the survey by Winkler and Pilgrim [134] reports twelve different explicit relation models for capturing traceability information. In the following, I describe relation models in general not only trace models. Existing explicit relation models are most often tailored to a particular domain but they share a high degree of commonality. They all express relations by dedicated model elements in separate models linking structures or contents of development artifacts.

In different domains and communities different terminology is used for explicit relation models. The most common names are *megamodels* [29, 73], *trace models* [99, 54, 79, 34, 97, 72], or *macromodels* [109].

The notion of megamodels was introduced in by Bézevin et al. [29], who discuss how to integrate models utilized in development of software systems by explicitly linking them with a separate model. The authors argue, that this model should be maintained by the modeling platform to facilitate working with interrelated models across tools. The modeling platform is a collection of interrelated modeling tools. The *AtlanMod MegaModel Management (AM3)*⁵ is such a modeling platform. It allows for specification and maintenance of trace links between heterogeneous models. In [73], Jouault presents a small software system mockup, and he explains how AM3 handles relations between various models exploiting megamodels and weaving models. Megamodels and weaving models maintain relations on two levels of granularity separately. A megamodel describes the physical structure of the software system, i.e., it lists all present models, metamodels, etc. and links them together. Weaving models link model elements from different models with each other. Megamodels and weaving models are both explicit relation models.

Paige et al. [99] argue for the use of domain-specific explicit relation models keeping typed trace links. According to the authors, general-purpose trace metamodels are not always desirable, because illegitimate trace links, with respect to the traced development artifacts, can be created. Furthermore, tools cannot provide rich and elaborate support based on the types of traces, since this information is not encoded in general-purpose trace metamodels. This argument is similar to the argumentation in Paper F. There, I discuss the impact of choice of a per language representation versus a universal language representation in multi-language development environments. The domain-specific trace metamodels in [99] are specified using the Traceability Metamodeling Language (TML) [34], a dedicated language for defining explicit relation models. The author's main idea is to allow to define domain-specific trace types maintaining rich information, which can be exploited by tools.

Similarly, macromodels [109] are a means to specify typed relations between heterogeneous models. Macromodels are used to capture the meaning of such

5. <http://wiki.eclipse.org/AM3>

relations in an explicit model. Salay et al. perceive *submodel of*, *refinement of*, *refactoring of*, *transformations of*, etc. as possible macromodel relation types. Macromodels are most similar to megamodels as they focus on specification of relations between models, but not between model elements.

Guerra et al. [54] present the *PAMOMO* tool. PAMOMO relies on a search-based relation model using triple graph patterns defining traces between models. Whenever the triple graph patterns are evaluated on a set of models an explicit relation model is populated with the relations established by pattern matches. Similar to triple graph patterns, the explicit relation model proposed by Bernstein [21] expresses relations between models based on two morphisms. The morphisms describe the interrelated structures in the respective models separately.

Jouault [72] discusses a solution to automatically populate trace models while executing transformations in the Atlas Transformation Language (ATL). He presents a simple trace model, an explicit relation model, which allows to generically link model elements in arbitrary heterogeneous models with each other. A similar version of this simple trace model appears in [79], where the authors oppose two types of representing model trace links. First, a trace model explicitly keeps trace links in an external model or second, a tag based trace link representation (see Section 4.2.2) is kept. A structurally similar explicit relation model, is the trace model presented by Oldevik et al. [97]. Since the authors describe a model to text transformation language which automatically establishes trace links, the explicit relation model does not contain relations between model elements but between model elements and blocks of text.

Ráth et al. [104] rely on an explicit relation model when synchronizing abstract and concrete syntax of visual models. The explicit relation model links model elements in abstract and concrete syntax with each other. It is exploited by incremental model transformations to reflect modifications applied to model elements in either syntax.

With the *Visual Trace Modeling Language (VTML)*, Mäder et al. [92] present a language for specification and execution of traceability queries. The language allows to specify queries against a traceability information model. A traceability information model is an explicit relation model interrelating all traceable artifacts and their relations of a software system. Interestingly, the authors provide evidence, that VTML allows non-technical users to specify and understand queries on this explicit relation model better than in a general purpose query language.

FeatureMapper [62, 58] is a tool to interrelate feature models with development artifacts having an EMF-based per language representation. The relations between features and fragments of artifacts is maintained in a mapping model. The mapping model is a domain-specific explicit relation model for the domain of feature modeling. Conceptually, it just links model elements with each other.

Contribution: In Papers **E** and **F** I propose the unified notion of *explicit relation model* for all previously presented models which explicitly keep relations between development artifacts. Especially in Paper **B** and Paper **D**, I deploy my own domain-specific explicit relation models linking heterogeneous development artifacts. The two explicit relation models link artifacts represented by different language representations, syntactic per language and universal representation respectively.

I do not develop different explicit relation models for their own sake. Instead, I create them to enable a technical experiment (Paper **B**) and a user experiment (Paper **E**). I am not aware of any previously published experiment, which enables inference of relations between heterogeneous development artifacts in an industrial strength multi-language software system by combining a search-based and an explicit relation model (Paper **B**). Furthermore, I am not aware of the evaluation of CLS mechanisms (Paper **E**), which are enabled by the combination of a universal language representation and an explicit relation model.

4.2.2 Tags

Hypertext systems link fragments of artifacts or complete artifacts with each other via tags. For example, in HTML, links are defined by tags in HTML source code [56]. Hypertext systems interpret tags within artifacts as anchors and links. After interpretation, a relation is established. *HyperPro* [98, 95] is a programming environment which treats development artifacts in a software system as hypertext. That is, development artifacts can be enriched with tags linking their contents across language boundaries. Contrary to *HyperPro*, which uses one text editor to edit heterogeneous development artifacts, *Chimera* [8] is a heterogeneous *Software Development Environment (SDE)* as it integrates different programs like text editors, PDF viewers, and web-browsers to form a development environment. It provides navigation across heterogeneous development artifacts and over application boundaries by relying on hypertext functionality. The programs are viewers through which developers work on different artifacts. Artifacts displayed by one of these viewers are views. Within views anchors can be created. Via links, anchors can be interrelated into a hyperweb. However, after modification of development artifacts links and anchors have to be checked and eventually reestablished by the developer.

DEFT [133] is a development environment for tutorials. Precisely, it is an environment for elucidative development [94]. In elucidative development, the documentation of a software system and its consistency with other development artifacts such as models and source code is of high priority. *DEFT* uses tags to mark regions in natural language documents in which model elements or source code fragments are merged. Furthermore, *DEFT* informs developers to keep the documentation synchronized when interrelated artifacts are modified.

Reuseware [63, 61], is a composition framework for invasive composition. Components encoding concerns are defined separately and they are composed with each other when a system is specified. Both works [63, 61] consider language definitions as components and apply *Reuseware* to extend languages with certain concepts, such as modularization or aspect-orientation. *Reuseware* relies on *slots*, *hooks*, and *anchors*, which are all tags defining variation points. That is, they are referable fragments, which can be filled or replaced with separately defined fragments.

Kolovos et al. [79] discuss two ways of representing trace links between models. Trace links can either be embedded in models themselves, e.g., by marking relation ends via tags, or they can be kept as external separate models. The authors propose to use both representations simultaneously and to merge models and trace links from explicit relation models into a tag-based relation model on user request. The authors reuse UML stereotypes to tag elements in UML models and to establish trace links from merged model elements back to their source models. Interestingly, tag-based and explicit relation models in combination

are also discussed as two possible ways of representing trace links in the survey of model transformation languages by Czarnecki and Helsen [32].

In this thesis, I did not experiment with tag-based relation models. I believe that information about relations across development artifacts should not be merged into artifacts, in the interest of separation of concerns [124]. However, I find the argument of Kolovos et al. [79] for usage of tags convincing: enhanced system understanding. Perhaps emerging tags, similar to emergent interfaces [107], would be a good compromise between merging a concern into a development artifact and explicitly storing it there. An emergent tag would be a tag that is presented to a developer as soon as a fragment with relations to other development artifacts is in focus. The tag itself would not be physically stored in the artifact but computed out of one of the other relation models.

4.2.3 Interfaces

Alfaro and Henzinger [6] define different kinds of interfaces for component-based software development. Informally, they define an interface model to specify what a component expects from its environment. The authors distinguish between static (stateless) and dynamic (stateful) interfaces. Static interfaces are most relevant for my work, as I consider interrelated development artifacts at development time. Dynamic interfaces could be interesting to consider, as soon as dynamic fragments in development artifacts need to be interrelated. For example, dynamic fragments could be source code which is modified or evaluated at runtime and first then the relation to another development artifact is established. However, I believe that this is a special case that can be tackled by interrelating larger parts of development artifacts statically. I do not discuss dynamic interfaces further.

Hessellund and Wąsowski [67] define interfaces for interrelated models and metamodels to explicitly describe relations between models crossing language boundaries. The interfaces are adapted input/output (I/O) interfaces [6]. That is, the authors consider models and metamodels as components and let interfaces describe relations across models as requirements and provisions of model elements. On top of the interface definitions, Hessellund provides a tool, which automatically infers interfaces out of XML models in OFBiz sources. His inference engine is based on heuristics.

OSGi is a module system for Java. It allows to create and execute modular systems [86]. OSGi systems are composed of *bundles*. Bundles aggregate Java source code organized in packages and other non-source code artifacts organized by containing folder hierarchy. Each bundle is defined by an OSGi interface, which for example, declares imported and exported Java packages. OSGi interfaces can be considered interface-based relation models as they specify visibility of packages. That is, OSGi interfaces are more coarse grained compared to the interfaces in [67].

Despite their name, Emacs' [116] tags files are actually interfaces. Tag files store a set of tags pointing to development artifacts or fragments of them. For example, tags point to methods and classes in source code or to chapters and paragraphs in documentation. Tag files do not encode an explicit relation model. Instead, relations are established by users navigating on top of tagged information.

Contribution: In Paper C, I present Tengi, a toolkit including a DSL for specification of interfaces for heterogeneous development artifacts. Similar to [67], these interfaces are input/output interfaces specifying required or provided fragments of an artifact. The Tengi interfaces locate fragments by referring to physical locations (ports) in a lexical language representation. Although interfaces are used in related work to enable interrelation of heterogeneous development artifacts, I am not aware of any previously published work enabling uniform description of interfaces for visual and textual languages based on a lexical language representation.

4.2.4 Search-based Relation Model

Search-based relation models encode relations as queries over development artifacts. Relations and relation ends are established after evaluation of these queries. That is, search-based relation models usually do not provide a persistent representation of relations.

Egyed [36] presents a solution to automatically and efficiently detect and track inconsistencies between models in different UML languages based on consistency rules. These rules consist of queries over two separate models. The query results are set into a Boolean relation. Whenever the Boolean expression, the actual relation specification, evaluates to false, the relation is broken and the two models are in an inconsistent state. Here, the collection of consistency rules defines a search-based relation model.

Hessellund et al. [66] apply code flow analysis to statically check interrelated XML and Java source code. The authors formalize cross-language relations as consistency constraints and check them by traversing the abstract syntax trees of parsed XML files and Java source code. The search-based relation model is formed by the formal consistency constraints querying two distinct per language representations.

Recall *PAMOMO* [54], it utilizes triple graph patterns to define constraints, i.e., relations between models. The tool allows to specify positive and negative patterns. Positive patterns define two conditions, one for each relation end, under which a relation is present. Negative patterns define single constraints for contents forbidden to occur in models. That is, a set of positive patterns manifests a search-based relation model.

Also GPLs are used to express search-based relation models. For example, in *SmartEMF* [64] heterogeneous XML models are compiled to Prolog knowledge bases on which cross-language relation constraints, written as Prolog rules, are executed. The Prolog rules encoding constraints manifest a search-based relation model. Störrle's *VMQL* [119], a visual query language based on graph patterns, also compiles models and queries to Prolog knowledge bases and rules. Interestingly, the author provides evidence that the visual concrete syntax enables domain experts, who are not computer professionals, to formulate queries effectively.

Antkiewicz et al. [10, 9] propose development of framework completion code utilizing *framework-specific modeling languages (FSMLs)*. FSMLs capture framework concepts. For example, framework completion code is code provided by plugins to implement new functionality to a framework. Framework concepts are requirements that framework completion code has to satisfy to produce expected behavior. For example, interfaces, which need to be implemented or methods, which need to be called. Usually, such concepts are given in a

developer documentation. FSML models are used to formally model framework concepts on a higher level of abstraction compared to source code examples in documentation. To map between FSMLs and Java source code, the authors provide a mapping language. The mappings compile to static AspectJ pointcuts querying the Java source code for certain structures like method calls, subclasses, etc. The results of the AspectJ code queries populate FSML instance models representing the framework completion code. Here, the mapping model can be considered a search-based relation model. It queries Java code to establish in memory relations between FSML instances and Java AST nodes.

Search-based relation models rely on a query language or query framework to formulate and execute queries on development artifacts. In the following, I present some query languages applicable in search-based relation models. In the realm of EMF, the *Epsilon Comparison Language (ECL)* [78], *EMF-IncQuery* [20, 57], or *OCL* [108, 69] are prominent query languages. ECL is a high-level rule-based language on top of the Epsilon framework [101, 77]. Based on matching, it allows for specification and identification of pairs of model elements in different models. Similarly, EMF-IncQuery allows to declare queries over EMF models based on graph patterns. Due to incremental evaluation, EMF-IncQuery is well suited for efficient evaluation of complex structural queries over large models. OCL is UML's constraint language. OCL allows to effectively query model, since constraints are formulated on top of query results. The Epsilon Validation Language (EVL) [3] can be considered as a successor of OCL. Paige et al. [99] specify search-based relation models including EVL constraints. In the latter work, search-based relation models populate domain-specific explicit relation models with relations across models.

Contribution: In Paper **B**, I present GenDeMoG a language for specification of search-based relation models interrelating heterogeneous development artifacts. The paper illustrates an example of a search-based relation model expressed in GenDeMoG, which interrelates various of OFBiz' DSLs and Java source code. The relation model utilizes the expression language of Xtend to encode relation constraints. Xtend is a GPL often used for implementation of model transformations. Similarly, in Coral (Paper **F**) the DSL for specification of search-based relation models utilizes a GPL (Groovy) to encode constraints. In both works I decided to research search-based relation models as they are particularly suited in open environments. Software systems are open environments, as the set of development artifacts is not constrained.

Coral, a prototype of a multi-language development environment, can be parametrized with search-based relation models in the form of libraries containing cross-language relation constraints. I am not aware of any other IDE which is similarly customizable with explicit relation specifications to support development of multi-language software systems incorporating various application frameworks.

4.3 Relation Types

Relations between development artifacts do not necessarily have to keep rich type information. For example, they may interrelate fragments and leave developers to interpret the reason of a relations existence [4]. In that case, relations encode merely the existence of relations themselves. Different communities

propose many different possible relation types to encode information to support developers or to enhance tools. For example, Winkler et al. [134] present a taxonomy for traceability models in model-driven software development. A part of this work is an overview over research related to typed trace links. The authors collect many different relation types, which were presented in previous research. The relation types range from coarse-grained trace link types in requirements engineering [114], such as, *dependency*, *refinement*, *evolution*, *satisfiability*, *overlap*, *conflict*, *rationalization*, and *contribution*, to fine-grained technical trace link types [93], such as, *Class-imports-Class* or *MethodInvocation-calls-MethodDefinition*. Both works [99, 34] tackle the problem of heterogeneous trace link types by allowing for specification of domain-specific⁶ trace models. Furthermore, in [99] Paige et al. propose an incremental process for specification of trace link type hierarchies. The authors highlight the importance of typed trace links tailored to specific domains. However, they admit, that classifications of trace link types and trace link type hierarchies are difficult to compare and unify, due to the diverse levels of abstraction for relation types.

Aizenbud-Reshef et al. [4] survey literature and tools on model traceability. The authors abstract current relation models into two types. One for tag-based relation models and another one for explicit relation models. More importantly, they describe the need for differently typed trace links and characterize two distinct categories to encode rich information alongside traces. First, customizable attributes on trace links can be utilized to capture information. Second, special trace link types can be encoded in the trace metamodel, such as, *justifies*, *describes*, *depends on*, etc. However, customizable attributes on trace links allow to maintain enriched information but this information is not comprehensively typed. The solution of letting attributes encode rich knowledge about relations is applied by Nørmark et al. [95]. In *HyperPro*, development artifacts are represented as rich hypertext and relations between them are hyperlinks. Most importantly, rich hypertext allows anchors and links to keep additional meta-information, such as, *link creator*, *link creation date*, etc. in customizable attributes.

Both, Jouault et al. [73], in the work on megamodels, and Salay et al. [109], in the work on macromodels, recognize the importance of typed links. They propose standard link types and they argue to keep an open set of possible relation types, so that domain-specific extensions supporting various traceability scenarios can be accommodated.

Free relations are utilized by Steinberger et al. [118] to let a tool visualize and highlight relations across applications interrelating arbitrary fragments of development artifacts. The interrelated fragments may contain information in visual or textual languages. The authors evaluate the quality of the visualization in a user experiment and highlight its positive impact on users when understanding scattered but interrelated information. Similarly, Waldner et al. [131] discuss visualization of fixed relations across applications and documents. Their tool visualizes relations between occurrences of a search result in different documents, which are viewed in different applications.

Contribution: The previous discussion illustrates, that the modeling community categorizes relation types mostly not with respect to their physical appearance but more conceptually, with respect to what I call domain-specific relation types, see the taxonomy in Figure 6.5. Such relations encode the in-

6. The authors call them case-specific trace models

formation about the reason of existence as type information. However, in the same way as I am applying abstraction to represent languages, I am applying abstraction to represent relations. I believe, that the most useful abstractions for relation types are the physical types *free*, *fixed*, and *string-transformation relations*. These types do not directly encode rich information to support developers. But they allow to construct tools, which effectively support developers working on interrelated development artifacts, see Paper E. The three physical relation types are fundamental for creation of more elaborate domain-specific relation types. This is the reason, why my prototypes directly support free, fixed, or string-transformation relations and why I left their relation models extensible for more specific relation types. I am not aware of any other work formalizing physical relation types as basis for domain-specific relation types.

4.4 Inference of Relations between Development Artifact

Grammel et al. [52] categorize the generation of relation models into two major groups. These are *a)* inference of relation models by instrumentation of programs transforming development artifacts and *b)* inference of relation models solely out of development artifacts, i.e., independently of potential transformation programs. Actually, Grammel et al. focus on trace models. But to me, the same categorization is applicable to relation models in general. That is, I apply these categories to structure the related work on inference of relation models in this section.

4.4.1 Inference by Program Instrumentation

Few programming languages provide first class support for traceability. Often such languages are model transformation languages and they automatically establish trace links between model elements or objects, which are in relation due to a transformation directive. For example, *Epsilon Transformation Language (ETL)* [80] automatically generates a trace model for each model transformation guarded by a post condition. *ATL* [136] establishes a trace models via a similar mechanism. Also the *QVT* [96] transformation language has built-in support for traceability [12]. All three languages are rule-based transformation languages targeting model-to-model transformations. Model-to-text transformations can handle traceability similarly. For example, the *MOF Model-to-Text* transformation language [97] automatically establishes trace links between model elements and fragments in generated files.

Operations interrelating development artifacts can be instrumented by other programs. Thereby, relations are automatically established without modification of the operation. Jouault [72] automatically merges traceability rules into existing ATL transformation rules before execution. Whenever such enhanced transformations are executed, trace models are automatically generated. His solution can be seen as aspect-oriented programming targeting ATL, where the ATL metamodel is the joinpoint model for static weaving.

Grammel et al. [51] infer trace links not by instrumentation of transformations, but by connecting a generic traceability framework to the framework executing transformations. In their solution, a traceability interface abstracts over different transformation languages. The interface has to be implemented via a concrete connector in the corresponding frameworks to allow to automatically establish trace links.

Contribution: Tengja and Lässig (Papers **A** and **G**) rely on aspects, which instrument transformations to automatically establish trace links. In Paper **A**, I present an aspect-based mechanism instrumenting the serialization of visual models. It automatically establishes trace links between model elements, their visual concrete syntax, and textual serialization syntax. Similarly, in Paper **G** I rely on aspects, which are generated out of metamodels, to instrument the Java virtual machine to add traceability support to any programming language compiling to Java bytecode. Lässig generates 100% correct trace models.

It is a new idea to provide language independent traceability by automatic generation of aspects, which instrument transformation programs with traceability directives. I am not aware of a similar traceability solution providing language independent traceability at such a low cost; only a code generator needs to be parametrized with metamodels.

4.4.2 Inference out of Development Artifacts

In the following, I discuss related work for the second category identified by Grammel et al. [52]: inference of relation models solely out of development artifacts independently of transformation programs.

The modeling community often relies on *model matching* [27, 52, 126, 127] to infer relation models. In model matching, models and metamodels, in general object graphs, are processed and compared with each other. Relations are then created automatically, whenever a certain similarity measure between sub-graphs is fulfilled.

The database community applies a similar solution: *schema matching* [103, 112]. Similar to model-matching, database schemas, i.e., graph structures, are compared with each other. In addition to structural analysis, schema matching often incorporates semantic analysis of the schemas.

.

Favre et al. [40] describe a scenario of reverse engineering of a software system written in C and COBOL. They deploy the Obeo Reverse engineering tool, which treats source code as models relying on two per language representations. The authors create a single integrated model for the entire software system consisting of model elements for C and COBOL code respectively. Importantly, the relations between the development artifacts are now explicitly contained in the inferred model. The inferred model of the software system is step-wise transformed to a visual representation of the system, to a new implementation in Java, and to HTML documentation. Here, the relations between C and COBOL code on model level are established based on structural properties of the models of the source code.

Mahé et al. [85] infer megamodels interrelating development artifacts when reverse engineering existing software systems. They apply their solution to the multi-language software system TopCased and demonstrate that its components are quite interrelated. Their tool infers relations by interpreting structures of manifest files, which are XML-based models.

In two case studies Antoniol et al. [11] infer trace links out of source code and natural language documents. The first case study considers development artifacts in C++ and corresponding manual pages. The second case study infers trace links between development artifacts in Java and requirements documents. The authors extract and index information out of heterogeneous development artifacts. Trace links are established whenever a similarity measure between the

extracted information holds. The authors demonstrate that the proposed solution infers correct trace links. However, the inferred trace models are incomplete and the authors propose that the inferred traces serve as initial models, which may be refined by developers.

Hessellund [67] infers a dependency graph out of OFBiz XML models. The inference is based on heuristics encoded in Java programs. Precisely, the heuristics are specifications of structures and contents of XML models, which potentially constitute cross-language references. The inferred dependency graph is transformed into interfaces and metainterfaces for the XML models and their schemas.

Contribution When search-based relation models are used in combination with explicit relation models, the population of the explicit relation models by evaluation of the search queries can be seen as inference of a relation model out of development artifacts. In Paper **B**, I present GenDeMoG a tool, which similarly to [67], infers an explicit relation model out of development artifacts. Unlike Hessellund et al. [67], my tool encodes heuristics about relations explicitly as constraints in a search-based relation model.

Similarly, Coral's search-based relation model (Paper **F**) populates an explicit relation model. Additionally, Paper **F** presents the Coral inference tool, which allows to semi-automatically infer a search-based relation model, i.e., cross-language relation constraints out of given development artifacts. For that, heterogeneous development artifacts are compared with respect to similar contents in their lexical representation. Subsequently, similar fragments are compared with each other based on syntactic per language representations. Fragments matching a similarity measure are then interrelated by generated constraints in a search-based relation model. This two staged inference process can be seen as an extended solution of Hessellund's [67] semi-automatic interface inference mechanism. Compared to [67], my solution enables semi-automatic inference of a relation model independently of a concrete multi-language software system. The only requirement is the availability of corresponding language representations.

4.5 Cross-language Support Mechanisms

In a limited form, CLS mechanisms are sometimes found in contemporary IDEs. Many IDEs can be extended to support new development scenarios. Some development frameworks, such as Hibernate, Spring, Wicket, etc. provide extensions to major IDEs. The extensions contain either specific editors for particular languages or they add certain mechanisms supporting development for the particular framework to the IDE. For example, *QWickie*⁷ is an Eclipse plugin that implements navigation and renaming support between interrelated Java and HTML files containing Wicket code. The *Hibernate Tools*⁸ visualize and propose code completions when working on Hibernate XML and Java files. The *Spring Tool Suite*⁹ provides visualizations of source code targeting the Spring framework. In particular, relations between AspectJ and Java source code are visualized. However, the drawback of all framework-specific tools

7. <http://code.google.com/p/qwickie>

8. <http://www.hibernate.org/subprojects/tools.html>

9. <http://www.springsource.org/sts>

is their limited applicability. They are tailored to a particular domain and cannot be utilized for development with other frameworks relying on the same combination of languages.

The *IntelliJ IDEA* is an IDE, which implements some support mechanisms for multi-language development [71]. It provides code completion and refactorings across particular languages, e.g., for HTML and CSS in combination¹⁰. Or it provides visualization and code completion for certain language combinations, e.g., SQL statements embedded as strings in Java code, which are required by the Spring or Grails frameworks¹¹. However, these mechanisms are provided only for some exclusive language combinations.

In the following I summarize research which could inspire CLS mechanisms in future multi-language development environments. The presented publications indicate the usefulness of certain mechanisms suggesting them for integration into multi-language development environments.

Waldner et al. [131] visualize interrelated fragments of development artifacts across different applications. Interrelated information is highlighted by boxes surrounding fragments. The relations between fragments are explicitly highlighted by visual links. The authors conduct a user experiment evaluating the benefits of explicit visualization of scattered but interrelated information. They conclude that visual links prevent users from cumbersome manual search for interrelated information. Furthermore, they facilitate understanding of scattered information. An extended solution [118] focuses on and optimized visualization for the links themselves. Visual links should not obstruct contextual information. Thus, they are routed around fragments of development artifacts, which possibly contribute to understanding of the relations themselves. The authors conduct an extensive study and conclude that context preserving visual links facilitate understanding of interrelated information even more.

Treemaps are an elaborate visualization mechanisms used for example in the Aspect-Oriented Programming (AOP) community to visualize cross-cutting concerns [42] or the impact of woven advice to source code [102]. Pfeiffer and Gurd [102] encode the relation between advice of aspects and source code as a containment relation in a treemap. They allow developers to navigate the interrelated development artifacts via their visualization. By a user experiment, the authors demonstrate that a visualization of relations between advice of aspects and advised source code significantly improves developer's comprehension of interrelated fragments. However, treemaps are likely only beneficial for visualization of directed relations as containment in treemaps suggests directed relations.

The research prototype *Code Bubbles* [25] visualizes fragments of interrelated Java code and documentation in bubbles. Code bubbles are small windows displaying text. Interrelated source code fragments are highlighted and relations are illustrated as navigable arrows pointing to other code bubbles. The authors conduct a user experiment [26] and provide evidence, that program comprehension is facilitated by visualizing source code in navigable code bubbles compared to the file-based editors of contemporary IDEs.

Without being explicit about CLS mechanisms, Freude et al. [48] formulate visualization, navigation, and static checking of consistency of relations as

10. http://www.jetbrains.com/editors/html_xhtml_css_editor.jsp?ide=idea

11. http://www.jetbrains.com/idea/features/spring_framework.html, http://www.jetbrains.com/idea/features/groovy_grails.html

requirements for their *ToolNet* prototype. *ToolNet* provides an implementation of these three CLS mechanisms across all integrated tools.

Recent research on the use of automatic refactorings in single language settings [91, 135] hints on what kind of refactorings are most useful in development of multi-language software systems. Both works evaluate large data sets of how developers use refactorings. Murphy-Hill et al. [91] evaluate more than 240,000 refactorings supported by Eclipse applied by more than 13,000 developers. They find, that rename refactorings are the most widely deployed automatic refactorings. Languages engineered with the *Xtext* language workbench support automatic rename refactoring across language boundaries by default [128].

More complex refactorings and in particular cross-language refactorings are studied in [30, 110]. Schink et al. [110] apply so called multi-language refactorings to multi-language software systems. The authors argue that not all such refactorings can be automatized. In particular, if cross-language refactorings should preserve semantics of multi-language software systems. The reason is, that not only the combination of languages is crucial but also application frameworks which make use of certain language combinations. Frameworks have idiosyncratic semantics that is hard to capture in generic tools. The authors give examples of cross-language refactorings, which are correct with respect to one application framework but incorrect with respect to another.

Contribution In Papers **D** and **F**, I identify the four CLS mechanisms visualization, navigation, static checking, and refactoring as elementary support mechanisms for multi-language development environments. I implement them in TexMo (Paper **D**) and in Coral (Paper **F**). I evaluate their impact on multi-language software system developers with TexMo (Paper **E**). Both prototypes implement the CLS mechanisms in a similar manner as they are implemented today in IDEs for single languages. The reason is, that the multi-language development environment in the experiment should be familiar to developers. It should not disturb developers with completely unfamiliar mechanisms. However, the presented related work in this section provides alternative solutions which should be considered for integration in next generation multi-language development environments. Many results provide evidence of the usefulness of the corresponding mechanism compared to more traditional support.

I am not aware of any IDE, which implements the elementary CLS mechanisms generically comparable to TexMo and Coral. The CLS mechanisms of both prototypes are applicable for development of multi-language software system independently of the used languages or the used development frameworks.

4.6 Overview and Comparison of Related Work

The following tables (Table 4.1 and Table 4.2) provide an overview of the related work presented in the previous sections with respect to the taxonomy of design decisions for multi-language development environments. This overview is compiled out of my notes of the related work. It may serve as a starting point for future PhD students working in the field of tool support for development of multi-language software systems.

Resource	Lang. Rep.		Rel. Model	Rel. Types		Inference	CLS Mech.											
	Syntactical																	
	Lexical	Per Language	Per Lang. Group	Universal	Explicit	Interfaces	Tags	Search-based	Free	Fixed	String-trans.	Domain-specific	Prog. Instrum.	Artifact Interpret.	Visualization	Navigation	Static Checking	Refactoring
[87]	✓																	
[116]		✓				✓										✓		
[122]	✓																	
[59]		✓								✓		✓				✓		
[13]		✓	✓		✓		✓			✓	✓	✓				✓	✓	
[35]		✓								✓	✓	✓				✓	✓	
[39]		✓								✓	✓	✓				✓	✓	✓
[74]		✓								✓	✓	✓					✓	✓
[33]		✓	✓		✓											✓	✓	
[129]		✓	✓		✓											✓	✓	
[89]		✓																
[28]		✓											✓					
[17]		✓																
[60]		✓																
[22]		✓																
[16]		✓																
[64]		✓	✓					✓								✓	✓	
[90]		✓	✓	✓														
[7]		✓				✓										✓	✓	✓
[48]		✓				✓										✓	✓	✓
[53]			✓													✓	✓	
[68]			✓													✓	✓	
[121]		✓	✓									✓				✓	✓	✓
[120]		✓	✓									✓				✓	✓	✓
[55]			✓														✓	
[81]		✓	✓													✓	✓	
[70]		✓	✓						✓			✓			✓		✓	
[111]		✓	✓						✓			✓			✓		✓	
[44]		✓	✓															
[43]		✓	✓															
[132]		✓	✓															
[38]		✓	✓															
[106]		✓	✓															
[105]		✓	✓															
[123]		✓	✓															
[88]						✓								✓				
[134]		✓				✓						✓						
[29]		✓				✓						✓					✓	
[73]		✓				✓						✓					✓	
[99]		✓				✓						✓						
[109]		✓				✓						✓					✓	
[54]		✓				✓		✓									✓	
[72]		✓				✓							✓	✓				
[97]		✓				✓						✓	✓					
[104]		✓				✓											✓	
[92]		✓				✓		✓										
[62]		✓				✓										✓	✓	
[58]		✓				✓										✓	✓	
[56]								✓										
[98]		✓						✓								✓	✓	
[95]		✓						✓				✓				✓		
[8]		✓							✓							✓	✓	
[133]		✓						✓								✓		
[63]		✓						✓									✓	
[61]		✓						✓										
[79]		✓				✓		✓										
[32]		✓				✓		✓										
[107]		✓						✓										
[6]						✓								✓			✓	
[67]		✓				✓		✓						✓			✓	✓
[86]						✓											✓	✓
[36]		✓															✓	✓
[66]		✓													✓	✓	✓	

Table 4.1: Comparison of related work with respect to the taxonomy of Figure 6.5

Resource	Lang. Rep.			Rel. Model	Rel. Types		Inference		CLS Mech.									
	Syntactical																	
	Lexical	Per Language	Per Lang. Group	Universal	Explicit	Interfaces	Tags	Search-based	Free	Fixed	String-trans.	Domain-specific	Prog. Instrum.	Artifact Interpret.	Visualization	Navigation	Static Checking	Refactoring
[119]	✓		✓					✓										
[10]	✓							✓									✓	
[9]	✓							✓									✓	
[4]					✓		✓			✓		✓						
[114]	✓				✓							✓						
[93]	✓				✓							✓						
[118]	✓				✓				✓						✓	✓		
[131]	✓				✓				✓						✓	✓		
[52]	✓				✓									✓				
[80]	✓				✓							✓	✓					
[136]	✓				✓							✓	✓					
[12]	✓				✓							✓	✓					
[72]	✓				✓							✓	✓					
[51]	✓				✓							✓	✓					
[27]	✓				✓							✓		✓				
[126]	✓				✓							✓		✓				
[127]	✓				✓							✓		✓				
[103]	✓				✓							✓		✓				
[112]	✓				✓							✓		✓				
[40]	✓				✓							✓		✓				
[85]	✓				✓							✓		✓				
[11]	✓				✓		✓					✓		✓				
[71]	✓				✓							✓				✓	✓	✓
[42]	✓				✓							✓			✓			
[102]	✓	✓			✓							✓			✓			
[25]	✓				✓							✓			✓			
[91]																		✓
[135]																		✓
[128]	✓	✓	✓													✓	✓	✓
[30]	✓															✓	✓	✓
[110]	✓																	✓

Table 4.2: Table 4.1 continued.
Comparison of related work with
respect to the taxonomy of
Figure 6.5

“Old’s cool me tell ya how its cool:
I like the way the girls are dancing to this beat
without the right moves and feet.
I love the way them just a twee, sweet, and neat.
Old’s cool somethings come back fi real!”

Dr. Ring Ding, Old’s Cool

Software development aims at constructing software systems, which fit customer's requirements. The systems should be evolvable, customizable, and adaptable to changing environments and changing requirements. Perhaps most importantly, software systems should be free of errors. Various factors hinder development of software systems in particular when they are multi-language software systems. Development, customization, and evolution of large multi-language software systems is challenging. Such systems are constructed out of a multitude of interrelated development artifacts in a broad variety of languages. The artifacts range from high-level artifacts such as specification and requirements documents over modeling artifacts to low-level programming artifacts. Depending on the phase in software development and on the abstraction level of a development artifact different languages are used. Used languages range from natural languages to visual and textual languages.

Many of the challenges are caused or exacerbated by the fact that modern software systems are multi-language software systems, and in particular by the lack of good and appropriate development tools (multi-language development environments). I detail these problems in the following:

Problem i) Missing Cross-language Support Mechanisms Usually, contemporary IDEs do not provide support mechanisms across development artifacts of different types. For example, navigation across development artifacts in different languages is usually not supported, static checking exists usually separately for each programming language but not across different programming languages, etc.

Problem ii) The Architecture of Contemporary IDEs Targets Development with Single Languages Usually, IDEs integrate development tasks on single programming languages. For example, IDEs provide integrated code editors, debugging, and refactoring tools separately for each language used. The various editors and tools for different languages are typically not integrated with each other. The underlying reason is, that the architecture of contemporary IDEs allows for instantiation of integrated tools targeting single languages.

Problem iii) Lack of Generic Tools for Development of Multi-language Software Systems For some IDEs exist extensions supporting development with interrelated development artifacts in few different languages. Such tools usually support few specific tasks across languages. The problem is, that these tools are usually not generic. That is, they can only be used in particular domains and in particular setups for development of multi-language software systems. For example, in development with few fixed frameworks or particular language combinations.

Development, customization, and evolution of multi-language software systems is tedious, cumbersome, and error-prone due to the shortcomings in contemporary IDEs and software development processes given above.

5.1 Research Questions

Given the context of my research, the overall research question addressed in this thesis is:

How to support development of multi-language software systems?

In order to tackle the broad research question, I refine it into eight detailed questions and organize them under three high-level goals. The structure is given by three major perspectives on development of multi-language software systems. The first perspective considers characteristics of multi-language software systems (**Goal A**), the second perspective is concerned with the needs of developers with respect to features in tools supporting development of multi-language software systems (**Goal B**), and the third perspective addresses conceptual and technical foundations for tool builders constructing multi-language development environments (**Goal C**).

Goal A Investigate characteristics of multi-language software systems. (**System Perspective**)

RQ 1 What are the characteristics of language usage in contemporary multi-language software systems?

RQ 2 What are the characteristics of relations between development artifacts within and across language boundaries?

RQ 3 What type of relations between development artifacts exist in multi-language software systems?

Goal B Identify and evaluate features needed in a tool to support or enhance development of multi-language software systems. (**Developer Perspective**)

RQ 4 What are the elementary features in multi-language development environments that support developers of multi-language software systems?

RQ 5 Do developers of multi-language software systems require certain features or properties?

Goal C Explore the domain of multi-language development environments and identify major design decisions and building blocks for creation of multi-language development environments. (**Tool Builder Perspective**)

- RQ 6** How can software development artifacts be represented? How to characterize and reference the information in software development artifacts that contribute to relations?
- RQ 7** How can relations between development artifacts, within and across language boundaries, be represented? How to characterize and formalize relations between development artifacts?
- RQ 8** How to automatically reveal or infer relations between different development artifacts? Is this feasible at all?

5.2 Theses

Corresponding to the three major perspectives structuring the research questions, I formulate three precise theses:

- T1** Multi-language software systems exist. Contemporary software systems mostly consist of multiple heterogeneous development artifacts, which are interrelated.
- T2** Multi-language software systems can be developed, evolved, customized, and adapted more effectively, when appropriate multi-language development environments are provided. Developers make less errors, are more efficient, and work faster, when supported with a set of cross-language support mechanisms.
- T3** Provision of a theoretical framework enables tool builders to create multi-language development environments with effective CLS mechanisms tailored to the domain and use cases of the multi-language development environment.

*“Problems, everyday is problems,
problems, problems, and problems...”*

Desmond Dekker, Problems

In this chapter I introduce the different methods (Section 6.1), which I applied in the various research papers to tackle the research questions (Section 5.1). I summarize and discuss each research paper. For each it, I discuss the addressed goals and contributions to answering the research questions. The section concludes by giving an overview of all publications, their contribution to the goals, the addressed research questions, and the applied methodology (Section 6.3).

6.1 Methodology

The research papers forming this thesis rely on the following methods.

Tool prototyping This thesis is conducted in the field of software engineering. Software engineering in a scientific setting relies on construction of software prototypes, which are evaluated in experiments with respect to research questions. All research papers in this thesis present and evaluate tool prototypes, see Section 1.1.2 for a summary of the prototypes.

Literature surveys are usually performed to define the scope of a research contribution with respect to similar solutions and related work or they discover the state of the art in a certain research field. Each paper surveys literature to discuss related work and to compare the presented tool prototypes with it. Papers **D** and **F** present a more elaborate survey of literature and tools to discover the state of the art of tools supporting development of multi-language software systems.

Technical experiments evaluate entire prototypes or certain of their properties either by observing the application of the prototypes to artifacts with interesting or representative properties in a laboratory environment, or, they let other computer programs instrument the prototypes. The majority of tool prototypes included in this thesis are evaluated by technical experiments, see Paper **A**, Paper **B**, Paper **F**, and Paper **G**.

User experiments evaluate tool prototypes or certain of their properties by observing users performing tasks in a controlled environment. That allows to measure the effect of the prototype to the users with respect to

certain criteria. Paper **G** reports on a controlled experiment in which the effectiveness of CLS mechanisms in development of a multi-language software system is evaluated.

Case study demonstrate feasibility and applicability of tool prototypes by applying them to a representative case. The case can be either a real-world case or a simplified controlled case in a laboratory setting. Paper **C** presents a case study on a small exemplary multi-language software system.

Surveys discover user's opinions or practices regarding certain research questions. Usually, surveys are conducted by issuing questionnaires to a group of users or surveys are conducted by interviewing groups of users. Especially Paper **F** presents the results of an online survey, which aims to discover current practices of language developers and tool builders with respect to support development with interrelated languages.

6.2 Summary & Contributions per Paper

6.2.1 An Aspect-based Traceability Mechanism for Domain Specific Languages – ECMFA-TW'10 (Paper A)

Summary

This paper investigates how to automatically infer relations between different concrete syntaxes of visual modeling languages. Usually, model elements of the same visual language have different representations. In abstract syntax they are objects in memory, which are rendered in different visual concrete syntaxes and, which are stored in serialization syntaxes (often XML-based). Figure 6.1 illustrates a model element in visual concrete syntax (top left), abstract syntax (top right), textual serialization syntax (bottom), and the relations across the different syntaxes (highlighted in red).

Especially in this paper, I take the view that visual languages on computers are always rendered representations of textual languages. The rendered visual representations are usually displayed to and edited by developers. For any development artifact in visual languages, many interrelated development artifacts in textual languages physically exist in a software system. Most modeling tools only keep trace links for in memory visual languages but they do not keep trace links between the visual representations and the serialization syntaxes. Instead, the trace links are implicitly specified in the implementations of persistence mechanisms.

The paper presents Tengja, a tool prototype, enabling element to element traceability from visual and abstract syntax to serialization syntax. The prototype automatically infers trace links between the various syntaxes by instrumenting the serialization process of visual models. Tengja is implemented as an aspect in AspectJ. Thereby, it is non-invasive and highly reusable. Neither models, languages, nor editors need to be modified. Tengja can be applied in combination with all Eclipse model editors relying on EMF and GMF. The only requirement is that these editors rely on the standard persistence mechanism. That is, Tengja is generic with respect to the particular visual language.

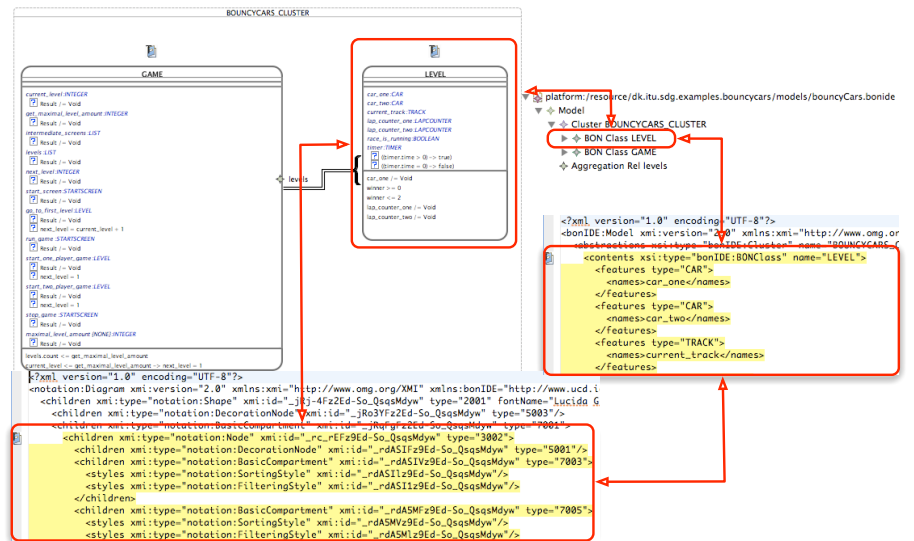


Figure 6.1: A model element in visual concrete syntax (top left), abstract syntax (top right), their textual serialization (bottom), and their relations (highlighted)

As a proof of concept demonstrating Tengja's effectiveness, I develop simple CLS mechanisms extending current Eclipse editors, with the ability to highlight relation ends and to request for visual language elements the corresponding textual representations. Tengja can serve as a foundation for further IDE extensions.

To evaluate Tengja's robustness, the paper presents a technical experiment. The results indicate that Tengja is indeed applicable to multiple visual languages.

Methods. This paper applies tool prototyping and technical experimentation as methods. The tool prototype Tengja is constructed to demonstrate the general feasibility of automatic inference of trace links between different concrete syntaxes. This is evaluated in a technical experiment.

Quality of the Solution

The robustness of the prototype is evaluated in a technical experiment. A test program automatically traverses all elements in various visual languages. It requests trace links from visual language elements to the corresponding counterparts in serialization syntax. The trace links are logged into files, which are manually checked for correctness. The check reveals that all important language elements, those that are displayed to developers in editors, are correctly traced to the corresponding fragments in serialization syntax. The languages used in this experiment are of different size and they are all applied in real products, mostly in the EMF. The different languages highlight that Tengja is generic, supporting more than only one language. Currently, Tengja is tied to the EMF framework which is used for language implementation. But, since it is implemented using an aspect, the amount of work when adapting it to a new language development framework is relatively small.

Contributions

This paper primarily contributes to **Goal C** (Tool Builder Perspective) as Tengja is a potential building block for a multi-language development environment. Research question **RQ 8** asks for feasibility of automatic inference of relations between artifacts and it asks for possible solutions. Based on an aspect, Tengja

automatically establishes trace links between visual, abstract, and serialization syntax. That is, for this particular use case, the paper proves the feasibility of automatic inference of trace links between different interrelated concrete syntaxes. Furthermore, Tengja relies on lexical language representation of any language to enable highlighting of relation ends of trace links. All the visual languages have a syntactic per language representation. This suggests two possible language representations (**RQ 6**).

The paper also contributes to **Goal A** (System Perspective). The experiment reveals that the trace links between the various concrete syntaxes, i.e., one particular type of relations (**RQ 3**) are quite frequent for development artifacts in visual languages (**RQ 2**). Additionally, the paper provides insight into research question **RQ 1** asking for characteristics of language usage in multi-language software systems. It is obvious that there exist different representations already for the concrete syntaxes of single languages.

Although the paper does not directly address the research question **RQ 4**, which asks for elementary CLS mechanisms, it implements the CLS mechanisms navigation and visualization. As both mechanisms are basic features of multi-language development environments, this paper contributes partly to **Goal B** (Developer Perspective).

6.2.2 Taming the Confusion of Languages – ECMFA’11 (Paper B)

Summary

This paper researches OFBiz, an industrial-strength open-source ERP system, with respect to relations between development artifacts crossing OFBiz’ component boundaries. Usually, such relations are implicit, i.e., they are not explicitly declared in a relation model at development time but first established runtime. Relations across languages and across components cause a number of problems for software developers. First, implicit cross-language relations require substantial domain knowledge to correctly perform simple system evolution steps. Second, implicit relations may cross component boundaries, which couples components tightly together. Third, errors caused by broken relations are most often only exposed at runtime, which requires thorough testing of the modified code to detect any errors. That is, implicit cross-development artifact and cross-component relations hinder development, customization, and evolution of component-oriented multi-language software systems, as no relation representation mechanism of exchangeable components capture hidden relations.

The paper presents the prototype GenDeMoG, a generic tool for specification of cross-component relation patterns for arbitrary multi-language software systems. As such relation patterns are constraints between structures in development artifact described by queries, GenDeMoG essentially is a tool for specification of search-based relation models. GenDeMoG’s patterns declare relations explicitly, enabling automatic inference of an explicit model containing previously implicit cross-component relations. The relation patterns are declared on languages used in the respective multi-language software systems. Both, GPLs like Java, and DSLs are supported by GenDeMoG. GenDeMoG is a generic tool. It is neither tied to certain languages nor to certain applications. Furthermore, it is non-invasive, i.e., it does not require related artifacts to be modified in any way.

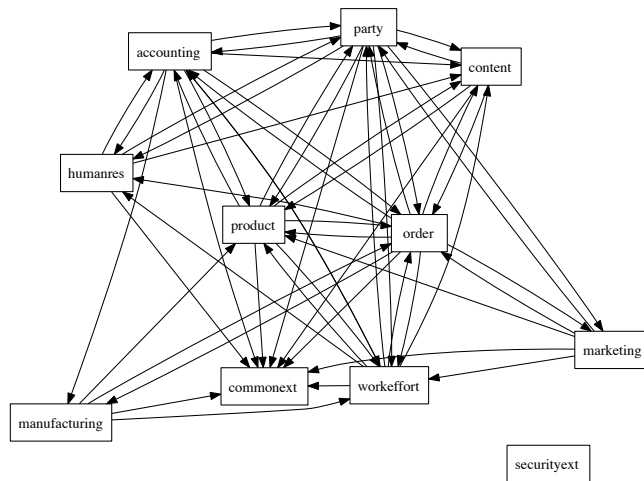


Figure 6.2: OFBiz components and automatically inferred cross-component relations in an aggregated view

Methods. The paper applies tool prototyping and a technical experiment as methods. The tool prototype GenDeMoG is constructed, to enable the analysis of a contemporary multi-language software system with respect to relations between development artifacts crossing component boundaries. Additionally, it demonstrates the general feasibility of automatic inference of such relations out of given development artifacts. The technical experiment is the application of GenDeMoG to OFBiz.

Quality of the Solution

To demonstrate GenDeMoG’s applicability, it is used to analyze an OFBiz application (Section 2.2.2) for the presence of implicit cross-component relations. For this experiment, I identified 22 examples of cross-component relation patterns involving seven languages. GenDeMoG automatically revealed 1,737 cross-component relations of the kind specified by these patterns. The result of this analysis is a cross-component relation graph. Figure 6.2 shows an aggregated view of all inferred relations between OFBiz’ components. The paper discusses the established relations to get a deeper insight in the characteristics of cross-component relations in a contemporary multi-language software system. The results confirm that relations between heterogeneous development artifacts do exist, their number is quite large, and even worse, they cross component boundaries. This means that components of development artifacts are coupled tightly and circularly, see Figure 6.2. With this results, the experiment confirms the informal assessment that modification and evolution of OFBiz’ core application components is difficult. The experiment indicates, that development of such large systems needs first, better IDE support and second, that OFBiz’ component mechanism inappropriately tracks cross-component relations between development artifacts.

Contributions

This paper contributes to **Goal A** (System Perspective) and **Goal C** (Tool Builder Perspective). All three research questions contributing to **Goal A** are addressed. The paper provides statistics for characteristics concerning language usage in OFBiz (**RQ 1**). The number of languages used in OFBiz is quite high. More than 30 languages are used. The majority of languages are XML-based

DSLs but multiple GPLs are used as well. However, all languages in OFBiz are textual languages. Furthermore, the paper describes characteristics of relations between development artifacts with a focus on cross-component relations (**RQ 2**). It is demonstrated that such cross-component relations are frequent. Additionally, the kinds of relations and their distribution within OFBiz is discussed. The paper focuses on fixed and string-transformation relations (recall Section 3.3). But since the relation patterns are specified in a Turing-complete language with full access to OFBiz' code base, any arbitrary type of relation can be implemented as a relation pattern (**RQ 3**).

GenDeMoG represents development artifacts using syntactic per language representations. That is, each development artifact is treated as a model adhering to a certain language representation (**RQ 6**). Relations between development artifacts are expressed in two ways. First, as relation patterns forming a search-based relation model. Second, after evaluation of the relation patterns an explicit relation model accumulates all relation instances. Search-based and explicit relation models are two ways to represent relations between development artifacts (**RQ 7**). GenDeMoG and its application to OFBiz demonstrates that cross-artifact relations can be effectively inferred out of development artifacts in combination with relation patterns encoding framework knowledge (**RQ 8**).

6.2.3 Tengi Interfaces for Tracing between Heterogeneous Components – GTTSE'11 (Paper C)

Summary

This paper presents Tengi, a toolkit for defining, reusing, and relating interfaces. Tengi interfaces can be defined for any development artifact or for components of heterogeneous artifacts, no matter of the used language. The interfaces can be created for artifacts in languages ranging from high-level specification languages (natural language) and visual languages to low-level implementation languages. Tengi extends numerous Eclipse editors with the ability to define *ports* on development artifacts. Ports are fragments of artifacts forming relation ends. Figure 6.3 shows an excerpt of an analysis document in the textual informal BON language (on top) with a defined port (highlighted in yellow). The bottom of Figure 6.3 shows the corresponding Tengi interface. Via ports, Tengi interfaces enable explicit description of relations between heterogeneous development artifacts. Ports in interfaces describe what an artifact requires from or what it provides to its environment.

In addition to the DSL for interface specification, the paper describes a set of operators on top of the interfaces to automatically check for compatibility and refinement, or to perform composition.

Methods. This paper applies tool prototyping and a small case study as research methods. Unlike in the other papers, the tool prototype Tengi is not evaluated by a technical or a user experiment. Instead, Tengi is applied to a small use case demonstrating its feasibility and its application to development of multi-language software systems.

Quality of the Solution

The case study applies Tengi to a small-sized multi-language software system consisting of development artifacts in multiple visual and textual languages.

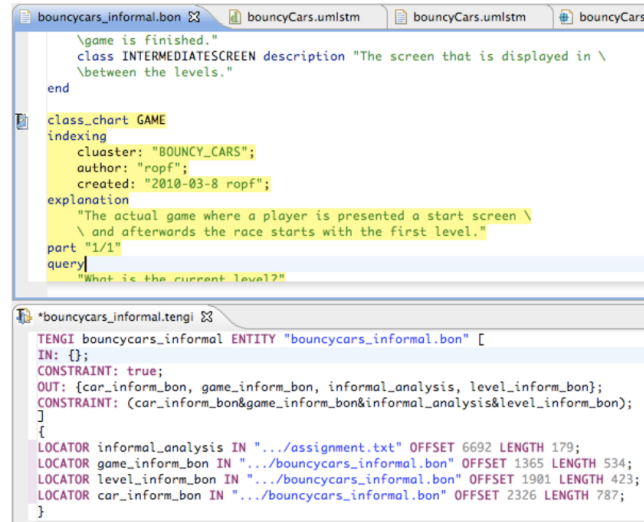


Figure 6.3: Excerpt of an analysis document in informal BON (bouncycars_informal.bon) with a marked port on top and below the corresponding Tengi interface (bouncycars_informal.tengi)

From the point of view of Tengi interfaces, all development artifacts are abstracted to their physical structure, i.e., they rely on a lexical language representation. Thereby, the interfaces are applicable to development artifacts in any language used in development of multi-language software systems. Tengi is a generic tool, as it is independent of the concrete languages used in a multi-language software system. Instead, it is only tied to the Eclipse IDE, as its editors rely on the EMF and it reuses Tengja from Paper A.

Contributions

This paper mainly contributes to **Goal C** (Tool Builder Perspective). Tengi demonstrates that heterogeneous development artifacts can be uniformly represented by a lexical language representation (**RQ 6**). That is, visual languages, natural languages, and other languages can be effectively represented by relying on the physical representation of development artifact contents in serialization syntax. Ports of interfaces can be specified on such a lexical language representation by describing ranges in a character stream. Therefore, the relations expressed by Tengi interfaces are free relations, i.e., the contents of relation ends are not constrained, they just need to be present (**RQ 7**). However, the interfaces explicitly describe previously implicit relations. Integrity of relations between heterogeneous development artifacts can be checked by application of interface operations.

6.2.4 TexMo: A Multi-Language Development Environment – ECMFA'12 - (Paper D)

Summary

The paper focuses on development environments for development of multi-language software systems. It elaborates on the fact that existing IDEs do not directly support development of such systems. IDEs usually do not visualize cross-language relations, they do not provide navigation along cross-language relations, they do not statically check cross-language relations for consistency, nor do they provide refactorings across development artifacts in heterogeneous languages. These mechanisms are not provided by contemporary IDEs, even though relations across language boundaries are frequent and fragile, i.e., easily broken at development time without specific support for developers. The

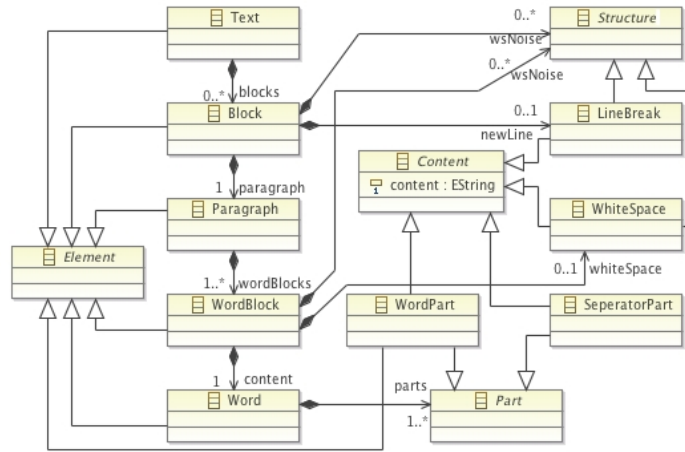


Figure 6.4: TexMo, a model for universal language representation

paper sets out to improve development of multi-language software system by enhancing IDEs into multi-language development environments. This paper is the first of my publications to use the terms multi-language software systems and multi-language development environment.

The paper addresses multi-language development environments in two ways. First, from a general perspective a taxonomy of design choices for multi-language development environments is introduced. Second, a particular multi-language development environment, an instance of these design choices is presented. The taxonomy and the CLS mechanisms are identified by a survey of literature and tools.

The taxonomy makes major necessary design choices for multi-language development environments explicit. For example, these are language representation, representation of relations between languages, and different types of relations. The paper distinguishes between *lexical language representation*, the physical representation of a development artifacts contents as a sequence of characters, and *syntactic language representation*, representations using trees and graphs as underlying data structures to describe abstract concepts and their relations, see Section 3.1. *Relation models* in the taxonomy are the four relation models discussed in Section 3.3.1, i.e., explicit relation models, interfaces, tag-based relation models, and search-based relation models. The purpose of this taxonomy is twofold. First, it serves as requirements list for implementing multi-language development environments, and second it allows for classification of such. The taxonomy is the result of a survey of related literature and tools. Furthermore, this paper identifies and defines the four mechanisms, visualization, navigation, static checking and refactoring of cross-language relations as elementary cross-language support (CLS) mechanisms.

In addition to the taxonomy, the paper presents TexMo, a prototype of a multi-language development environment providing the four elementary CLS mechanisms across all textual GPLs and DSLs uniformly. The CLS mechanisms are implemented by leveraging an explicit relation model keeping track of relations between heterogeneous development artifacts. To be applicable to any textual language, TexMo represents development artifacts via a universal language representation. In TexMo, the universal language representation is a text model representing contents of development artifact structurally as words, whitespaces, blocks of words, etc, as illustrated in Figure 6.4.

Methods. The presented research applies literature survey, tool prototyping, a small case study, and a user test as research methods. The literature survey is conducted to identify the state of the art in the field of multi-language development environments and to establish a set of elementary CLS mechanisms. The tool prototype TexMo is constructed to demonstrate the general feasibility of a multi-language development environment implementing the four CLS mechanisms, visualization, navigation, static checking, and *refactoring* on top of a universal language representation in combination with an explicit relation model. In preparation of the user tests it is evaluated that it is indeed possible to represent all of JTrac’s development artifacts as instances of the universal language representation model. For the user test an example of an explicit relation model is created.

Quality of the Solution

The paper positions TexMo in the taxonomy of design choices of multi-language development environments. Furthermore, TexMo is evaluated by applying it to development of the web-application, JTrac Section 2.2.1. The universal language representation for textual languages is evaluated by demonstrating that all of JTrac’s 291 development artifacts are represented by an instance of the universal language representation model, which allows for interrelation of development artifact concepts by the explicit relation model. The example explicit relation model, created for the user test, interrelates nine development artifacts with in total 87 cross-language relations. By running user tests and interviews, the paper provides preliminary evidence of TexMo’s feasibility. A complete controlled experiment using TexMo in development of JTrac is reported in the following paper in Section 6.2.5 (Paper E).

Contributions

All three goals **Goal A** (System Perspective), **Goal B** (Developer Perspective), and **Goal C** (Tool Builder Perspective) are addressed in this paper. JTrac as example of a multi-language software system, illustrates the existence of various textual GPLs and DSLs in such systems. The languages range from GPLs like Java over mark-up languages like HTML and XML to DSLs such as properties files (**RQ 1**). The paper describes an example explicit relation model containing fixed relations, i.e., relations between identical fragments of development artifacts (**RQ 3**). The relation model is manually created, however, it demonstrates that heterogeneous development artifacts, at least in parts of JTrac, are quite interrelated (**RQ 2**).

This paper is the first one contributing to **Goal B** (Developer Perspective), i.e., elaborating on required multi-language development environment features. The paper identifies the four CLS mechanisms visualization of, static checking of, navigation along, and refactoring of cross-language relations as essential mechanisms for a Multi-language Development Environment (MLDE) (**RQ 4**). Even though, the paper does not provide strong evidence for which features multi-language software system developers require from a multi-language development environment, the collected user feedback demonstrates that already the four CLS mechanisms are highly appreciated by MLDE users, which partially answers **RQ 5**.

As the paper addresses multi-language development environments in two ways, it contributes to **Goal C** (Tool Builder Perspective) in two ways. The taxonomy provides general answers to **RQ 6** and **RQ 7**. That is, it explicitly describes de-

sign decisions which existing tools and the studied literature make for language representation, relation models, and relation types, etc. Making the design choices for multi-language development environments explicit, the taxonomy lists all possible solutions for implementing a multi-language development environment. Since TexMo implements a multi-language development environment instance, it provides specific answers to the research questions **RQ 6** and **RQ 7**. It illustrates that heterogeneous development artifacts can be effectively represented using a universal language representation model abstracting over structures of their contents (**RQ 6**). Thus, an explicit relation model can reference development artifact contents, for example, by provision of expressions navigating along paths in the universal language representation (**RQ 7**). The relations in TexMo's explicit relation model can be fixed and free relations (**RQ 7**).

6.2.5 Cross-Language Support Mechanisms Significantly Aid Software Development – MODELS'12 (Paper E)

Summary

Multi-language software systems are a matter of fact, see Section 2.1. Developers constructing such systems constantly deal with heterogeneous development artifacts. Therefore, development tools should support relations, in particular cross-language relations, between heterogeneous development artifacts at development time. I believe that development of multi-language software systems could be significantly improved if IDEs included multi-language development support, as known from single languages. For example, if IDEs provided visualization, static checking for consistency, navigation, and refactoring of cross-language relations, developer's understanding of the system would improve and the number of errors made at development time would be reduced. This paper investigates whether the four elementary CLS mechanisms indeed improve efficiency and quality of development of multi-language software systems. To test this, I conduct a controlled experiment in which 22 participants perform typical software evolution tasks on the JTrac web-application using TexMo (see Paper **D**), a MLDE which implements the four elementary CLS mechanisms. The results speak clearly for integration of cross-language support mechanisms into software development tools and they justify research on automatic inference, manipulation and handling of cross-language relations.

Methods. This paper applies tool prototyping and a user experiment as research methods. The tool prototype TexMo, implements the four CLS mechanisms to allow for measuring their impact on developers performing certain development tasks on JTrac.

Quality of the Solution

To evaluate the impact of the four CLS mechanisms on development of multi-language software systems, I conduct a controlled experiment with 22 participants, the experimental subjects. The subjects are software professionals, PhD, MSc, and undergraduate students, who are between 18 and 48 years old. The subjects are divided into two groups. A treatment group uses TexMo with all four CLS mechanisms enabled and a control group performs the experiment tasks using TexMo with disabled CLS mechanisms. The JTrac system is the experimental unit and TexMo with enabled and disabled cross-language support

is the experimental variable in two alternatives. All subjects perform the same three tasks within half an hour, i.e., ten minutes per task. The tasks include (i) location and fix of a broken cross-language relation, (ii) renaming of a source code element what breaks a cross-language relation which should be fixed again, and (iii) replacement of a code block what breaks multiple cross-language relations. All tasks include reasoning about the effects and potential problems of the performed tasks.

The results of the experiment are that visualization, static checking, navigation, and refactoring across language boundaries are highly beneficial. CLS mechanisms perceptibly improve effectiveness of developers working on JTrac, a representative multi-language software system. In the experiment scenario, users of CLS are more effective than the control group with respect to both error rate and productivity (working speed). Furthermore, it is shown that, within the experiment, CLS mechanisms are actually used by developers and that they improve understanding of complex unknown multi-language source code.

Contributions

This paper is my main contribution to **Goal B** (Developer Perspective). Since the MLDE prototype TexMo is used as experimental variable, the paper also contributes to goals **Goal A** (System Perspective) and **Goal C** (Tool Builder Perspective). The contribution is analogous to the contribution described in Section 6.2.4 for Paper **D**. Here, TexMo serves the purpose to enable the user experiment.

The result of the experiment is that the CLS mechanisms are actually used by multi-language software system developers and that they are beneficial with respect to speed of work, error rate, and understanding of developed multi-language software system source code. The quantitative results of the experiment suggest the four CLS mechanisms visualization, static checking, navigation, and refactoring of cross-language relations as essential features of an MLDE (**RQ 4**). The qualitative results suggest that developers of multi-language software systems actually require such mechanisms (**RQ 5**).

6.2.6 The Design Space of Multi-language Development Environments – SoSyM’13 (Paper F)

Summary

This paper is an extended version of Paper **D**. It extends and revises its literature survey and adapts and reuses elements from Paper **E**. The implementation of the Coral MLDE, the comparison of Coral with TexMo, the technical experiment, and the survey in the language developer community are entirely new.

Similar to Paper **D**, this paper focuses on multi-language development environments. It addresses them in two ways. First, from a general perspective, where a taxonomy of design choices for multi-language development environments is introduced and second, two particular multi-language development environments, which are instances of these design choices are presented, compared, evaluated, and discussed.

The literature survey documents the main design choices for MLDEs which are summarized in a taxonomy. The taxonomy contains both the defining requirements for multi-language development environments and the variability in their implementation. The literature survey is extended even further in this

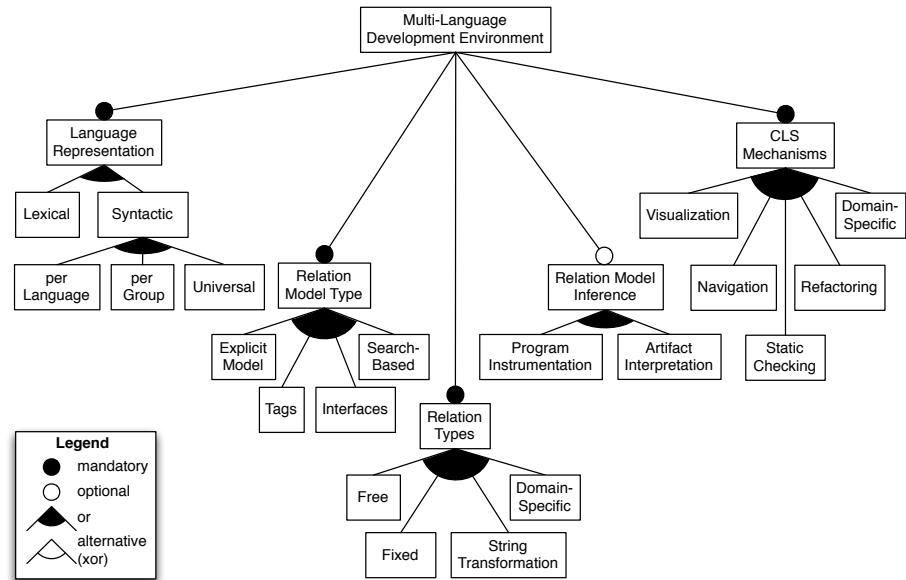


Figure 6.5: The taxonomy for multi-language development environments, see Paper F

thesis in chapter Chapter 4. Note, that the taxonomy also incorporates inference of relation models, what is prominently described in Paper F.

In addition to the research of design space for MLDEs, this paper presents Coral and TexMo, two prototypes of multi-language development environments. Both tools implement the four elementary CLS mechanisms of cross-language relations similarly. With respect to the design choices in the taxonomy, the prototypes are radically different. As described in Paper D, TexMo represents development artifacts via a universal language representation and interrelates them by encoding cross-language relations in an explicit relation model. On the other hand, Coral represents development artifacts using per language representations and specifies cross-language relations in a search-based relation model. The search-based relation model specifies constraints interrelating development artifacts on language level, see Figure 2.13. Coral allows to distribute the relation specifications into libraries. Consequently, Coral can be customized with various libraries for different development scenarios. Additionally, Coral provides an inference tool enabling semi-automatic inference of search-based relation models. Both, TexMo and Coral are generic. They do not depend on any particular languages interrelated in multi-language software systems. Thus, they can be applied to development of arbitrary multi-language software systems.

The two MLDE implementations illustrate the challenges, which tool builders face when constructing different kinds of MLDEs. They also materialize two, design points in the taxonomy. The experiences with both tools applied to development of the multi-language software system JTrac are discussed and the differences between the two tools are analyzed qualitatively.

Finally, to evaluate the need for and the usefulness of multi-language development environments two experiments approach the communities of multi-language software system developers and language and tool builders. Concerning TexMo, the paper presents a condensed discussion of the evaluation of the CLS mechanisms (Paper E). Coral is evaluated in a technical experiment and indirectly in a survey in the community of language developers. The survey investigates the current practices in language development and the kind of tool support provided by language developers.

Methods. This paper relies on five different research methods. These are: literature survey, tool prototyping, a controlled experiment, a technical experiment, and a survey in the community of language developers.

The literature survey is conducted to identify the state of the art in the field of multi-language development environments and to establish a set of common CLS mechanisms. The two multi-language development environment prototypes TexMo and Coral are used to discuss experiences and the multi-language development environment design decisions. In a technical experiment Coral's inference tool is applied to JTrac illustrating its multi-language characteristics. This paper argues for that multi-language development environments should be generic to support development of diverse multi-language software systems. Accordingly, the paper reports on an online survey in the community of language developers indicating the need for generic language integration tools.

Quality of the Solution

The paper presents an investigation of the design space of multi-language development environments from the three different perspectives, the system perspective, the developer perspective, and the tool builder perspective.

First, concerning the system perspective, the paper reports on a technical experiment in which Coral's inference tool is used to semi-automatically infer cross-language relation constraints (the search-based relation model) out of JTrac's development artifacts. The result are two libraries containing ten constraints. With these two libraries Coral is applied to automatically establish cross-language relations within JTrac's artifacts. The result is, that JTrac contains at least 4,941 cross-language relations. This illustrates that heterogeneous development artifacts in JTrac are quite interrelated and that there exist clearly too many cross-language relations to be handled manually.

Concerning the developer perspective, the paper demonstrates, by recapitulation of the results of the user experiment in Papers **D** and **E**, that developers benefit from CLS mechanisms provided in a multi-language development environment when developing multi-language software systems.

Finally, concerning the tool builder perspective, the paper reflects for TexMo and Coral on experiences with constructing two new and different MLDEs prototypes, following two different design choices. TexMo is highly adaptable to development of diverse multi-language software systems due to its universal language representations. This representation however comes at a cost of limited scope of functionality. Coral is highly adaptable to various multi-language software systems due to its generic search-based language relation model. This representation however comes at a cost of many required per language representations for development artifacts. A survey in the community of language developers confirms that the constructed languages are very frequently related to other languages, that tooling to integrate the various languages is partially provided, and that the provided tools are rarely generic. Indirectly, this evaluates Coral's potential benefit in development of multi-language software systems, as it is a generic tool.

Contributions

This paper is the most elaborate paper in the collection of papers forming this thesis. It addresses all three goals with all research questions **RQ 1 – RQ 8**.

Goal A (System Perspective) is addressed by the technical experiment which

applies Coral to JTrac. First, it is demonstrated that diverse textual GPLs and DSLs exist in this contemporary multi-language software system. The various languages range from GPLs like Java over mark-up languages like HTML and XML to DSLs such as properties files (**RQ 1**). The paper reports on a technical experiment, in which Coral is used to automatically collect cross-language relations of JTrac’s heterogeneous development artifacts. The results illustrate, that heterogeneous development artifacts in JTrac are quite interrelated (**RQ 2**). Furthermore, it is shown that fixed and string-transformation relations are quite prominent relation types in heterogeneous source code, i.e., in a technical domain (**RQ 3**).

As the results from the user experiment (Papers **D** and **E**) are recapitulated and summarized, the contributions of this paper to **Goal B** (Developer Perspective) are analogous to the contributions described in Section 6.2.4 and Section 6.2.5. However, the survey in the language developer community reveals that language development community is lacking a generic parametrizable MLDE. This is not a CLS mechanism but more a property of MLDEs, but the result partially addresses **RQ 5**, asking for required features of MLDEs.

Similarly to Paper **D**, this paper contributes to **Goal C** (Tool Builder Perspective) in two ways. First, the taxonomy provides general answers to **RQ 6** and **RQ 7**. The taxonomy captures design decisions of existing tools and related literature with respect to language representation, relation models, relation types, etc. Thus, the taxonomy lists all possible solutions for implementing multi-language development environments. Both, TexMo and Coral are instances of a MLDE. Due to their opposing design decisions, they provide specific but differing answers to research questions **RQ 6** and **RQ 7**. They illustrate that, depending on the use case of an multi-language development environment, heterogeneous development artifacts can be effectively represented (**RQ 6**) by a universal language representation model, which abstracts over structures of development artifact or by per language representations, which abstract over the concepts of heterogeneous languages independently. Similarly, depending on the use case, TexMo and Coral rely on different types of relation models (**RQ 7**). TexMo leverages an explicit relation model to describe cross-language relations linking fragments of development artifacts via path expressions on instances of universal language representation model. Contrary, Coral represents cross-language relations as constraints in a search-based relation model. Both MLDE prototypes implement differently typed relations (**RQ 7**). In TexMo, relations can be of fixed or free relation type and in Coral the four relation types, fixed, string-transformation, free, and domain-specific can be specified.

The paper also demonstrates that search-based relation models can be inferred semi-automatically out of development artifacts and that, once a correct search-based relation model is in place, an explicit relation model can be correctly established (**RQ 8**).

6.2.7 Language-Independent Traceability with Lässig – Under Submission (Paper G)

Summary

Trace links between heterogeneous development artifacts allow to implement efficient tools supporting developers working on such artifacts. However, most of contemporary programming languages and model transformation languages do not provide support for traceability in the first place. Today, if traceability

support is required, either systems need to be implemented in one of the few programming language with built-in traceability support, such as ETL or ATL, or traceability directives need to be added to existing system implementations. Implementing traceability with languages with built-in traceability support is not suitable for legacy systems, as it would require reimplementation. Adding traceability to existing implementations is not suitable as business logics is polluted with the application independent concern of traceability. Furthermore, the same concern has to be implemented repeatedly in different domains. Traceability is a typical cross-cutting concern [124] with respect to a single software system. At the same time it is a multi-domain concern with respect to many software systems. Cross-cutting concerns are effectively handled with aspect-oriented programming [82]. Reoccurring similar concerns in different domains are effectively handled with model-driven software development [115].

This paper presents a generic model-driven solution to add traceability support to all programming languages and model transformation languages, which are compiled to and executed on a virtual machine. Lässig is an implementation of this solution targeting all languages compiling to the Java Virtual Machine (JVM). Lässig, is parametrized with metamodels declaring traceable types. Lässig generates aspects, which instrument the execution of methods transforming objects with directives to automatically generate trace models. The kinds of instrumented method executions are based on two heuristics. A technical experiment demonstrates that only these two heuristics allow to establish correct trace models. However, potential alternative heuristics for tracing are discussed too.

Methods. The paper applies two research methods, tool prototyping and a technical experiment. The tool prototype Lässig is applied to a set of model transformations in different programming languages.

Quality of the Solution

Lässig is evaluated in a technical experiment. First, the correctness of the automatically inferred trace models is evaluated. Second, a controlled experiment evaluates the performance overhead of model transformations with Lässig's traceability support. For the correctness test, Lässig is applied to three model to model transformations. Each of them is implemented in each of the three programming languages Xtend, Java, and Groovy. The resulting trace models are manually compared with trace models generated by comparable ETL transformations. The results demonstrate that Lässig automatically generates complete and correct trace models introducing a moderate performance overhead to the transformations.

Contributions

Similarly to the first paper in this thesis, this last paper is primarily contributing to **Goal C** (Tool Builder Perspective), since Lässig can be integrated in other modeling tools and multi-language development environments. The technical experiment provides evidence that it is feasible to automatically infer trace links between interrelated objects by instrumenting model transformations executed on the JVM with aspects, which are automatically generated out of metamodels (**RQ 8**). As Lässig automatically establishes relations between objects that are subject to a transformation, it also answers research question **RQ 3** (asking

for the different types of relations). The type of relations across development artifact in this case are trace links.

6.3 Contributions in a Nutshell

In this section, I compare all research papers forming this thesis in Table 6.1. For each paper, the table illustrates its contribution to the goals, the addressed research questions, and the applied methodology.

Tool Name	Paper A	Paper B	Paper C	Paper D	Paper E	Paper F	Paper G
	Tengja	GenDeMoG	Tengi	TexMo	TexMo	Coral	Lässig
Goal Contribution							
MLSS Characteristics (Goal A)		✓(OFBiz)	visualization, navigation	✓(JTrac) visualization, navigation, static checking, refactoring	visualization, navigation, static checking, refactoring	✓(JTrac) visualization, navigation, static checking, refactoring	
CLS Mechanisms (Goal B)	visualization, navigation			universal language representation; explicit relation model	universal language representation; explicit relation model	universal language representation; explicit relation model, search-based relation model	traceability, per language representation
MLDE Foundations (Goal C)	traceability; lexical language representation	per language representation, explicit relation model, search-based relation model	lexical language representation; interface relation model	universal language representation; explicit relation model	universal language representation; explicit relation model		
Addressed Research Question	RQ 8, RQ 2, RQ 3, RQ 1, RQ 4	RQ 1, RQ 2, RQ 3, RQ 6, RQ 7, RQ 8	RQ 6, RQ 7	RQ 1, RQ 2, RQ 3, RQ 4, RQ 5, RQ 6, RQ 7	RQ 1, RQ 2, RQ 3, RQ 4, RQ 5, RQ 6, RQ 7	RQ 1, RQ 2, RQ 3, RQ 4, RQ 5, RQ 6, RQ 7, RQ 8	RQ 8, RQ 3
Methodology							
Literature Survey				✓		✓	
Tool Prototyping	✓	✓	✓	✓		✓	✓
Technical Experiment	✓	✓				✓	✓
User Experiment					✓		
Case Study			✓				
Survey						✓	

Table 6.1: Comparison of each paper with respect its contribution to the goals, the addressed research questions, and the methodology applied

*“She said the lift doesn’t work run up the stairs and come
And if you don’t come quick your not gonna see your son
So I grab a bunch of rose, and I started to run
Here I come”*

Barrington Levy, Here I Come

7.1 Discussion & Conclusions

7.1.1 Thesis T1 – Multi-language Software Systems

Multi-language software systems exist. Contemporary software systems mostly consist of multiple heterogeneous development artifacts, which are interrelated.

In this dissertation I investigate characteristics of multi-language software systems (**System Perspective – Goal A**) with respect to language usage and language interrelation. I provide both quantitative and qualitative data characterizing contemporary software systems as multi-language systems with many relations between the development artifacts.

In Section 2.1, I illustrate the language composition of twelve projects on GitHub and the language composition of the Linux kernel. These numbers are new and unpublished data. The result is, that all these systems are multi-language systems. The GitHub projects consist of everything from two to at least 19 languages. The Linux kernel consists of more than 20 languages.

Paper **B** demonstrates that OFBiz is a multi-language system. OFBiz is a large system as it contains multiple thousands of development artifacts (6,522) in various languages such as Java (1,122 files), Groovy (365) files, XML (1,283), etc. Furthermore, a current distribution of OFBiz consists of more than 30 languages, both GPLs and DSLs.

In Paper **B** I conduct a technical experiment applying a search-based relation model to OFBiz' development artifacts. The search-based relation model consists of 22 relation constraint patterns specifying relations in OFBiz. Nineteen of these patterns specify cross-language relations and three specify intra-language relations. The GenDeMoG tool automatically infers an explicit relation model containing 1,737 previously implicit relations. This number, even though already large, is a strict lower bound for the number of actual relations between development artifact in OFBiz. This is because the number of constraint pat-

terns specifying the relations in GenDeMoG's search-based relation model may be incomplete. Additionally, GenDeMoG enables specification and inference of cross-component relations only. The number of relations between development artifacts within components is even larger. The experiment confirms that a large number of implicit relations exist in OFBiz and that they couple its core components quite tightly and circularly.

In Paper **F**, I investigate JTrac, a medium-size software system containing multiple hundreds of development artifacts (372) in Java (140), HTML (66), property files (30), XML (16), JavaScript (8), etc. JTrac is a multi-language software system consisting of more than these five languages. In the technical experiment, I semi-automatically infer a search-based relation model out of JTrac's development artifacts. After inference, the search-based relation model consists of two libraries with five cross-language relation constraints each. With just these ten constraints Coral automatically establishes 4,941 previously implicit cross-language relations. Similar to the OFBiz experiment, the number of inferred relations is a lower bound as the constraint libraries may be incomplete.

These examples illustrate the vast amount of relations coupling development artifacts in multi-language software systems. In both experiments the relations are automatically inferred out of development artifacts represented by syntactic per language representations. Remember, some relations (trace links) between heterogeneous development artifacts are caused by programs processing development artifacts. Even though Papers **A** and **G** are mainly concerned about technological solutions for instrumenting programs causing trace links, they provide evidence that already small development artifacts in model-driven systems are quite heavily interrelated by trace links. The results in Paper **G** demonstrate that models which are transformed to each other are interrelated with trace links whose number corresponds to 40% to 100% of the model sizes.

On top of the quantitative results, Paper **C** demonstrates qualitatively the multi-language property of a small system consisting of development artifacts in six languages ranging from a natural language document over visual languages to textual programming languages. All these development artifacts are related via view or refinement relations, which are explicitly specified in interfaces.

RQ 1 What are the characteristics of language usage in contemporary multi-language software systems? The recapitulated results above demonstrate that contemporary systems are multi-language systems. Small contemporary systems utilize at least two languages, whereas medium-sized and large scale systems contain many more languages. In fact, during my project, I did not come across a contemporary software system, which is constructed out of only a single language.

RQ 2 What are the characteristics of relations between development artifacts within and across language boundaries? The results above illustrate that in contemporary software systems many relations between development artifacts exist. They exist both within languages and across language boundaries. In Paper **B** around half of the 1,737 automatically inferred relations are intra-language relations and the other half are cross-language relations. In Paper **F** the focus is solely on inference of cross-language relations, as the main language in the project is Java and contemporary IDEs provide extensive support for intra-language relations for most GPLs. The lower bound of 4,941 cross-language relations for a medium-sized multi-language software system is

quite impressive.

Another observation concerning relation characteristics is that they are of different type. As indicated earlier in this thesis, I am focusing on basic physical relation types. All automatically established relations specified in the search-based relation models (Papers **B** and **F**) are fixed and string-transformation relations. Those relations resulting from the qualitative case study (Paper **C**) and program instrumentation (Papers **C** and **G**) are free relations or trace links across language boundaries. All the cross-language relations captured in the TexMo's explicit relation model (Paper **D**) are fixed relations.

RQ 3 What type of relations between development artifacts exist in multi-language software systems? In addition to the three relation types (free, fixed, and string-transformation) encoding physical properties of relation ends, there exists a wide variety of domain-specific relation types in multi-language software systems.

Domain-specific relation types encode the reason of existence for relations [99] between artifacts via types. They are extensively discussed in particular by the model-driven development community (see literature in Paper **F** and Section 4.3). However, the physical relation types are fundamental to any other domain-specific relation types, which are always encoded on top of them.

Conclusion. Given the previous discussion, I conclude that thesis **T1** holds. Utilizing various cases, I demonstrate that contemporary software systems are indeed multi-language systems. Additionally, I provide evidence that the many development artifacts in contemporary multi-language software systems are interrelated by different types of relations. The presented numbers for relations in multi-language software systems are all under approximations of the real amount of relations. Including more types of relations in the inference processes easily increases these numbers. These results emphasize the challenges in development of multi-language software systems. Modification and evolution of artifacts in multi-language systems currently requires substantial domain knowledge as implicit relations are frequent and thereby easily broken. Tools maintaining explicit relation models can assist developers in their work.

7.1.2 Thesis T2 – Developer Support

Multi-language software systems can be developed, evolved, customized, and adapted more effectively, when appropriate multi-language development environments are provided. Developers make less errors, are more efficient, and work faster, when supported with a set of cross-language support mechanisms.

To address this thesis, I identified and evaluated features (CLS mechanisms) needed in a tool to support or enhance development of multi-language software systems (**Developer Perspective – Goal B**). To identify such features, I surveyed tools and literature and I evaluate them via a user experiment.

To evaluate the impact of CLS mechanisms to developer's performance, I created TexMo, a prototypical multi-language development environment (Paper **D**), implementing the four elementary CLS mechanisms visualization, navigation, static checking, and refactoring. In a controlled experiment, the prototype is applied as experimental factor with two alternatives. Once with CLS mechanisms enabled and once with CLS mechanisms disabled. During the experiment 22

experiment subjects, equally distributed over two treatment groups, perform three tasks representing typical development and customization tasks on the JTrac system. The first task asks to locate and fix a broken cross-language relation between Java and HTML code. The second task asks for renaming a relation end in a properties file, what breaks a cross-language relation. The subjects should fix the broken relation. The third task asks to replace a fragment of source code, what breaks multiple cross-language relations.

During the experiment I collect quantitative and qualitative data. The quantitative results are discussed in detail in Paper E and the qualitative results are presented in Papers D and F. In short the quantitative results demonstrate that visualization, navigation, static checking, and refactoring when offered across language boundaries are highly beneficial. The CLS mechanisms perceptibly improve the effectiveness of developers working on a multi-language software system. Developers supported with CLS mechanisms find and fix more errors in a shorter time than those in the control group. They perform development tasks on language boundaries more efficiently. They are more effective than developers in a control group as error rate decreases and productivity (working speed) increases. Interestingly, even inexperienced developers provided with CLS mechanisms perform similarly or better than developers experienced in developing multi-language systems.

RQ 4 What are the elementary features in multi-language development environments that support developers of multi-language software systems?

To answer this question I survey tools and literature (Paper F and Chapter 4) to understand the kind of development support they provide. The four features, visualization, navigation, static checking, and refactoring are implemented by all IDEs and by some programming editors for single languages. Consequently, I believe that these four features provided across language boundaries are elementary features (CLS mechanisms) in multi-language development environments.

Of course there are other CLS mechanisms. For example, many IDEs provide code completion for single languages or some recommend code fragments or actions to accomplish certain tasks. Providing other than the four elementary CLS mechanisms across languages, is likely beneficial for developers too. But since none of the experiment subjects (Paper G) mentioned the lack of a certain CLS mechanism, I do not consider them to be elementary. A dedicated survey amongst multi-language software system developers could enhance confidence in which CLS mechanisms actually are elementary. The taxonomy is extensible with respect to CLS mechanisms (domain-specific mechanisms). I consider such a survey as future work.

RQ 5 Do developers of multi-language software systems require certain features or properties?

As stated above, I did not conduct a survey amongst developers of multi-language software system to establish a set of required CLS mechanisms. I consider such a survey future work. But the quantitative results and the qualitative feedback of the experiment evaluating the four elementary CLS mechanisms demonstrate that they not only improve understanding of complex unknown multi-language systems, also developers appreciate the offered CLS mechanisms. They indicate that such mechanisms are beneficial and that such mechanisms are missing in existing IDEs. Some developers in the control group were negatively surprised that current IDEs do not provide

CLS mechanisms in a similar way as they are provided for development with single languages. Especially static checking seems to be required. Developers in the control group were searching for problem markers to find possible errors and were disappointed when there are no such.

Conclusion. I conclude that thesis **T2** holds. The results of the experiment provide evidence that, at least for a fixed set of development tasks on a representative multi-language software system, developers evolve, customize, and adapt multi-language artifacts more effectively, when supported by a multi-language development environment providing CLS mechanisms. Error rates decrease, work speed increases, and understanding of multi-language software systems is facilitated.

However, to increase confidence in the presented results an extended experiment on larger samples including different multi-language software systems and a greater variety of tasks needs to be executed. This is planned as future work. An extended experiment should be conducted in evaluation of an industrial strength multi-language development environments in scenarios of industrial development of multi-language software systems.

7.1.3 Thesis T3 – Tool Builder Support

Provision of a theoretical framework enables tool builders to create multi-language development environments with effective CLS mechanisms tailored to the domain and use cases of the multi-language development environment.

In this thesis, I explore the domain of multi-language development environments and identify major design choices and building blocks for multi-language development environments creation (**Tool Builder Perspective – Goal C**). By surveying related work (Papers **D** and **F** and Chapter 4) I conceptualize design choices in a taxonomy. Thereby, the design space is made explicit and readily available to guide tool builders.

The theoretical framework for development of multi-language development environments is the taxonomy of design choices first presented in Paper **D** and refined in Paper **F** (Figure 6.5).

Contemporary tools implementing CLS mechanisms or implementing support for development with artifacts in different languages appear to apply quite ad hoc solutions. For example, when relations across artifacts in different languages should be represented, encoded, or leveraged, it seems to be a natural decision for most researchers and developers to opt for syntactic per language representations in combination with explicit relation models. This impression seems to be supported by the survey of related work (Section 4). I discuss many publications with respect to the taxonomy of design decisions. Indicated by the number of publications, it seems that the most prominent language representation in research and in tools are syntactic per language representations and the most prominent relation model are explicit relation models. I believe that these choices often do not fit well the domain, which the research or a tool targets. For example, many publications are concerned about maintaining relations between various models in software systems. Actually, the presented explicit relation models are a bad design choice for representing relations in software systems, as such systems are usually open. New development artifacts are created and thereby relations to already existing artifacts are created too.

Explicit relation models are not well suited for maintaining relations in open systems. Due to their static nature they need to be updated whenever new artifacts are created or whenever artifacts are modified. Updating an explicit relation model in evolving systems is quite costly. I realized this when applying TexMo (see Paper **D**). That is, explicit relation models are recommendable for expressing relations in closed static systems, which are not evolving. Software systems, are open dynamic systems. Artifacts can be added, removed and modified. For such, systems any of the other three relation models is more recommendable.

I claim, that one problem for today's tool builders is that they are not aware of all possible design choices in the field of multi-language development environments. I perceive, that the current practices in construction of tools targeting multi-language environments are not ruled by a deeper analysis and comprehension of possible design choices and their impact to the constructed solutions. Instead, tool builders are driven by the growing size of software systems and the growing number of utilized languages in them and the resulting complexity.

I believe, that the presented taxonomy contains all fundamental design choices for multi-language development environments. The taxonomy makes the solution space explicit. Additionally, in combination with the discussion of all my tools, which are all instances of the taxonomy (Papers **A** to **G**), and especially with the discussion of the impact of design choices (Paper **F**), tool builders are aware of all possible solutions. Likely, by careful reflection and weighing advantages and disadvantages of certain decisions before engineering a solution tool builders will create tools, which appropriately fit the domain.

In the following, I discuss my contribution to thesis **T3** relying on argumentation and on analogy. That is, this thesis is not researched by execution of formal experiment or survey in the tool builder community.

The analogy is that during my PhD project I am a representative tool builder creating various prototypes targeting multi-language environments. In this discussion, I focus on the creation of the multi-language development environment prototypes TexMo (Paper **D**) and Coral (Paper **F**). Before creating the tools, I conducted a domain analysis.

TexMo Domain: The task is to create a multi-language development environment prototype implementing the four CLS mechanisms visualization, navigation, static checking, and refactoring, so that a controlled experiment can be executed evaluating the effectiveness of these mechanisms. A priori it is not predefined on which multi-language software system TexMo is applied. The only requirement is to implement a multi-language development environment applicable to any multi-language software system comprising only of development artifacts in textual languages. It is predefined that the tasks in the experiment are concerned about modification of existing development artifacts but not about creation of new artifacts.

Coral Domain: The task is to create a multi-language development environment prototype with particular support for multi-language software systems built on top of application frameworks, such as, persistence, web-frameworks, etc. The relation model should encode relations with respect to framework-specific knowledge. It is required that the solution is generic with respect to the utilized application frameworks and languages. Additionally, the four previously described CLS mechanisms need to be implemented.

Obviously, the TexMo domain is concerned about a closed world, no new

artifacts in an multi-language software system need to be considered, whereas the Coral domain is concerned about an open world.

Before constructing the tools for the respective domains, I consult the taxonomy of multi-language development environment design choices. By reflecting on the possible solutions with respect to language representation, relation models, etc. and by incorporating the domain description, I decide that for the first domain a universal language representation in combination with an explicit relation model is best suited and that for the second domain a syntactic per language representation in combination with a search based-relation model is best suited.

The rationale for the former is that *a*) a universal language representation allows to easily apply the tool to any textual language and *b*) that an explicit relation model is suitable for a closed world. The rationale for the latter is, that a search-based relation model in combination syntactic per language representations allows for central encoding of framework-specific knowledge as cross-language relation constraints in libraries corresponding to the frameworks and languages.

Provided a domain analysis and careful reasoning, the taxonomy guides the tool builder in making design decisions. The effectiveness of the constructed multi-language development environment TexMo is proven by the successful execution of the experiment (Paper G). TexMo implements the requirements and allows the experiment subjects to perform their tasks on an a posteriori selected multi-language software system. The effectiveness of CLS mechanisms is evaluated by application of TexMo, see the discussion in the previous section. That Coral is tailored to its domain (generic encoding of framework-specific knowledge) and its use cases is indicated by the results of a survey in the language builder community (Paper F). In short, the results of the survey are that *a*) many languages are created, *b*) languages are in fact interrelated and thus, artifacts in various languages are interrelated, and *c*) the systems constructed using the created languages are multi-language system, which rely on multiple application frameworks. Furthermore, many language developers provide tools which check for cross-language relations. But most of the provided tools are not generic. That is, whenever a new language is used in multi-language system development the tools have to be adapted to support the changed development architecture. Coral is a generic tool. It is well suited for language integration as it only needs to be parametrized with language specifications and possible cross-language relation constraints. All of the developers indicating that they provide no tools for language integration or that their tools are non-generic, could be efficiently supported by a generically parametrizable MLDE, such as Coral.

Essentially, the design space spanned by the taxonomy provides an answer to the research questions **RQ 6 – RQ 8**.

RQ 6 How can software development artifacts be represented? How to characterize and reference the information in software development artifacts that contribute to relations? I argued for the representation of development artifacts via language representations (lexical, syntactic per language, syntactic per language group, and syntactic universal representation), see Section 3.1 and 4.1. The main reason to apply language representations is to enable for explicit representation of relations between development artifacts.

Depending on the chosen language representation, fragments of development

artifacts can be referenced differently. In Section 3.3.1, the three fundamental reference mechanisms are described. For example, fragments of development artifacts in lexical representation can be referenced by physical positions in the character stream (physical navigation), see for example the locators in the interfaces of Paper C. As syntactic representations offer higher abstractions captured in graph data structures, fragments can be referred by referencing graph structures. Generally, there exist two fundamental mechanisms (path navigation and query evaluation). For example, the explicit relation models in Paper B, Paper D, and Paper G utilize URIs, to reference graph structures by navigating along paths. Both, Papers B and F, present possible search-based relation models which rely on queries to locate relation ends.

RQ 7 How can relations between development artifacts, within and across language boundaries, be represented? How to characterize and formalize relations between development artifacts? The answer to this question is given by the four different kinds of relation models (explicit, tag-based, interface, and search based relation models), which can contain or represent differently typed relations, see Section 3.3.1 for definitions and examples of the models. An interesting observation in Paper F is that explicit, tag-based, and interface relation models represent relation instances interrelating artifacts, whereas search-based relation models specify relations on language level. First after evaluation the concrete relations are established.

I demonstrate the feasibility of explicit relation models (Paper B, Paper D, and Paper G), interfaces (Paper C), and search-based relation models (Paper F). In related work (Section 4.2.2) I point to publications indicating the feasibility of tag-based relation models.

RQ 8 How to automatically reveal or infer relations between different development artifacts? Is this feasible at all? In this thesis I present two possible solutions for automatic inference of relations between development artifacts. First, inference of relations out of development artifacts (Papers B and F) and second, inference of trace links by program instrumentation (Papers A and G). In both cases fixed and string-transformation relations are automatically inferred. It is demonstrated that for these cases and these relation types automatic inference is feasible, since no false positive relations are inferred.

However, the inference of free and domain-specific relations out of development artifacts is problematic. The former cannot be automatically inferred without complete replication of the properties of the relation ends on a per relation basis. Therefore, free relations are better created manually. The latter can, to a great extend be inferred automatically but it is only worthwhile encoding inference code or search-queries when many domain-specific relations of the same type exist in a system. Otherwise, they are also better created manually.

The degree of automatization depends on the quality of the search queries. Especially, the Coral inference tool (Paper F) illustrates that the first phase, the inference of search-based relation models out of development artifacts based on heuristics, i.e., imprecise search queries, generates false positive cross-language relations constraints, which have to be manually removed rendering the first inference phase semi-automatic. So the general answer to the second question of RQ 8 is that relations in multi-language software systems can, depending on their type, be automatically inferred to a large extent.

Conclusion. I conclude that thesis **T3** holds. The presented discussion illustrates that provision of a theoretical framework enables a tool builder to create multi-language development environments with effective CLS mechanisms, tailored to the domain and use cases of the multi-language development environment. I am not aware of any comprehensive framework guiding tool builders similar to the presented taxonomy. However, an experiment or survey increasing the confidence in the positive impact of explicit design decisions in form of the taxonomy on tool builders need to be conducted. I did not execute such an experiment during my thesis as it requires a longer period of research.

7.2 Contribution to Community's Research Agendas

In this section I discuss results of my work and contributions with respect to selected items of two research agendas of the community of model-driven engineering [24, 125].

7.2.1 On the Unification Power of Models [24]

On top of addressing the theses and research questions formulated in Chapter 5, my thesis also contributes to some of the research directions presented by Bézivin [24] in his discussion of the unification power of models. The unification principle is the concept that “*Everything is a model*”. In the following I discuss my contributions with respect to three of his research paths in application of the unification principle. Additionally, with my work, I contribute to one open research issue mentioned in the paper.

Programs as Models

All my papers describing tools with syntactic language representations are also about representation of programs as models. In [24] it is suggested that explicitly linking grammars and metamodels would allow to represent programs via equivalent models. The language representations I provide are either created using EMFText [59, 13] or Xtext [35, 39], two concrete syntax mappers, which link grammar rules and metamodels to generate language specific tools such as editors, parsers, etc. One deliverable of my work is a set of languages as models. For example, together with the tools GenDeMoG, Coral, and TexMo, I provide language representations for Java 5, HTML, XML, properties files, Hibernate mapping files, textual languages, etc.

Traces as Models

The tool Lässig (Paper **G**) generates trace models. The trace models focus on tracing the relation of model elements which are interrelated due to execution of programs transforming them. Currently, Lässig does not generate models of complete stack traces, i.e., sequences of program directives as suggested in [24]. This is not the focus of the corresponding research. However, it can be easily extended to automatically generate complete stack trace models for programs executed on a virtual machine.

Also Tengja (Paper **A**) generates trace models linking model elements in various concrete syntaxes with each other.

Legacy as Models

I do not deploy my tools to automatically migrate software systems to new platforms or environments. However, the tools GenDeMoG and Coral represent artifacts and their relations as models. These models in combination form complete models of software systems. They are quite concrete but higher-level representations, such as, reports or new systems on new platforms can be generated via model transformations. For example, the numbers and the graphs in Paper **B** are generated out of the explicit relation model.

Open Research Issue: Interoperability

As stated by Bézevin, model-driven engineering may be applied to bridge the gap between different tools by representing the data structures, on which tools are editing artifacts, as models. The models should be interchangeable between tools. In my work these tools are for example editors for different languages, which are integrated into the Eclipse IDE. Especially Coral, presented in Paper **F**, implements CLS mechanisms across existing editors. This is realized not by modification of existing tools but by representing development artifacts in different languages and their relations as models. This allows to implement CLS mechanisms on top of these representations and thereby providing interoperability of existing tools.

The concept is generic. Coral could also be applied, reusing the same mechanisms and concepts, to implement the same CLS mechanisms across tools, which are not integrated in an IDE. Conceptually, it is of no importance if language representations and relation models enrich IDE editors or other applications. Perhaps, the appearance of the concrete CLS mechanisms such as visualizations may differ to those presented in Papers **D** and **F**. But nothing prevents integration of language representations, relation models, CLS mechanisms, etc. into an operating system and its graphical user interface. The concepts remain the same but more integration on a technical level is required, since the architectural focus of language-specific IDE editors and other IDE tools is even deeper for entire applications, which are not designed for interoperability. However, this is an interesting topic which I will address in future work.

7.2.2 A Model-based Approach to Language Integration [125]

Also Tomassetti et al. [125] discuss a research agenda on open topics when enhancing IDEs with CLS mechanisms based on syntactic per language representations. In particular, the authors state the following open problems as research agenda. I discuss my contributions along the lines of their problem statements.

Categorize Language Interaction Mechanisms

The authors state, that literature only discusses cross-language relations based on common identifiers (fixed relations) and they notice that alternative possible relation types are not discussed at all. The authors intend to research possible cross-language relation types and formalize them into an ontology.

I contribute to this research topic with my discussion of basic physical relation types in Definition 12. I discuss the relation types and their occurrence in real multi-language software systems in Paper **B**, Paper **D**, and Paper **F**. The relation

types and their representation in the taxonomy of multi-language development environment design decisions can be considered as an ontology. A more elaborate discussion of domain-specific relation types in the literature is provided in Section 4.3.

Techniques to Express Cross-language Constraints

The authors ask for development of multi-language development environments and relation models. Additionally, they seek for their evaluation in different domains and on different multi-language software systems.

I contribute to this research question in various ways. First, I describe and formalize different relation models in the taxonomy of design choices for multi-language development environments (Paper **F**). In Papers **B** and **F**, I present two different search-based relation models. The Coral DSL for description of framework-specific cross-language relation constraints in Paper **F** and the GenDeMoG DSL in Paper **B** encode relations between development artifact via constraints. I discuss a formalization of relation models in general in Section 3.3.1.

Furthermore, I evaluate both tools in two technical experiments on two multi-language software systems, OFBiz and JTrac. Both systems are of different size, medium-size and large scale, and of different application domains. Additionally, I provide evidence that CLS mechanisms in multi-language development environments have a positive impact on developers rendering them more effective and enhance their understanding of multi-language software systems, see Section 7.1.1 and the Papers **D** and **G**.

I agree with the authors that it is time to implement industrial-strength multi-language development environments and evaluate them not in a laboratory environment but over a long period in an industrial setting.

Queries Involving Multiple Languages

The authors envision that a representation of languages as models would permit to query an entire multi-language software systems for all relations across languages interrelating fragments of heterogeneous development artifacts. The authors plan to explore the feasibility and potential of querying such representations for cross-language relations.

My Papers **B** and **F** present two possible realizations for search-based relation models including two languages for formulating queries. The feasibility and potential are demonstrated in the respective papers, where the solutions are applied to existing multi-language software systems. Both search-based relation models are applied to automatically infer cross-language relations and cross-component relations in existing systems. The results demonstrate the large numbers of previously implicit cross-language relations. I discuss alternative query languages, which could be integrated in search-based relation models as related work in Section 4.2.4.

7.3 Future Work

Additionally to the future work mentioned in the previous sections, I discuss open research problems grouped in two categories, research related and technology related future work.

Research: From a research perspective an interesting question is to evaluate the application of an industrial strength multi-language development environment to development of large scale multi-language software systems in diverse industrial settings. The experiment in Paper E indicates the potential of multi-language development environments applied to development of multi-language systems. However, it evaluates the CLS mechanisms on textual languages in one domain and one development phase. It needs to be evaluated how an industrial strength multi-language development environment integrating all development artifacts over all development phases can support different types of developers. For example, a consultant modifying UML diagrams of a complex system, may be overwhelmed, when confronted with the results of static checking telling her that she has broken relations to low level code artifacts or to documentation, of which she is not aware. It is an open question how to adjust CLS mechanisms to be equally useful for different groups involved in development.

Another open research question is how to support distributed development with multi-language development environments. In distributed development, artifacts constituting a system are distributed over various repositories on many different computers. Here, the question is how to best represent and handle, from the perspective of a single multi-language development environment, relations to temporarily non-existing or invisible artifacts. Likely, interfaces are the appropriate relation model in such a setting, as they can either be mirrored locally for all remote artifact or they may have the best properties in distinguishing if a relation is temporally broken due to an invisible remote artifact or if it is really broken due to a local change in a related artifact. This also points to necessary research in CLS mechanisms as a certain level of uncertainty needs to be included in the corresponding mechanisms. For example, a visualization of an existing relation with one invisible relation end should be different from the visualization of a relation to a visible artifact. Even more important, static checking should report different results for the reasons of broken relations. Concerning static checking, it needs also to be researched how to perform static checking when parts of the information required for checking is not available.

Technology: From a technological point of view it needs to be researched how to efficiently mix and interrelate various language representations for the same languages. For example, I plan to extend Coral with a language group representation based on Prolog, Clojure, or Neo4J, similar to the one in SmartEMF [64] and VMQL [119]. The reason is that mapping all per language representations to such a language would allow to perform static checking across languages more efficiently. A promising solution seems to interrelate all language representations for different languages again with relation models. The given relations between the various representations for development artifacts could be leveraged to minimize updates when development artifacts are modified. To maximize efficiency when the number of language representations per language grows, incremental parsing technologies [50] and incremental model update technologies [20, 57] should be integrated into multi-language development environments.

A similar problem that needs further attention is how to efficiently interrelate various relation models containing different representations of the same relations? It seems that explicit and search-based relation models (Papers B and F), explicit and tag-based relation models ([79, 32]), and tag-based relation models

and interfaces (Paper **F**) form combinations of relation models, which might be used interchangeably. Again a relation model keeping relations between interrelated relation models seem to be appropriate.

For the two latter open problems, efficient interrelation of various language representations and efficient interrelation of relation models, a fundamental research question needs to be addressed. How to handle cascading levels of relation models? From an engineering point of view and with respect to this thesis the answer is likely that it is worthless to utilize relation models for more than one level of interrelation. But how to support tool builders developing support mechanisms?

*“Look at me standing
Here on my own again
Up straight in the sunshine*

*No need to run and hide
It’s a wonderful, wonderful life
No need to hide and cry
It’s a wonderful, wonderful life”*

Seed/Black Wonderful Life

Bibliography

- [1] Zend Technologies Ltd.: Taking the Pulse of the Developer Community. static.zend.com/topics/zend-developer-pulse-survey-report-0112-EN.pdf, seen: Feb. 2012
- [2] THE OPEN SOURCE DEVELOPER REPORT – 2010 Eclipse Community Survey. eclipse.org/org/press-release/20100604_survey2010.php (2011), seen: Mar. 2012
- [3] Abrial, J.R., Glässer, U. (eds.): Rigorous Methods for Software Construction and Analysis, Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday, Lecture Notes in Computer Science, vol. 5115. Springer (2009)
- [4] Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model Traceability. *IBM Systems Journal* 45(3), 515 –526 (2006)
- [5] Alanen, M., Porres, I.: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels (2004)
- [6] Alfaro, L.d., Henzinger, T.A.: Interface Theories for Component-Based Design. In: EMSOFT (2001)
- [7] Altheide, F., Dörr, H., Schürr, A.: Requirements to a Framework for Sustainable Integration of System Development Tools. In: Proc. of the 3rd European Systems Engineering Conference (EuSEC. pp. 53–57 (2002)
- [8] Anderson, K.M., Taylor, R.N., Whitehead, Jr., E.J.: Chimera: Hypermedia for Heterogeneous Software Development Enviroments. *ACM Trans. Inf. Syst.* 18 (July 2000)
- [9] Antkiewicz, M., Bartolomei, T.T., Czarnecki, K.: Fast Extraction of High-quality Framework-specific Models from Application Code. *Autom. Softw. Eng.* 16(1), 101–144 (2009)
- [10] Antkiewicz, M., Czarnecki, K.: Framework-Specific Modeling Languages with Round-Trip Engineering. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) *MoDELS’06. Lecture Notes in Computer Science*, vol. 4199. Springer (2006)

- [11] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering Traceability Links between Code and Documentation. *IEEE Trans. Softw. Eng.* 28(10), 970–983 (Oct 2002), <http://dx.doi.org/10.1109/TSE.2002.1041053>
- [12] Aranega, V., Etien, A., Dekeyser, J.L.: Using an Alternative Trace for QVT. *Electronic Communications of the EASST* 42 (2011)
- [13] Assmann, U., Bartho, A., Bürger, C., Cech, S., Demuth, B., Heidenreich, F., Johannes, J., Karol, S., Polowinski, J., Reimann, J., Schroeter, J., Seifert, M., Thiele, M., Wende, C., Wilke, C.: DropsBox: the Dresden Open Software Toolbox. *Software & Systems Modeling* pp. 1–37 (2012), <http://dx.doi.org/10.1007/s10270-012-0284-6>
- [14] Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. In: *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. pp. 19–33. *UML'01*, Springer-Verlag, London, UK, UK (2001), <http://dl.acm.org/citation.cfm?id=647245.719475>
- [15] Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. *IEEE Softw.* 20(5), 36–41 (Sep 2003), <http://dx.doi.org/10.1109/MS.2003.1231149>
- [16] Badros, G.J.: JavaML: A Markup Language for Java Source Code. *Comput. Netw.* 33 (June 2000)
- [17] Barbier, F., Eveillard, S., Youbi, K., Guitton, O., Perrier, A., Cariou, E.: Model-Driven Reverse Engineering of COBOL-Based Applications, pp. 283–299. Morgan Kaufmann (2010), <http://www.sciencedirect.com/science/article/B6MH5-508779H-7/2/6b3199748873fdfa42e3a892ba1b4d19>
- [18] Becks-Malorny, U.: *Kandinsky*. Benedikt Taschen Verlag GmbH (2003)
- [19] Bengtson, E., Roth, D.: Understanding the Value of Features for Coreference Resolution. In: *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)* (10 2008), <http://cogcomp.cs.illinois.edu/papers/BengtsonRo08.pdf>
- [20] Bergmann, G., Horváth, A., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental Evaluation of Model Queries over EMF Models. In: *Proceedings of the 13th International Conference on Model-driven Engineering Languages and Systems: Part I*. pp. 76–90. *MODELS'10*, Springer-Verlag, Berlin, Heidelberg (2010), <http://dl.acm.org/citation.cfm?id=1926458.1926467>
- [21] Bernstein, P.A.: Applying Model Management to Classical Meta Data Problems. In: *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)* (2003)
- [22] Bettini, L.: An Eclipse-based IDE for Featherweight Java Implemented in Xtext. In: *Proc. ECLIPSE-IT*. pp. 14–28 (2010), <http://2010.eclipse-it.org/proceedings/>
- [23] Bettini, L.: A DSL for Writing Type Systems for Xtext Languages. In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. pp. 31–40. *PPPJ '11*, ACM, New York, NY, USA (2011), <http://doi.acm.org/10.1145/2093157.2093163>

- [24] Bézivin, J.: On the Unification Power of Models. *Software and System Modeling* 4(2), 171–188 (2005)
- [25] Bragdon, A., Reiss, S.P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola, Jr., J.J.: Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. ICSE '10*, ACM, New York, NY, USA (2010)
- [26] Bragdon, A., Zeleznik, R., Reiss, S.P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola, Jr., J.J.: Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. pp. 2503–2512. *CHI '10*, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1753326.1753706>
- [27] Branco, M.C., Troya, J., Czarnecki, K., Küster, J.M., Völzer, H.: Matching Business Process Workflows across Abstraction Levels. In: *France et al. [47]*, pp. 626–641
- [28] Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In: *Proc. of the IEEE/ACM International Conference on Automated Software Engineering* (2010)
- [29] Bézivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2004)
- [30] Chen, N., Johnson, R.: Toward Refactoring in a Polyglot World: Extending Automated Refactoring Support across Java and XML. In: *Proceedings of the 2nd Workshop on Refactoring Tools* (2008)
- [31] Colburn, T.: *Philosophy and Computer Science. Explorations in Philosophy*, M.E. Sharpe (2000), <http://books.google.dk/books?id=luF4ElMxqg4C>
- [32] Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal* 45(3), 621–646 (2006)
- [33] Dmitriev, S.: *Language Oriented Programming: The Next Programming Paradigm*. Tech. rep., JetBrains (2004), <http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf>
- [34] Drivalos, N., Kolovos, D.S., Paige, R.F., Fernandes, K.J.: *Software Language Engineering*. chap. *Engineering a DSL for Software Traceability*, pp. 151–167. Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-00434-6_10
- [35] Efftinge, S., Völter, M.: OAW XText: A Framework for Textual DSLs. In: *Workshop on Modeling Symposium at Eclipse Summit*. vol. 32 (2006)
- [36] Egyed, A.: Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Trans. Software Eng.* 37(2), 188–204 (2011)

- [37] Erlikh, L.: Leveraging Legacy System Dollars for E-Business. *IT Professional* 2 (May 2000)
- [38] Estublier, J., Vega, G., Ionita, A.D.: Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications. In: *MoDELS*. pp. 69–83 (2005)
- [39] Eysholdt, M., Behrens, H.: Xtext: Implement your Language Faster than the Quick and Dirty Way. In: *Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages and Applications Companion*. pp. 307–309. *SPLASH '10*, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1869542.1869625>
- [40] Favre, J.M., Musset, J.: *Rétro-ingénierie Dirigée par les Métamodèles: Concepts, Outils, Méthodes*. In: *Deuxième édition des Journées sur l'Ingénierie Dirigée par les Modèles* (June 2006)
- [41] Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, Kazman, R., Klein, M., Northrop, L., Schmidt, D., Sullivan, K., Wallnau, K.: *Ultra-Large-Scale Systems*. Carnegie Mellon University (2006)
- [42] de Figueiredo Carneiro, G., Mendonça, M.G., Magnavita, R.C.: An Experimental Platform to Characterize Software Comprehension Activities Supported by Visualization. In: *ICSE Companion* (2009)
- [43] Fjeldberg, H.C.: *Polyglot Programming – A business perspective*. Master's thesis, Norwegian University of Science and Technology (2008)
- [44] Ford, N.: *The Productive Programmer*. O'Reilly, first edn. (2008)
- [45] Fowler, M.: *Domain-Specific Languages*. Addison-Wesley Signature Series, Pearson Education (2010), http://books.google.dk/books?id=ri1muolw_YwC
- [46] Fowler, M.: *Language Workbenches: The Killer-App for Domain Specific Languages?* (2005), <http://martinfowler.com/articles/languageWorkbench.html>
- [47] France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.): *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7590*. Springer (2012)
- [48] Freude, R., Königs, A.: Tool Integration with Consistency Relations and their Visualisation. In: *ESEC/ FSE Workshop on Tool Integration in System Development* (2003)
- [49] Ghosh, D.: *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edn. (2010)
- [50] Goldschmidt, T.: *Software Language Engineering*. chap. Towards an Incremental Update Approach for Concrete Textual Syntaxes for UUID-Based Model Repositories, pp. 168–177. Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-00434-6_11

- [51] Grammel, B., Kastenholz, S.: A Generic Traceability Framework for Facet-based Traceability Data Extraction in Model-driven Software Development. In: Proceedings of the 6th ECMFA Traceability Workshop. pp. 7–14. ECMFA-TW '10, ACM, New York, NY, USA (2010)
- [52] Grammel, B., Kastenholz, S., Voigt, K.: Model matching for trace link generation in model-driven software development. In: France, R., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 7590, pp. 609–625. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-33666-9_39
- [53] Groenewegen, D.M., Hemel, Z., Visser, E.: Separation of Concerns and Linguistic Integration in WebDSL. *IEEE Software* 27(5) (2010)
- [54] Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-Modelling: From Theory to Practice. In: Proc. of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I (2010)
- [55] Gómez, P., Sánchez, M., Florez, H., Villalobos, J.: Co-Creation of Models and Metamodels for Enterprise Architecture Projects. XM 2012 - Extreme Modeling Workshop (2012)
- [56] Halasz, F.G., Schwartz, M.D.: The Dexter Hypertext Reference Model. *Commun. ACM* 37(2) (1994)
- [57] Hegedüs, Á., Horváth, Á., Ráth, I., Varró, D.: Query-Driven Soft Interconnection of EMF Models. In: France et al. [47], pp. 134–150
- [58] Heidenreich, F.: Towards Systematic Ensuring Well-Formedness of Software Product Lines. In: In Proceedings of the 1st Workshop on Feature-Oriented Software Development. pp. 69–74. ACM, New York, NY, USA (oct 2009)
- [59] Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications. pp. 114–129. ECMDA-FA '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02674-4_9
- [60] Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: Proc. of the 2nd International Conference on Software Language Engineering (SLE 2009), Revised Selected Papers (2010)
- [61] Heidenreich, F., Johannes, J., Zschaler, S.: Aspect Orientation for Your Language of Choice. In: Workshop on Aspect-Oriented Modeling (AOM at MoDELS) (2007)
- [62] Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping Features to Models. In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08). pp. 943–944. ACM, New York, NY, USA (May 2008)
- [63] Henriksson, J., Johannes, J., Zschaler, S., Aßmann, U.: Reuseware - Adding Modularity to Your Language of Choice. *Journal of Object Technology* 6(9) (2007)

- [64] Hesselund, A.: SmartEMF: Guidance in Modeling Tools. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (2007)
- [65] Hesselund, A.: Domain-Specific Multimodeling. Ph.D. thesis, IT University of Copenhagen (2009)
- [66] Hesselund, A., Sestoft, P.: Flow Analysis of Code Customizations. In: Proceedings of the 22nd European conference on Object-Oriented Programming. pp. 285–308. ECOOP '08, Springer-Verlag, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-70592-5_13
- [67] Hesselund, A., Wąsowski, A.: Interfaces and Metainterfaces for Models and Metamodels. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (2008)
- [68] Holst, W.: Meta: A Universal Meta-Language for Augmenting and Unifying Language Families, Featuring Meta(oopl) for Object-Oriented Programming Languages. In: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (2005)
- [69] Hußmann, H., Demuth, B., Finger, F.: Modular Architecture for a Toolset Supporting OCL. In: Evans, A., Kent, S., Selic, B. (eds.) UML. Lecture Notes in Computer Science, vol. 1939, pp. 278–293. Springer (2000)
- [70] Jarzabek, S.: Specifying and Generating Multilanguage Software Development Environments. *Softw. Eng. J.* 5(2), 125–137 (Apr 1990), <http://dx.doi.org/10.1049/sej.1990.0015>
- [71] Jemerov, D.: Implementing Refactorings in IntelliJ IDEA. In: Proceedings of the 2nd Workshop on Refactoring Tools. pp. 13:1–13:2. WRT '08, ACM, New York, NY, USA (2008), <http://doi.acm.org/10.1145/1636642.1636655>
- [72] Jouault, F.: Loosely Coupled Traceability for ATL. In: In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability. pp. 29–37 (2005)
- [73] Jouault, F., Vanhooff, B., Bruneliere, H., Doux, G., Berbers, Y., Bezivin, J.: Inter-DSL Coordination Support by Combining Megamodeling and Model Weaving. In: Proceedings of the 2010 ACM Symposium on Applied Computing (2010)
- [74] Kats, L.C.L., Visser, E.: The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) OOPSLA. pp. 444–463. ACM (2010), <http://dblp.uni-trier.de/db/conf/oopsla/oopsla2010.html#KatsV10>
- [75] Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels (2008)
- [76] Kleppe, A.G.: A Language Description is More than a Metamodel. In: Fourth International Workshop on Software Language Engineering, Nashville, USA. megaplanet.org, Grenoble, France (October 2007)
- [77] Kolovos, D., Rose, L., Paige, R., Polack, F.A.C.: The Epsilon Book. Structure 178 (2010)

- [78] Kolovos, D.S.: Establishing Correspondences between Models with the Epsilon Comparison Language. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA. Lecture Notes in Computer Science, vol. 5562, pp. 146–157. Springer (2009)
- [79] Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On-Demand Merging of Traceability Links with Models. (2006)
- [80] Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Proceedings of the 1st international conference on Theory and Practice of Model Transformations. pp. 46–60. ICMT '08, Springer-Verlag, Berlin, Heidelberg (2008)
- [81] Kullbach, B., Winter, A., Dahm, P., Ebert, J.: Program Comprehension in Multi-Language Systems. In: Proceedings of the Working Conference on Reverse Engineering (WCRE'98) (1998)
- [82] Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co., Greenwich, CT, USA (2003)
- [83] Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. *Commun. ACM* 21 (June 1978)
- [84] Lotufo, R., She, S., Berger, T., Czarnecki, K., Wąsowski, A.: Evolution of the Linux kernel variability model. In: SPLC'11. LNCS, vol. 6287. Springer (2010)
- [85] Mahé, V., Jouault, F., Bruneliere, H.: Megamodeling Software Platforms: Automated Discovery of Usable Cartography from Available Metadata (2009)
- [86] McAffer, J., VanderLei, P., Archer, S.: OSGi and Equinox: Creating Highly Modular Java Systems. Addison-Wesley Professional, 1st edn. (2010)
- [87] McMahon, L.E.: Sed—A Non-interactive Text Editor. Computer Science Technical Report (77) (1978)
- [88] Mens, T., Gorp, P.V.: A Taxonomy of Model Transformation. *Electr. Notes Theor. Comput. Sci.* 152, 125–142 (2006)
- [89] Merkle, B.: Textual Modeling Tools: Overview and Comparison of Language Workbenches. In: Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages and Applications Companion. pp. 139–148. SPLASH '10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1869542.1869564>
- [90] Meyers, S.: Difficulties in Integrating Multiview Development Systems. *IEEE Softw.* 8 (1991)
- [91] Murphy-Hill, E., Parnin, C., Black, A.P.: How We Refactor, and How We Know it. In: Proc. of the 31st International Conference on Software Engineering (2009)
- [92] Mäder, P., Cleland-Huang, J.: A Visual Language for Modeling and Executing Traceability Queries. *Software & Systems Modeling* pp. 1–17 (2012), <http://dx.doi.org/10.1007/s10270-012-0237-0>

- [93] Netta Aizenbud-Reshef, Richard F. Paige, Julia Rubin, Yael Shaham-Gafni and Dimitrios S. Kolovos: Operational Semantics for Traceability. In: European Conference in MDA. pp. 7–14 (2005)
- [94] Nørmark, K.: Elucidative Programming. *Nord. J. Comput.* 7(2), 87–105 (2000)
- [95] Nørmark, K., Østerbye, K.: Representing Programs as Hypertext. In: Lund Institute of Technology, Lund University. pp. 11–24 (1994)
- [96] Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, V1.1. <http://www.omg.org/spec/QVT/1.1/> (Jan 2011)
- [97] Oldevik, J., Neple, T.: Traceability in Model to Text Transformations. In: Proceedings of ECMDA Traceability Workshop (ECMDA-TW) (2006)
- [98] Østerbye, K., Nørmark, K.: An Interaction Engine for Rich Hypertexts. In: Ritchie, I., Guimarães, N. (eds.) ECHT. pp. 167–176. ACM (1994)
- [99] Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering. *Softw. Syst. Model.* 10 (October 2011)
- [100] Paige, R.F., Olsen, G., Kolovos, D., Zschaler, S., Power, C.: Building Model-Driven Engineering Traceability Classifications. In: 4th ECMDA Traceability Workshop (2008)
- [101] Paige, R.F., Varró, D.: Lessons Learned from Building Model-driven Development Tools. *Software and System Modeling* 11(4), 527–539 (2012)
- [102] Pfeiffer, J.H., Gurd, J.R.: Visualisation-based Tool Support for the Development of Aspect-oriented Programs. In: Filman, R.E. (ed.) AOSD. pp. 146–157. ACM (2006)
- [103] Rahm, E., Bernstein, P.A.: A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal* 10(4), 334–350 (Dec 2001), <http://dx.doi.org/10.1007/s007780100057>
- [104] Ráth, I., Ökrös, A., Varró, D.: Synchronization of Abstract and Concrete Syntax in Domain-specific Modeling Languages. *Software and System Modeling* (2009), available online first.
- [105] Renggli, L., Denker, M., Nierstrasz, O.: Language Boxes: Bending the Host Language with Modular Language Changes. In: *Software Language Engineering: Second International Conference, SLE 2009* (2010)
- [106] Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding Languages without Breaking Tools. In: *Proceedings of the 24th European Conference on Object-oriented Programming*. pp. 380–404. ECOOP’10, Springer-Verlag, Berlin, Heidelberg (2010), <http://dl.acm.org/citation.cfm?id=1883978.1884006>
- [107] Ribeiro, M., Pacheco, H., Teixeira, L., Borba, P.: Emergent Feature Modularization. In: *Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages and Applications*

- Companion. pp. 11–18. SPLASH '10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1869542.1869545>
- [108] Richters, M., Gogolla, M.: OCL: Syntax, Semantics, and Tools. In: Clark, T., Warmer, J. (eds.) *Object Modeling with the OCL*. Lecture Notes in Computer Science, vol. 2263, pp. 42–68. Springer (2002)
 - [109] Salay, R., Mylopoulos, J., Easterbrook, S.: Using Macromodels to Manage Collections of Related Models. In: *Proc. of the 21st International Conference on Advanced Information Systems Engineering* (2009)
 - [110] Schink, H., Kuhlemann, M., Saake, G., Lämmel, R.: Hurdles in Multi-Language Refactoring of Hibernate Applications (2011)
 - [111] Schulze, G., Chimiak-Opoka, J., Arlow, J.: An Approach for Synchronizing UML Models and Narrative Text in Literate Modeling. In: France et al. [47], pp. 595–608
 - [112] Shvaiko, P., Euzenat, J.: A Survey of Schema-Based Matching Approaches. *J. Data Semantics IV* 3730, 146–171 (2005)
 - [113] Sommerville, I.: *Software Engineering*. International Computer Sciences Series, Addison Wesley, Harlow, UK, 8th edn. (2006)
 - [114] Spanoudakis, G., Zisman, A.: Software Traceability: A Roadmap. In: *Handbook of Software Engineering and Knowledge Engineering*. pp. 395–428. World Scientific Publishing (2004)
 - [115] Stahl, T., Völter, M., Czarnecki, K.: *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons (2006)
 - [116] Stallman, R.M.: EMACS The Extensible, Customizable Self-documenting Display Editor. In: *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*. pp. 147–156. ACM, New York, NY, USA (1981), <http://doi.acm.org/10.1145/800209.806466>
 - [117] Standish, T.A.: An Essay on Software Reuse. *IEEE Trans. Software Eng.* (1984)
 - [118] Steinberger, M., Waldner, M., Streit, M., Lex, A., Schmalstieg, D.: Context-Preserving Visual Links. *IEEE Transactions on Visualization and Computer Graphics (InfoVis'11)* 17(12) (2011)
 - [119] Störrle, H.: VMQL: A Visual Language for Ad-hoc Model Querying. *J. Vis. Lang. Comput.* 22(1), 3–29 (Feb 2011), <http://dx.doi.org/10.1016/j.jvlc.2010.11.004>
 - [120] Strein, D., Kratz, H., Lowe, W.: Cross-Language Program Analysis and Refactoring. In: *Proc. of the 6th IEEE International Workshop on Source Code Analysis and Manipulation* (2006)
 - [121] Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An Extensible Meta-Model for Program Analysis. *IEEE Trans. Softw. Eng.* 33 (September 2007)
 - [122] Sufrin, B.: Formal Specification of a Display-oriented Text Editor. *Science of Computer Programming* 1(3), 157 – 202 (1982), <http://www.sciencedirect.com/science/article/pii/0167642382900144>

- [123] Synytskyy, N., Cordy, J.R., Dean, T.R.: Robust Multilingual Parsing Using Island Grammars. In: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research. pp. 266–278. CASCON '03, IBM Press (2003), <http://dl.acm.org/citation.cfm?id=961322.961364>
- [124] Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.M.: N Degrees of Separation: Multi-dimensional Separation of Concerns. In: Proceedings of the 21st International Conference on Software Engineering. pp. 107–119. ICSE '99, ACM, New York, NY, USA (1999)
- [125] Tomassetti, F., Vetro, A., Torchiano, M., Völter, M., Kolb, B.: A Model-based Approach to Language Integration (2013), <http://porto.polito.it/2506234/>, to appear in MISE@ICSE
- [126] Voigt, K.: Semi-automatic Matching of Heterogeneous Model-based Specifications. In: Engels, G., Luckey, M., Pretschner, A., Reussner, R. (eds.) Software Engineering (Workshops). LNI, vol. 160, pp. 537–542. GI (2010)
- [127] Voigt, K., Ivanov, P., Rummler, A.: MatchBox: Combined Meta-model Matching for Semi-automatic Mapping Generation. In: Proceedings of the 2010 ACM Symposium on Applied Computing. pp. 2281–2288. SAC '10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1774088.1774563>
- [128] Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013), <http://dslbook.org>
- [129] Völter, M., Solomatov, K.: Language Modularization and Composition with Projectional Language Workbenches Illustrated with MPS. Software Language Engineering, SLE (2010)
- [130] Wagner, S., Deissenboeck, F.: Abstractness, Specificity, and Complexity in Software Design. In: Proc. of the 2nd International Workshop on the Role of Abstraction in Software Engineering (2008)
- [131] Waldner, M., Puff, W., Lex, A., Streit, M., Schmalstieg, D.: Visual Links Across Applications. In: Proc. of Graphics Interface (2010)
- [132] Wende, C.: Language Family Engineering – with Features and Role-Based Composition. Ph.D. thesis, Technische Universität Dresden (2012), <http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-88985>
- [133] Wilke, C., Bartho, A., Schroeter, J., Karol, S., Aßmann, U.: Elucidative Development for Model-Based Documentation. In: Furia, C., Nanz, S. (eds.) Objects, Models, Components, Patterns, Lecture Notes in Computer Science, vol. 7304, pp. 320–335. Springer Berlin / Heidelberg (2012)
- [134] Winkler, S., Pilgrim, J.: A Survey of Traceability in Requirements Engineering and Model-driven Development. Softw. Syst. Model. 9(4), 529–565 (Sep 2010), <http://dx.doi.org/10.1007/s10270-009-0145-0>

- [135] Xing, Z., Stroulia, E.: Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In: Proc. of the 22nd IEEE International Conference on Software Maintenance (2006)
- [136] Yie, A., Wagelaar, D.: Advanced traceability for ATL. In: 1st International Workshop on Model Transformation with ATL. pp. 78–87 (2009)

8

An Aspect-based Traceability Mechanism for Domain Specific Languages – ECMFA-TW'10 (Paper A)

An Aspect-based Traceability Mechanism for Domain Specific Languages

Rolf-Helge Pfeiffer and Andrzej Wasowski

IT University, Software Development Group,
Rued Langgaards Vej 7, 2300 Copenhagen, Denmark
{ropf,wasowski}@itu.dk

Abstract. Development environments for domain specific modeling usually represent elements of visual models as objects when in memory and as XML elements when persisted. Visual models are editable using different kinds of editors, and both the in-memory representations and the serialization syntax can be manipulated by automatic tools. We present *Tengja*, a toolkit, that automatically collects the traces between model elements in abstract, visual, and serialization syntax. Once the trace model is established by Tengja it can be used by other applications to synchronize representations involved, or to navigate across models. We demonstrate the toolkit by implementing a simple navigation support on top of it.

Keywords: Model-Driven Software Development, Traceability, Aspect-Oriented Software Development

1 Introduction

Modeling languages can be classified into visual and textual ones. Textual languages, such as XML, use alphabet characters to represent models. The *Unified Modeling Language* (UML) is an example of a visual modeling language. The language specification, gives its syntax using visual elements [1]. Visual and textual modeling languages are used on different levels of abstraction. Often, the more abstract a model is, the more likely a visual notation is used.

Somewhat controversially, we claim that visual models do not exist in practice. Instead textual models are interpreted by tools and represented visually. The visualizations vary, depending on the editors used; ranging from text processors, via structured forms and trees, to fully fledged diagramming editors. Often several views are used for the same model.

State of the art frameworks, such as *Eclipse Modeling Framework* (EMF) and *Graphical Modeling Framework* (GMF), allow automatic generation of rich editors for domain specific languages (DSLs). The editors are used by developers to produce new software artifacts—for instance specifications of product variants in model-driven product line architectures [16]. Since the editors rely solely on a description of abstract syntax

supplemented with visual diagram meta-data, a large degree of code reuse can be achieved.

Inclusion in mature development processes, imposes high requirements on modeling editors. Not only should they support developers in the same manner as usual programming environments, but also provide modeling-specific functionalities. These, among others, require supporting concurrent editing of multiple views of an interconnected network of models by multiple developers using different physical workstations. Since such applications often need to rely on serialized versions of models, we approach the question of traceability of model elements to their serialized representations.

A modern DSL editor distinguishes three representations:

- The *serialization concrete syntax* (*serialization syntax* for short) is the persistent representation of models. It often takes form of an XML file adhering to a particular schema. Tools use serialization syntax at least for storage, but also for transformation, versioning, etc.
- The *abstract syntax* is the object graph representing the domain specific model in memory. It takes the form of an object model adhering to a particular class model (the meta-model). The abstract syntax is what most researchers consider 'a model'.
- The *visual concrete syntax* (or *visual syntax* for short) is the diagram shown to the user in a visual editor. The visual syntax is how the users perceive models.

In programming languages the visual and the serialization syntax coincide: developers work with the textual representation, which is directly stored in files. For domain specific modeling languages this is rarely so: The *Eclipse DSL toolkit* [9], *Generic Modeling Environment* [6], *MetaEdit+* [10] and *Microsoft DSL Tools* [7], all hide the serialization syntax from users. In all these tools editors work with visual syntax, while abstract syntax and serialization syntax are used for transformation, synchronization and storage.

Modeling frameworks uniformly support loading and persisting models—a form of model-to-model traceability between the serialization and the abstract syntax. For the visual syntax there are well developed traceability mechanisms linking it to the abstract syntax: all frameworks use them to realize the *Model-View-Controller (MVC)* pattern [15], where changes to the visual syntax are immediately synchronized with the abstract syntax. This is a form of element-to-element traceability between the abstract and visual models. However, presently, element-to-element traceability between the abstract and the serialization syntax is not supported.

The ability to link abstract and serialization syntax at the element granularity enables the following use cases:

- *Model debugging* — a developer can navigate from visual syntax elements directly to XMI representation, in order to inspect the values saved by the editor. This is useful to debug various phases of the editor, to debug the

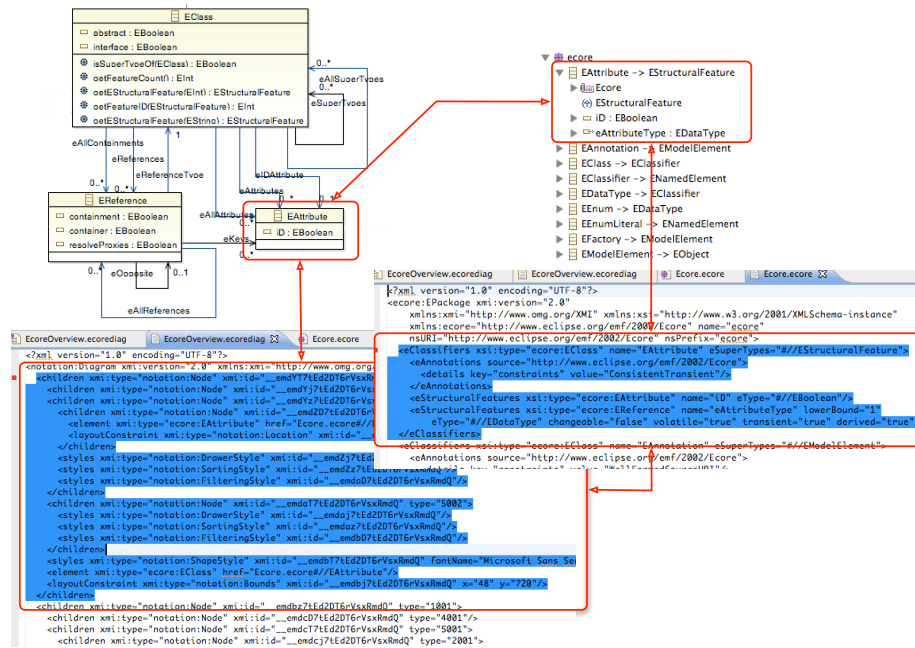


Fig. 1. A visual model element, the corresponding data model element and textual representations of both

models, and to debug model transformations. This use case is supported by our prototype (below).

- *Monitoring and navigating cross-model soft references* — stating soft references across models at the level of serialization syntax allows using external XML processing tools and XML transformations to manage models, while still maintaining inter-model consistency. Traceability between abstract syntax elements and the serialization syntax allows monitoring external changes to the abstract syntax, or to the concrete syntax, for example in order to notify the user about locks to a part of the model, or about concurrent updates.
- *Interactive creation of language independent soft references* — extending the GUI of interactive editors to support definition of soft references and ‘anchors’ for higher-level trace-based tools. A user could indicate a model element, and annotate it with a dependency constraint on another system component. This dependency can be stored as a dependency between the serialized versions of the element and that component, which can be processed by a traceability tool that is oblivious to the specific notion of models and components, but just knows about the concept of anchor and dependency. We are presently working on building such a tool for tracing between heterogeneous artifacts.

In this paper we present *Tengja*¹, a toolkit for element-to-element tracing from visual and abstract syntax to their serialized versions. Tengja is implemented as an aspect in AspectJ, which automatically establishes element-to-element links, by analyzing the process of saving the model. Our toolkit is non-invasive and highly reusable due to its aspect oriented nature. It works for existing models in EMF and GMF. Neither the models nor the editors need to be modified. It works with all Eclipse model editors (including tree editors, and GMF generated domain specific editors), as long as they rely on the standard persistence mechanism.

As a proof of concept, which demonstrates the effectiveness of the framework, we develop a simple debugging facility using the Tengja-toolkit. It extends the current Eclipse editors, by introducing an ability to highlight an element and request its view in other representations. For example, in Figure 1 the user highlights **EAttribute** element of the model opened in a diagram editor of GMF (left top). On user request, the framework automatically opens three other editors highlighting the corresponding model parts in each of them (a tree view to the right, an Ecore XML model to bottom-right and an Ecore diagram model serialized to XML to bottom-left). The user can then proceed to debug the modeling editors, the serialization mechanism, or the model itself, by inspecting or modifying the representations shown. As mentioned above the same mechanism could be used to trigger warnings, errors or updates depending on automatic or concurrent changes to the serialized or abstract syntax. We intend to investigate such applications in future.

We proceed as follows. Section 2 provides background on modeling languages and visualizations in Eclipse. In Section 3 we present Tengja itself, and evaluate it in Section 4. Sections 5–7 discuss our solution, compare it to published sources, indicate future research directions and conclude.

2 Models in Eclipse

In this paper we work with EMF and GMF—both visual modeling components of the Eclipse DSL toolkit. Both frameworks are representative of modern environments for model-driven software development. Eclipse, GMF and EMF together enable developers to easily define their DSLs and to generate specialized editors for them.

Eclipse provides three different kinds of model editors: diagram editors (**DiagramDocumentEditor** as part of GMF), structured tree editors (**Tree** used by EMF), and text editors. Diagram editors and tree editors allow interacting with visual syntax. Text editors allow for editing models in serialization syntax or in other textual representations. Eclipse’s modeling package contains examples of a few predefined editors, such like the UML class diagram editor. In Figure 1 the left-topmost model is shown in a diagram editor (in fact in the class diagram

¹ Tengja, Icelandic for *connect*, was chosen to avoid conflicts with “connects”, “connections”, and “connectors” appearing frequently in model-driven development literature.

editor), the right-topmost one in a tree editor, while the two others are shown in usual text editors.

Presently, the *XML Metadata Interchange* (XMI) format is used to persist models in Eclipse. The following gives an overview over the artifacts that are used to store the relevant information of a graphical model using the example of the Ecore Metamodeling Language [2].

The model in the top-right of Figure 1 is an excerpt of the `EcoreOverview.ecore` data model presented in a tree editor. Its serialization to an XMI file is shown underneath (bottom-right). Notice, that since the tree editor contains no model specific layout information, the serialization only contains data model elements. To the left (top-left) a visual diagram of the same model is shown using UML-like class diagram syntax. In physical world this image of boxes and lines is the actual model as it would appear on paper. However from the tool perspective this is just one possible visualization of the model. Finally, the bottom-left of the figure shows the serialization of the layout information of the class diagram.

Eclipse separates the visual information model from the actual data model, spreading their persistent representations over two files. These are integrated together by modeling editors following the MVC pattern [15]. Figure 2² illustrates the steps performed when loading, editing, and saving the `EcoreOverview.ecorediag` model. When an editor is opened, the visual information model is loaded first, then the data model is loaded, and both are interpreted, before the model is presented to the user (the left part of Figure 2).

3 Tengja

Our aim is to provide an extension to Eclipse, which recovers the links between the serialization syntax and the abstract syntax of models. Since in general, visual modeling languages are just graphs with different node types and different edge types, frameworks deploy graph traversal algorithms to persist the models.

The gear-wheels in Figure 3(a) symbolize the standard persistence mechanism of Eclipse serializing both the visual diagram information and the data model one after another. Both GMF and EMF use the mechanism implemented in the `org.eclipse.emf.ecore.xmi.impl` package for this. The XMI representation is generated by class `XMLSaveImpl`. It traverses the in-memory object graph and feature-wise generates the corresponding XML elements. If there exist corresponding, less abstract models, for instance corresponding data models, the save mechanism is called iteratively on all these dependent models.

Initially, we had anticipated to find a compositional bottom-up graph traversal algorithm that generates a block of serialization syntax for each model element. In reality the translation of each model element is scattered over multiple methods that are called sequentially. The model graph is not traversed bottom-up from simple element leaves to the top root. Rather, the persistent model is constructed sequentially starting from the root model element. The process does

² Here and in Figures 2 and 3 are artifacts shown to illustrate concepts. Details in them are not supposed to be legible.

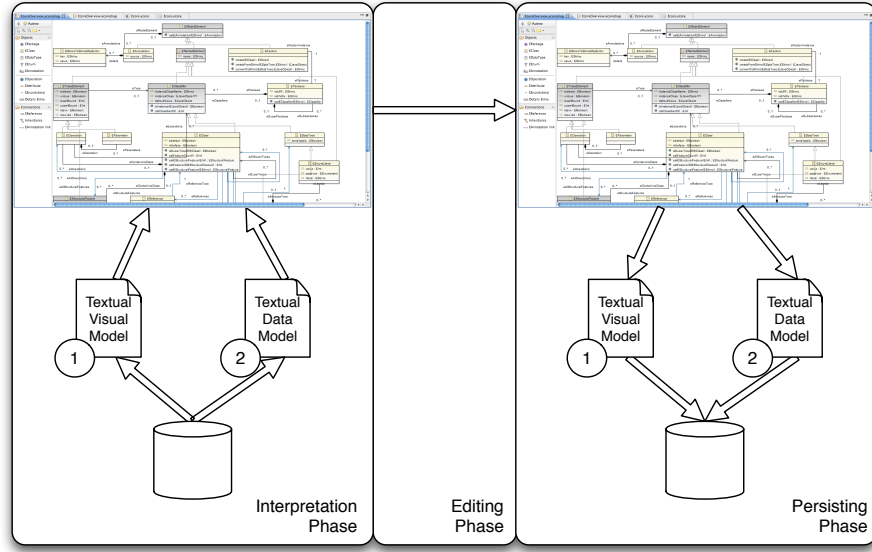


Fig. 2. Use of visual models in Eclipse

not provide any context anchor that would capture information about the scope: which model element is in scope and what is the serialization syntax generated for it. As a result, first it is challenging to understand the relations of a particular model element and its persistent representation, and, second, no explicit traces between a graphical model element and its serialization syntax are kept.

Since the standard persistence mechanism obscures the traces, and since we aim at a reusable and non-invasive tracing toolkit, we settle on using aspects to observe the standard persistence mechanism, recording the context elements and linking them to generated syntax. Thereby, we can trace each model element to its textual representation and thus, establish an explicit mapping between them. This explicit mapping is then exposed to the development environment, and it can be used for development of further tools.

The aspect observes the top-most model traversing method and the subsequently called corresponding methods in `org.eclipse.emf.ecore.xmi.impl`, it observes the sequence of model elements that get treated in the control-flow of those methods, and keeps track of start and stop positions in the generated stream of text in serialization syntax for a model element. Subsequently, it maps model elements to indices in the generated serialization stream.

Figure 3(b) visualizes the standard persistence mechanism saving visual and data models, being observed by Tengja. Thereby, a trace model in memory is generated on-the-fly that links the elements of the two syntax kinds.

In order to demonstrate the capabilities of Tengja, we have implemented a simple extension for Eclipse, able to navigate across the harvested traceability

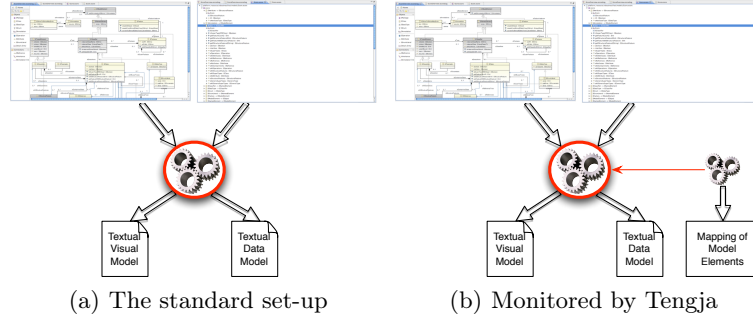


Fig. 3. Eclipse's persistence mechanism serializing two interrelated graphical models

Type of Model Elements	Data Models in .ecore files [correctly traced elements / all elements]			
	Ecore	XMLNamespace	XMLType	fmp
EClass	20/20	1/1	4/4	8/8
EGenericType	10/155	0/7	0/16	0/30
EAttribute	33/33	5/5	11/11	10/10
EEnum	n/a	1/1	n/a	2/2
EReference	48/48	2/2	4/4	16/16
EEnumLiteral	n/a	2/2	n/a	10/10
EParameter	22/22	n/a	n/a	n/a
EOperation	31/31	n/a	n/a	n/a
EStringToStringMapEntry	55/55	29/29	178/178	n/a
EDataType	32/32	3/3	58/58	n/a
ETypeParameter	5/5	n/a	n/a	n/a
EAnnotation	39/39	13/13	81/81	n/a

Table 1. Robustness test results for the data models

links. It allows to mark arbitrary model elements in Ecore-based models and to navigate from the respective element to all related other model elements and textual representations in abstract syntax, visual concrete syntax, and serialization syntax. Assume, as seen in Figure 1, that one opens a GMF model in a diagram editor and one is interested in the traces of a certain model element, in the example **EAttribute**, to other model elements. With Tengja it requires just a button click to move from a marked element to the persistent models opened in text editors, with the highlighted text corresponding to the original model element. Note that this functionality is instantly available for all DSLs defined with Ecore, and all GMF and EMF generated DSL editors.

4 Experimental Evaluation

We evaluate robustness of Tengja with a semi-automated test. A test program loads a number of models and requests traceability links for their elements. The results are stored in a log, which is verified by a human expert.

We used a number of GMF diagram models and a number of EMF data models to run the evaluation. The models were chosen to represent differ-

Type of Model Elements	Diagram Models in .ecorediag files [correctly traced elements / all elements]				
	EcoreAnnotations	EcoreDataTypes	EcoreGenerics	EcoreHierarchy	EcoreOverview
Node	18/18	96/96	73/73	232/232	184/184
DrawerStyle	6/6	n/a	14/14	38/38	34/34
SortingStyle	6/6	n/a	14/14	38/38	34/34
FilteringStyle	6/6	n/a	14/14	38/38	34/34
ShapeStyle	3/3	32/32	7/7	19/19	17/17
Bounds	3/3	32/32	7/7	19/19	17/17
DiagramStyle	1/1	1/1	1/1	1/1	1/1
Edge	3/3	n/a	18/18	65/65	48/48
Location	6/6	n/a	45/45	156/156	116/116
ConnectorStyle	3/3	n/a	18/18	65/65	48/48
FontStyle	3/3	n/a	18/18	65/65	48/48
RelativeEndpoints	3/3	n/a	18/18	65/65	48/48
IdentityAnchor	n/a	n/a	n/a	114/114	90/90
Ratio	n/a	n/a	n/a	n/a	2/2

Table 2. Robustness test results for the diagram information models

ent sizes and use in real projects. In Tables 1 and 2 column headers correspond to individual models. `Ecore.ecore`, `EcoreAnnotations.ecorediag`, `EcoreDataTypes.ecorediag`, `EcoreGenerics.ecorediag`, `EcoreHierarchy.ecorediag`, `EcoreOverview.ecorediag`, `XMLNamespace.ecore`, and `XMLType.ecore` are part of the Ecore implementation available in the `org.eclipse.emf.ecore` Eclipse plug-in (we used version 2.5.0.v200906151043). The model `fmp.ecore` is the metamodel of feature models as used in the Feature Modeling Plug-in [5], available online.

The robustness evaluation was conducted by running a test program on the above models. The program automatically opens the models in their corresponding editors and generates the mapping between model elements and their textual representations by saving the respective model before opening it. Subsequently, the test iterates over all elements of each model and stores their object identifiers with the textual representation to a log file. The generated log files were manually compared against the source files of the corresponding model. The results of this check, sorted by the type of model elements, can be found in Table 1 and in Table 2. In each table cell x/y means that x model elements out of y present in the model were correctly traced, whereas “n/a” means that the corresponding model does not contain a model element of the given type.

The result of the evaluation was positive, showing 93% recall and 100% precision. This means that 93% of the elements have been traced, and all of them traced correctly.

The only exception are model elements of type `EGenericType` (Table 1). Tengja is able to map graphical model objects to their serialization syntax only if the textual representation is a complete XML element: a string of either “`<.../>`” or “`<identifier>...</identifier>`” form. Most often, model elements of type `EGenericType` are in neither form, instead they are subparts of an XML element. We do not consider this a serious problem, since elements of this type are not

‘clickable’ in model editors anyway. If we only consider clickable elements the recall raises to 100% with the same precision.

Threats to Validity. There are two main threats to validity of this experiment. First, our assessment of logs of traces on evaluation models have been performed by the implementer of Tengja, which could introduce a bias. Moreover the evaluation targets were all simple class-diagram files, not actual domain specific models. We intend to expand the set of evaluation models, and improve the independence of the evaluation in our ongoing work on this project.

5 Discussion

Tengja is highly reusable, in that it minimizes the amount of code that need to be refactored when adapting it to a new modeling framework—only the pointcut specifications should be refactored. In this sense Tengja is ready to support modeling frameworks of Eclipse, that do not exist as of today. Tengja is non-invasive meaning that it does not require any modifications to models, metamodels, or existing editors. We have used AspectJ to reach these objectives, but we do believe that using other composition mechanisms, such as Object Teams, would yield similar results.

Alternatively we could have implemented an invasive solution, which completely replaces the persistence mechanism for one, which generates the trace model on the fly. However this requires deep changes in Eclipse’s implementation and would not be reusable across new modeling frameworks.

In model transformation systems with QVT-like architecture the trace models are automatically created while transformations are applied. Implementing the saving mechanism in such a system, would gain traceability for free, albeit still just for one particular framework at a time. Presently QVT itself does not standardize transformations involving serialization syntax, but we would need a language that supports both automatic element-to-element traceability storage and model-to-text transformations. One way to obtain this would be to weave an aspect similar to Tengja into an implementation of existing transformation toolkit.

One could disregard working with serialization syntax at all. Indeed much of the functionality can be achieved at the model level, using soft references and alike. However we have to recognize that a file is the single most popular unit of organizing software development artifacts. Interfaces for software development artifacts tend to be the simplest to implement based on files and they allow programmers to access them with regular text editors (which is very popular).

Since Tengja extends the standard persistence mechanism and supports any Ecore-based DSLs, it is directly interoperable not only with editors, but also with other modeling technologies of Eclipse, including transformation languages like XTend. For example, it can recover traces from abstract to serialization syntax for models resulting out of transformations. If the transformation framework supports materialization of traces, these could be combined with our traces, in order to provide complete end-to-end traceability for chains of transformations.

Tengja assumes a rather close relation between the abstract and serialization syntax. We believe that this is not a serious limitation—it has been expressive enough to succeed on a handful of DSLs of the Eclipse project, which we have used for initial evaluation. Indeed, we hypothesize that, unlike for visual syntax [14], the relation between the abstract syntax and concrete serialization syntax tends to be close for most modeling languages, as it is also known in compilers and interpreters for programming languages. Normally, there exists a mapping between the abstract and serialized representations, where elements of the abstract syntax map to convex fragments of serialization syntax. This assumption significantly simplifies our implementation.

6 Related Work

Traceability between abstract and serialization syntax is hardly discussed in modeling community. Oldevik and Neple [13] discuss it in the context of OMG’s model-to-text transformation standard. They propose to automatically generate trace models to textual models while the transformation is applied, essentially in the same manner as it is done for model-to-model transformations. Our work recovers these links automatically without directly modifying the persistence code, which would otherwise be required to realize their vision. In this sense, Tengja is a kind of model transformation [8, 12], which recovers a trace model by instrumenting a model-to-text transformation.

The literature is abundant in meta-models of trace languages, and in frameworks allowing defining, maintaining and querying tracability information. Much less attention is devoted to automatic recovery of such links. One example is the work of Antkiewicz and coauthors [3] on recovery of framework specific models from Java code. They use static analysis to recover models from Java source code (arguably a textual representation). Once the models are extracted, trace models are used to maintain synchronization links [4].

Ráth et al. [14] extend trace models to arbitrary relations between the abstract and the visual syntax. They use incremental model transformations to maintain the two layers in sync. While such a generality of constraints appears superfluous between the serialization and abstract syntax, it would still be interesting to see whether their technology could keep textual and visual models synchronized easily.

In [11] a rule based approach is proposed for automatic updating of traceability rules based on common model editing operations. It would be interesting to see whether this adapts also to automatic model transformations, and to links between serialization and abstract syntax, in particular in scenarios when the concrete representation is changed concurrently to the abstract representation.

7 Conclusion and Future Work

We have presented Tengja, a robust, non-invasive and reusable aspect-oriented solution for automatic harvesting of traceability links between abstract and se-

rialization syntaxes in modeling frameworks of Eclipse. We are not aware of any tool with similar objectives being available so far.

Tengja visualizes traces between elements of interrelated models on multiple levels of abstraction. Thereby, it may enhance the awareness and the understanding of traces. To demonstrate this we have developed a simple extension for navigation between model elements in various representations, which can be used to debug models and editors.

In future we will continue developing Tengja. For example, the tool presently does not handle models with multiple root elements. This is going to be addressed. We intend to use Tengja to develop higher level tools, for example supporting compositional reasoning about systems in the style of component algebras. Tengja will be the main linking mechanism of this prospective tool.

References

1. OMG Unified Modeling Language, Infrastructure. omg.org/spec/UML/2.2/Infrastructure/PDF, Mar. 2010.
2. Package `org.eclipse.emf.ecore`. download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org.eclipse.emf.ecore/package-summary.html#details, Mar. 2010.
3. M. Antkiewicz, T. T. Bartolomei, and K. Czarnecki. Fast extraction of high-quality framework-specific models from application code. *Autom. Softw. Eng.*, 16(1):101–144, 2009.
4. M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS'06*, volume 4199 of *LNCS*. Springer, 2006.
5. M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature Modeling Plug-In for Eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. ACM, 2004.
6. K. Balasubramanian, A. S. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. *IEEE Computer*, 39(2):33–40, 2006.
7. S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-specific development with Visual Studio DSL tools*. Addison-Wesley, 2007.
8. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
9. R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language Toolkit*. Addison-Wesley, 2009.
10. S. Kelly and J.-P. Tolvanen. *Domain-specific modeling: enabling full code generation*. Wiley-IEEE, 2008.
11. P. Mäder, O. Gotel, and I. Philippow. Semi-automated traceability maintenance: An architectural overview of traceMAINTAINER. In *Proceedings of ECMDA Traceability Workshop ECMDA Traceability Workshop (ECMDA-TW)*, 2009.
12. T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
13. J. Oldevik and T. Neple. Traceability in model to text transformations. In *Proceedings of ECMDA Traceability Workshop ECMDA Traceability Workshop (ECMDA-TW)*, 2006.

14. I. Ráth, A. Ökrös, and D. Varró. Synchronization of abstract and concrete syntax in domain-specific modeling languages. *Software and Systems Modeling*, 2009. Available online first.
15. T. M. H. Reenskaug. Models - Views - Controllers. heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf, 1979.
16. T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

9

Taming the Confusion of Languages – ECMFA'11 (Paper B)

Taming the Confusion of Languages

Rolf-Helge Pfeiffer and Andrzej Wąsowski

IT University of Copenhagen, Denmark
{ropf, wasowski}@itu.dk

Abstract. Large software systems are composed of diverse artifacts. The relations between these artifacts are usually not formalized, if the artifacts use different modeling or programming languages. This hinders component-oriented development, as interfaces of exchangeable components do not capture hidden artifact dependencies. We present GenDeMoG, a tool that allows for mining inter-component dependencies beyond those explicitly specified. GenDeMoG is a generic generator-generator parameterized with a high-level system model containing dependency specifications. So, unlike the language interface mechanisms, GenDeMoG is not restricted to any given kind of links. We apply GenDeMoG to a realistic case study—an open source enterprise system, OFBiz. The experiment confirms that the stereotypical opinion about unknown dependencies across artifact types is indeed correct. Just 22 specifications allowed GenDeMoG to uncover 1737 undocumented inter-component dependencies among OFBiz components.

1 Introduction

A modern enterprise system is heterogeneous—it combines development artifacts, expressed in various languages. These artifacts are *aggregated* into larger reusable entities, called components. However, when forming a system, the artifacts are not merely put together to form components, but they are interrelated via references or other dependencies. Depending on the language and the provided mechanisms, such references are either *direct* or *indirect*. Direct references are string-based references expressed by using the same datum in different locations. If at one place the datum changes, the reference is broken. *Indirect* references are established between artifacts at runtime; for example in adapter calls, such like when a Java method is calling a Prolog rule.

The larger the number of languages used at development time, the more artifacts containing references to artifacts in other languages appear. Further, since not all languages are general purpose (GPLs), many artifacts cannot use adapters to interact with code in other languages, instead they refer to other code artifacts *directly*. Usually such direct references are *implicit*, in the sense that their semantics is hidden in the execution platform (an interpreter, business logics, etc.). In contrast, *explicit* references exploit meta-data of the referenced datum, for example document structure like in unified resource locators (URI). Either type of references result in dependencies between artifacts.

The proliferation of references across languages and across components causes a number of problems for software developers:

- Since inter-language dependencies are usually implicit, they require substantial domain knowledge for a developer to correctly perform simple evolution steps.

- Implicit dependencies may cross the borders of system’s aggregation structure: components get coupled tighter together. When this happens, it is often not explicitly recorded in component interfaces.
- Errors caused by broken dependencies are most often only exposed at runtime; so detection of any errors requires thorough testing of the modified code—while, at least in principle, errors caused by dangling references between static artifacts, could easily be caught at compile time.

The first objective of this paper is to present GenDeMoG—a tool that allows for specifying inter-component dependency patterns for artifacts in heterogeneous systems. GenDeMoG automatically reveals the hidden dependencies using these patterns. The tool is generic. It is neither tied to certain languages nor applications. It is also non-invasive, in the sense, that it does not require that the related artifacts are modified.

Our second objective is to use GenDeMoG to analyze a larger realistic case study for the presence of unspecified dependencies, and their interaction with component structure of the system. We use *OFBiz* [10], an open-source enterprise automation software project, as the subject in this study. *OFBiz* is a component-based system comprising a multitude of heterogeneous languages, both general purpose languages (GPLs) like *Java*, and domain-specific languages (DSLs). We research *OFBiz* applications, that is programs running on the *OFBiz* framework. Such applications are again component-based and heterogeneous. Therefore, we identified 22 exemplary dependency patterns using 7 languages specific to *OFBiz*. GenDeMoG automatically revealed 1737 inter-component dependencies of the kind specified by these patterns.

The main findings of these study are that:

- There indeed exists a large number of references between *OFBiz* components not specified in the component description mechanism (see e.g., Fig. 1)—even though the mechanism provides for specifying such.
- These dense and circular references couple the components tightly together. This confirms the qualitative understanding (see Section 2) that evolution or refactoring of *OFBiz* component structure is difficult in practice.

A useful by-product of this work is an initial meta-model of Java 5 implemented in *Xtext* [14]. Since we could not find a pre-existing *Xtext* specification of Java, we adapted an existing ANTLR grammar, and optimized it to decrease the number of model elements created in each type, so that pattern matching, which relies on these types, is more efficient. The new Java model is available online together with GenDeMoG (<http://www.itu.dk/~ropf/download/gendemog.html>).

We further motivate the problem of unspecified references across languages and components using the *OFBiz* example in Sect. 2. In Sect. 3 GenDeMoG is introduced. Sect. 4 describes the experimental case study of applying GenDeMoG to *OFBiz*. We end with a discussion of future work (Sect. 5), related work (Sect. 6) and conclusion.

2 Background and Rationale

Software systems are implemented using many interrelated artifacts, expressed in multiple languages. We shall now investigate this architectural phenomenon by surveying

```

<ofbiz-component name="order"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ofbiz.apache.org/dtds/ofbiz-component.xsd">
  <resource-loader name="main" type="component"/>
  <classpath type="jar" location="build/lib/*"/>
  <classpath type="dir" location="config"/>
  ...
  <entity-resource type="model" ... location="entitydef/entitymodel.xml"/>
  ...
  <entity-resource type="data" ... location="data/OrderTypeData.xml"/>
  ...
  <service-resource type="model" loader="main" location="servicedef/services.xml"/>
  ...
  <service-resource type="eca" loader="main" location="servicedef/secas.xml"/>
  ...
  <webapp name="order" title="Order" ... location="webapp/ordermgr"
    base-permission="OFBTOOLS,ORDERMGR" mount-point="/ordermgr"/>
</ofbiz-component>

```

Fig. 1. An excerpt of the Order component descriptor.

```

<entity entity-name="FinAccount" package-name="org.ofbiz.accounting.finaccount"
  title="Financial Account Entity">
  <field name="finAccountId" type="id-ne"></field>
  <field name="finAccountTypeId" type="id"></field>
  <field name="statusId" type="id"></field>
  ...
</entity>

public static boolean validatePin(GenericDelegator delegator,
                                String finAccountId, String pinNumber) {
    GenericValue finAccount = null;
    try {
        finAccount = delegator.findByPrimaryKey("FinAccount",
            UtilMisc.toMap("finAccountId", finAccountId));
    } ...
}

```

Fig. 2. An inter-language cross-component string-based dependency: the marked string literals in the Java method validatePin refer to objects specified in the XML file on top (both red).

the example of the OFBiz project. OFBiz is a component-based *framework* on top of which OFBiz *applications* are run. A standard OFBiz distribution includes 11 application core components delivering key functionalities: accounting, commonext, content, humanres, manufacturing, marketing, order, party, product, securityext, and workeffort.

Each application component contains a *component descriptor*, by convention in a file named ofbiz-component.xml, expressed in a domain specific language using an XML syntax. The descriptor defines the visibility of artifacts, their types, etc. Fig. 1 presents a fragment of the descriptor for the order component, which supports functionality around management of customer orders. The component descriptor informs the framework about the existence and location of data models and initialization data (entity-resource) or about the existence and location of business logic (service-resource). Further it declares whether a component is a web application (webapp) and allows for the specification of a set of components that it depends on (depends-on). The latter, however, is not used in the above example, nor anywhere else in core application components of OFBiz.

Implementation artifacts are expressed using DSLs and GPLs. Dependencies between heterogeneous artifacts are expressed using string-based references, since the languages themselves do not support heterogeneous interface descriptions—i.e., only few languages

allow for expressing relations between language elements within the same language in a managed manner. That is, the wish to separate concerns by expressing development artifacts in different DSLs and GPLs and the lack of a uniform managed mechanism to describe relations between artifacts creates a confusion of languages.

Fig. 2 shows an example of an inter-language, inter-component, string-based dependency between a Java method call and an entity definition. A dependency (reference) goes from a class `FinAccountHelper` in the order component to the file `entitymodel.xml` in the accounting component (both red in the Figure). This reference, as many other similar, is not captured by the interface specification of Fig. 1.

Dependencies that cross component boundaries are problematic since they increase coupling and make components hard to exchange or remove, when customizing OFBiz. Still, OFBiz contains no mechanism for specification of inter-component dependencies, which could help developers by reporting violations of dependencies, or merely by documenting dependencies. The following quotes (original spelling) from the developer mailing lists show that indeed implicit coupling of components is an issue in practice (Note, these quotes are to motivate our work. We did not perform a systematic research on the OFBiz mailing lists):

Hansen: *I am looking for information regarding the inter-dependency among all ofbiz components ... Is there any effective way to know this kind of information. So that I can safely remove those components I do not want without affecting the functionalities of the other components that I want to keep*

skip@theDevers *I recently used just the party manager in a project and deleted all the rest. Took a couple of days replacing/commenting out the dependencies. ...*

Hansen *It would be useful to have such information (something like rpm dependency list) handy especially for those application components as they are supposed to be less dependent on other components comparing to framework components.*

[<http://ofbiz.135035.n4.nabble.com/component-dependency-td153157.html>]

Mustansar Mehmood *My company(a service company) is considering ofbiz as their ERP/CRM/accounts Receivable system but Ofbiz seems to have a few things that my company will not be using for instance E commerce or Manufacturing/. How do we removed those applications ? and keep it running stable. ...*

Divesh Dutta *You can not remove the any of the complete application. Each module is related to another module one or other way. ...*

cjhorton *Generally, because of dependancies you don't want to 'remove' a component, but rather just 'hide' it as talked about in the links above. ...*

[<http://ofbiz.135035.n4.nabble.com/Removing-applications-from-OfBiz-td160666.html#a160669>]

Our long term research agenda is to support smooth evolutions of systems with interrelated artifacts. We recognize that this cannot be solved without taking project specifics into account. Thus in this paper we present a *generic tool*, which parameterized with a *project-specific model* of a software system with its dependencies, is able to reveal unspecified relations between implementation artifacts across components. In the second part of the paper we will use this tool to further explore the nature of dependencies in an example of a mature business application, such as OFBiz.

3 GenDeMoG

This section introduces GenDeMoG a generic tool for mining an inter-component dependency graph of heterogeneous component-based software systems. GenDeMoG's architecture and other important aspects are described.

The central artifact to GenDeMoG is a *Component Descriptor Model (CDM)*. A CDM provides an external description of a component-based software system under analysis. It states which languages are used by development artifacts, what components are formed by which artifacts and, most importantly, includes the language-level patterns that describe the conditions for dependencies between development artifacts. A CDM comprises the following three sections:

Type definitions In our work we use the terms *types* and *languages* synonymously.

Types are defined by a path to a languages meta-model assigned to an alias. The alias is required to be equal to the meta-models name. GenDeMoG generally provides 2 different means of importing a language to the framework. These are first XSD meta-models and second EMF meta-models. Each language definition has to refer to its meta-meta-model.

Component List Components are specified by a unique name. This may be a relative path to a folder containing the components artifacts or any other name. Each component declares a list of artifacts it contains. Artifacts are specified by a unique path, with respect to a single CDM. Each artifact refers further to its meta-model or the language it is instance of.

Pattern Definitions Dependency patterns are defined by specifying a key pattern, and a reference pattern with respect to the corresponding language definitions. Patterns are specified using the EMF expression language that is used for writing model transformers and code generators in Xtend and Xpand [13, 12] respectively. Key patterns always start with the keyword `possibleKey` and reference patterns with the keyword `possibleReference` and may be followed by navigations to certain model sub-elements, see the dependency pattern in Fig. 3. The second element of both pattern specifications is the respective model element type. This states the type of the `possibleKey` and `possibleReference` model element respectively. Finally, the language for key and reference patterns is specified. A separate boolean expression over key- and reference patterns defines under which condition a key and a reference pattern are in relation.

Obviously, GenDeMoG's CDM is directly tied to the meta-model hierarchy [28, 16]. All used languages in a software system are defined to be instances of the same meta-meta-language (M3) (Ecore [11]). This allows for the definition of dependency patterns on meta-model level (M2) or language level. All artifacts are specified as models (on level M1) that are instances of the corresponding meta-model or languages stated in the type definitions.

In a nutshell, a CDM lists all languages used or of interest in the software system under analysis, it lists which artifacts form components and most importantly, which language constructs induce inter-component dependencies.


```

key pattern: "possibleKey.entityName" ofType "EntityType" in iof entitymodel
reference pattern: "possibleReference.relEntityName"
                  ofType "RelationType" in iof entitymodel
dependency relation: "_keyPattern_ == _refPattern_"

```

Fig. 3. Example pattern.

```

cached Boolean check(EntityType possibleKey, RelationType possibleReference) :
    possibleKey.entityName == possibleReference.relEntityName;

```

Fig. 4. Automatically generated code corresponding to the pattern in Fig. 3.

Supported Languages. Since GenDeMoG allows for mining inter-component dependencies for heterogeneous languages, it is crucial to provide a mechanism for including new languages. As mentioned above all languages are based on the Ecore meta-meta-model. XSD-based language definitions can either be automatically or manually converted to Ecore-based language definitions. In the experiment (Sect. 4) we use 6 XSD-based language definitions and one Ecore-based definition. The Ecore-based Java 5 model is an initial implementation setting up on XText. Generally, there are multiple sources for predefined languages such as e.g., EMFText Syntax Zoo [3] (currently, 88 languages), the Atlant Ecore Zoo [2] (currently, 304 languages), the MoDisco Project Page [9] (currently for Java and XML documents).

Dependency Patterns. GenDeMoG mines *Key-Reference* dependencies. That is, there is a key, the definition of a certain piece of information that may be referenced. Furthermore, there are references, which are pieces of information that specify the referenced keys. In GenDeMoG such pieces of information are model elements. A key and a reference are in relation, if and only if the boolean constraint that specifies their relation evaluates to true. That is, there are no keys and references without a relation.

Dependency patterns are defined between two language elements of a certain type (type) each which in turn belongs to a certain language (in iof). Each key and reference pattern may be defined more precisely by an optional `refine` statement. It may be used to provide another boolean expression that needs to evaluate to true. Key patterns are referenced in the dependency patterns using the key word `_keyPattern_`. Reference patterns are referenced in the dependency patterns using the key word `_refPattern_`. A dependency pattern, with substituted `_keyPattern_` and `_refPattern_`, is a boolean expression that evaluates to true if a dependency between a key and a reference exists. Since GenDeMoG is a generator-generator, the dependency patterns get transformed to dependency graph generator code that contains a boolean function call for the dependency relation, see Fig. 4, corresponding to the dependency relation in Fig. 3. The language for describing key and reference patterns is the *Eclipse Modeling Framework (EMF)* expression language [4].

GenDeMoG's Architecture. GenDeMoG is a generator-generator. “[It] compiles a query into a special-purpose search program, whose task is only to answer the given query. ... the input to the program generator is a general query answerer, and the output is a ‘compiler’ from queries into search programs.” [25] The queries in our case are the dependency patterns. The ‘compiler’ is the *ext_gen* model transformer. It takes the dependency pattern queries from a CDM and generates *graph_gen* as special purpose

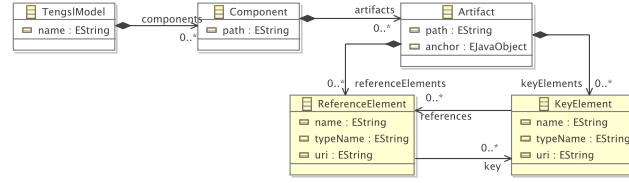


Fig. 5. Metamodel for the mined inter-component dependency graph.

search program, which in turn is a model transformer again. The following is a description of GenDeMoG’s architecture using the compiler notation from [25]:

$$graph_gen = \llbracket ext_gen \rrbracket_{x\text{pand}} [cdm, fst_wf] \quad (1)$$

$$snd_wf = \llbracket wf_gen \rrbracket_{x\text{pand}} [cdm] \quad (2)$$

$$dep_model = \llbracket graph_gen \rrbracket_{xtend} [\{artifacts\}, snd_wf] \quad (3)$$

Entities in $\llbracket \cdot \rrbracket$ describe semantic units, i.e., programs in a certain language (denoted as subscript) that are executed consuming the arguments in $[\cdot]$. Workflows (fst_wf , snd_wf) are programs that execute model generators and model transformers in a specified order. The first workflow calls the ext_gen code generator on the CDM. That means, that the generator-generator gets instantiated to the concrete generators for patterns. The second workflow takes the generated concrete generators and runs them on the corresponding artifacts in order to generate the dependency graph. Thus, workflows are programs that have a coordinating function and we denote them as arguments to model generators and transformers. The generated dependency model is an instance of the meta-model shown in Fig. 5. Such a dependency graph ($Tengsl^1$ model) contains a representation for each component, containing artifacts, which in turn may contain key elements and references to keys.

4 Experiment

We will now demonstrate GenDeMoG by applying it to a real-world medium-sized software system analysis scenario. This way we will extract an inter-component dependency graph for the subject system, which we will further analyze.

The Experiment Subject. OFBiz is an industrial-strength open source enterprise automation software project licensed under the Apache License Version 2.0. By open source enterprise automation we mean: Open Source ERP, Open Source CRM, Open Source E-Business / E-Commerce, Open Source SCM, Open Source MRP, Open Source CMM-S/EAM, and so on [10]. In this experiment we have used OFBiz ver. 9.04, available at <http://svn.apache.org/repos/asf/ofbiz/branches/release09.04>.

OFBiz’ source tree contains 6522 files, including 1122 Java source code files, 365 Groovy files, 1283 XML files, and files in various other languages. Out of 6522 files,

¹ Tengsl: Icelandic for relationship

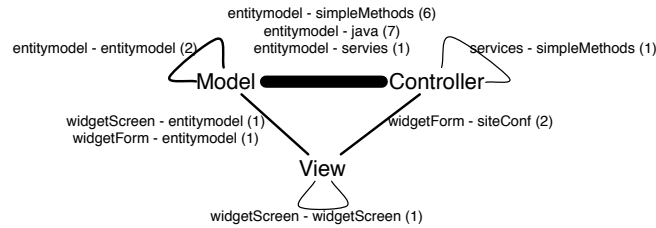


Fig. 6. Amount of dependency patterns between the parts of the MVC architecture.

2289 are related to the framework only, and 1539 are owned by the 11 core application components. The tree contains more than 388000 lines in XML models, and in excess of 310 KLOC of Java code. The OFBiz project uses not less than 33 domain specific languages and relies on about a dozen standards or open technologies. More than 2000 email addresses are subscribed to the OFBiz mailing list, most of these technically oriented participants ranging from core developers to technically skilled users who customize and deploy OFBiz. The project website lists (February 2011) several large customers, including internationally known brands such as telecom operators or large airlines and a multitude of smaller ones. All in all, OFBiz is a fairly representative example of an industrial quality modern enterprise system.

In this paper we focus on dependencies in OFBiz applications across the core components, i.e., inter-component dependencies. To customize an application the mentioned components can either be modified, removed, or extended and expanded by new components. We note that in case of OFBiz, the framework follows the same architectural principle as the applications built on top of it.

A typical application relies on the following 25 DSLs: componentLoader, jndiConfig, ofbizComponent, ofbizContainer, ofbizProperties, datafiles, entityConfig, entityEca, entitygroup, entitymodel, fieldtypemodel, simpleMethods, securityConfig, serviceConfig, serviceEca, serviceGroup, serviceMca, services, testSuite, regions, siteConf, widgetForm, widgetMenu, widgetScreen, and widgetTree. On top of the GPLs such like Java and Groovy [6], the FreeMarker Template Language (FTL) [5], and JavaServer Pages (JSP) [7] are used. DSLs are used to describe data models, visual application parts, and together with GPLs they are used to specify the controller level, i.e., required logics, following the model-view-controller design pattern.

The Experiment Set-up. The objective of this experiment is two-fold. First, we want to show that a large number of dependencies—precisely, direct references—exists between components and languages in a mature system like OFBiz. Second, we want to demonstrate effectiveness of GenDeMoG while revealing the dependencies. We do not aim at revealing all not specified dependencies—since this would require a substantial domain knowledge.

We are interested in dependencies within and across architectural layers. Since OFBiz follows the model-view-controller design pattern the interesting component boundaries can be divided in the following 6 categories: *model-model*, *model-view*, *view-view*, *view-controller*, *controller-controller*, and *controller-model*. We have identified patterns describing dependencies in each of these. The dependencies have been identified by

studying available documentation, in particular the developer guide book [24]. We have stopped searching for more dependencies as soon as we accumulated representatives for all categories (in total 22 dependency patterns, 19 inter-language and 3 intra-language patterns). Fig. 6 shows these categories by black edges—thickness of the edge, and the numeric labels, reflect how many dependency patterns we were able to find on each of the boundaries. Note that due to the inherent incompleteness of the collection method this does not mean that proportions between dependencies in OFBiz are precisely as indicated in the figure. The diagram merely shows the characteristics of our selected sample set, and it does show *qualitatively* that there exist dependencies on each of these boundaries.

The dependency mining algorithm is applied to 642 artifacts (relevant for the patterns) belonging to the 11 core application components of OFBiz.

Fig. 7 shows an excerpt of the CDM used for this experiment. The complete model showing all included languages, components, artifacts and dependency patterns is available online at <http://www.itu.dk/~ropf/download/gendemog.html>.

Let us survey Fig. 7 to provide further details. A CDM is defined per software system or project where the projects root folder is specified (`at`), see line 1. Used languages (`type`) are declared, see lines 3-6, by giving them a unique name, a path to their language specification (`metamodel`) or additionally their language package (see line 6), and stating the languages meta-meta-model (`typeOf`). Lines 8-24 show an excerpt of two component specifications (`component`) via a unique name, a path to the component's root folder (`at`), and finally the contained artifacts. Each artifact (`artefact`) is specified by a path to the artifacts location and the type of the language it is instance of (`typeOf`). Note, to minimize GenDeMoG's memory footprint, the experiment is run with preprocessed Java source code files. That is, a separate preprocessing step converted the textual Java files to an Ecore-based model representation (`jxmi`, e.g., line 13). Lines 26-42 show the declaration of two dependency patterns. We will have a closer look to the second one. Model elements that might be keys (`key pattern:`) are specified by a pattern to a model element, the model element's type (`typeOf`), and the type declaring language (`in io of`). The same holds for the specification of a reference element. In both are `possibleKey` and `possibleReference` reserved keywords. In case that a lot of model elements match the pattern definition with the corresponding type, it is possible to further refine the matching model element set (`refine`). By a boolean expression over key and reference patterns, are dependency relations defined (`dependency relation:`). In dependency relations and refine declarations are `possibleKey`, `possibleReference`, `_keyPattern_`, and `_refPattern_` reserved keywords.

Results and Analysis. A check for one pattern (out of the 22) over the 642 relevant artifacts takes about 2-3 minutes time on a 2GHz Intel Core 2 Duo Mac Book with 2GB 1067MHz DDR3 RAM.

Our 22 patterns, reveal in total 1737 inter-component dependencies (the number of elements of type `ReferenceElement` as per Fig. 5). These 1737 dependencies have 635 unique target elements (`KeyElements`). Remember, that since the list of patterns is highly incomplete, this number is a strict lower bound on the actual number of interactions between core application components in OFBiz!

```

1 project ofbiz at "/ofbiz_9_4"
2 ...
3 type entitymodel metamodel ".../dtd/entitymodel.xsd" typeOf XSD,
4 type simpleMethods metamodel ".../dtd/simple-methods.xsd" typeOf XSD,
5 type java3 metamodel ".../lang/xtext/Java3.ecore" typeOf Ecore
6   -P "dk.itu.sdg.lang.xtext.java3.Java3Package"
7
8 component accounting at "/ofbiz/applications/accounting" {
9   artefact ".../accounting/.../entitymodel.xml" typeOf entitymodel,
10  ...
11   artefact ".../accounting/.../TaxAuthorityServices.xml" typeOf simpleMethods,
12  ...
13   artefact ".../accounting/.../UtilAccounting.jaxmi" typeOf java3,
14  ...
15 }
16 ...
17 component order at "/ofbiz/applications/order" {
18   artefact ".../order/.../entitymodel.xml" typeOf entitymodel,
19   artefact ".../order/.../entitymodel_view.xml" typeOf entitymodel,
20  ...
21   artefact ".../order/.../PurchaseOrderTest.jaxmi" typeOf java3,
22   artefact ".../order/.../SalesOrderTest.jaxmi" typeOf java3,
23  ...
24 }
25 ...
26 /* entitymodel <-> entitymodel */
27 key pattern: "possibleKey.field.name" typeOf "EntityType" in ioof entitymodel
28 reference pattern: "possibleReference.relation" typeOf "EntityType"
29   in ioof entitymodel
30 dependency relation:
31   "_refPattern_.select(e|e.relEntityName == possibleKey.entityName).size > 0
32   && _keyPattern_.intersect(_refPattern_.keyMap.fieldName).size > 0"
33 ...
34 /* entitymodel <-> java */
35 ...
36 key pattern: "possibleKey.entityName" typeOf "EntityType" in ioof entitymodel
37 reference pattern: "possibleReference" typeOf "PrimaryVarCall"
38   refine "_refPattern_.name.name.last() == ('getRelatedOne') &&
39   !_refPattern_.eContents.typeSelect(Arguments).isEmpty &&
40   !_refPattern_.eAllContents.typeSelect(PrimaryLiteral).isEmpty" in ioof java3
41 dependency relation: "_refPattern_.eAllContents.typeSelect(PrimaryLiteral)
42   .select(e|e.literal == '\"'+_keyPattern_+'\"').size > 0" ...

```

Fig. 7. An excerpt of the CDM for the experiment.

It turns out that the density of dependencies varies a lot for patterns. This is visualized in Fig. 8. For example pattern number zero, leftmost in the figure, uncovers 700 dependencies, to 132 elements, while pattern number five applies rarely, uncovering only 4 dependencies to 3 key elements.

Fig. 9 shows qualitatively how dependencies are split across the components. Vertices represent components. There is an edge between two vertices if there is at least one dependency between the components in the given direction (arrow-heads point towards owners of target key elements). The only component that is independent from the remaining ones, as far the 22 dependency patterns are considered, is securityext. Note, that commonext depends on no component but is required by four others.

The graph contains a large number of cycles, and it is clear that each component depends on a handful of others. Also each has a few dependent ones. The median number of components depending on a given node is 6, while the median number of components on which a node depends is 5. This confirms quantitatively that problems indicated in the mailing list discussions (see Sect. 2) are well grounded—clearly it must be a challenge

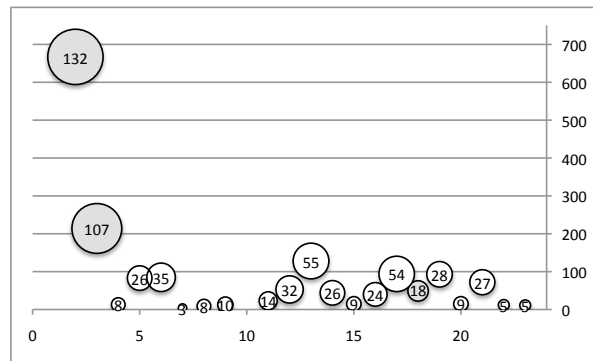


Fig. 8. Number of dependencies (vertical axis) for each pattern (horizontal axis). Number of key elements are shown in the bubbles. Grey bubbles are intra-language patterns.

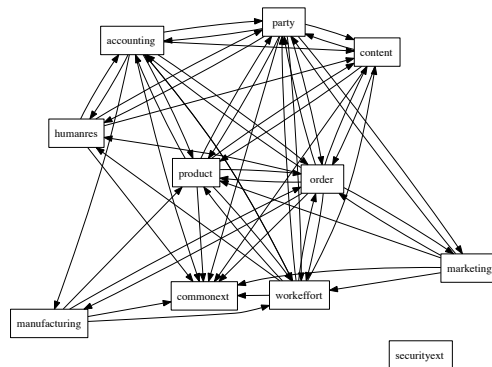


Fig. 9. Inter-component dependencies aggregated from the mined dependency graph.

to remove, replace or modify a component. Indeed, we have been able to confirm this coupling using merely 22 patterns!

Despite the fact that the component descriptions (ofbiz-component.xml) allow to specify component dependencies, none of the uncovered dependencies has been explicitly declared by developers. It turns out that this specification mechanism is not used in practice. This motivates further research in automatic dependency maintenance tools such as GenDeMoG.

Fig. 10 summarizes distribution of keys (referred elements) and references (referring elements) over the languages. Not surprisingly the biggest amount of keys is defined in the entitymodel DSL, which is used to define the data model. The large numbers of references outgoing from entitymodel (881) is caused by a large number of relations between the various concepts in the data model. This is natural in a data processing system. Also data modeling patterns were the easiest to identify for non-experts, so it is expected that we included some of these. As we have seen in Fig. 6—the model part of the architecture (containing the entitymodel language) has participated in the biggest share of our patterns, so it is expected that it generates most dependencies.

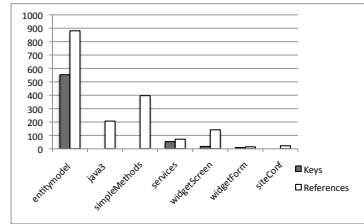


Fig. 10. Distribution of keys and references per language.

Finally, we have classified the dependencies that we have observed, in order to understand the circumstances in which they emerge. We observe three main categories of dependency patterns (and thus of the mined dependencies):

1. A *relation* provides a mechanism for distributed information specification within a language. For example, the relation mechanism of the entitymodel language allows specifying relations between data tables across artifact boundaries.
2. A *call* is an inter-language mechanism, where at least one language has runtime semantics and thus can call certain other code blocks or variables. For example, a Java method call that consumes a name of a data table as a parameter.
3. A *value source* is an inter-language mechanism that provides missing values to a template—for example, when displaying the value of a data field in a view.

Threats to validity The main external threat is that OFBiz is not a representative example of an enterprise systems. We have tried to argue that it has many of typical characteristics of such systems. Moreover, we have informally confirmed that indeed it resembles commercial systems a lot, through personal communication with engineers in the ERP industry. The main reason to use OFBiz in this experiment is the unrestricted access to its source code, which would be difficult for other products (and would also make our results difficult to reproduce or compare in future).

The main internal threat lies in the selection of the 22 dependency patterns. First, that they might be incorrect. Second, that they are not complete. We addressed the latter issue by deciding not to draw any conclusions about completeness. This allowed us to mitigate the former, by only focusing on the most obvious patterns, that can be extracted from the best available documentation [24].

Furthermore, GenDeMoG's algorithm can contain errors. Besides the usual testing effort aiming at establishing trust in the implementation we have manually verified that indeed the dependencies have been correctly matched by manually inspecting all of them for two selected patterns. This step has shown that 100% of dependencies have been mined correctly (no false positives). Since the mining program is automatically generated, our confidence that it operates correctly for other patterns is high.

Finally, the taxonomy of dependency kinds presented above is by design not complete, but only covers the categories that actually appeared in our 22 patterns.

5 Discussion and Future Work

Currently, GenDeMoG allows for the following:

- Non-invasive description of component-based software systems capturing used languages, components and contained artifacts.
- The definition of inter-component dependency patterns on the language level.
- Automatic generation of programs for mining inter-component dependencies.

GenDeMoG itself relies on *Eclipse Modeling Framework* (EMF) technologies, such as the model transformation language Xtend, the code generation language Xpand, and the DSL development framework Xtext.

It is specific to GenDeMoG that new languages may be included as Ecore-based models, i.e., all languages are described using the same meta-meta-language. This allows for the definition of dependency patterns on language level using one language, in this case Xtend. Importantly, this makes GenDeMoG a generic tool, i.e., applicable to a wide range of software systems. To demonstrate GenDeMoG’s general applicability we have created a meta-model for Java 5 and used it in the experiment. Currently, GenDeMoG’s CDM supports language dependency patterns that relate single language elements to each other. In future, we will investigate if relating single language elements is sufficiently expressive for general cases, or if language dependency patterns need to be more expressive.

One could consider relying on another meta-meta-model than Ecore, such as the *Kernel Metameta-Model* (KM3) [27]. Then, GenDeMoG’s model transformations could be implemented using ATL [1], QVT [8], or the VIATRA2 [15] framework. The particular choice of modeling and model transformation technologies is not essential for the design idea behind GenDeMoG. In future, it would be interesting to investigate, whether the technologies selected were the most efficient possible for the use case. Since models become very large for large artifacts, we intend to continue improving the tool’s performance. Also, further work is required to integrate GenDeMoG better into the IDE.

GenDeMoG only mines cross-component dependencies so far. It is however extremely easy to extend it so that intra-component dependencies are also searched—we have made preliminary attempts in that direction. Nevertheless, we have decided to focus on the cross-component dependencies, as they pose a much more pressing issue.

All in all, we spent two man-month for implementing GenDeMoG, setting-up the experiment, and for identifying the dependency patterns. We did not measure more precisely how much time we spent on reading documentation and reading source code. The working time necessary to apply GenDeMoG to other software systems, i.e., to identify the corresponding dependency patterns, depends on the degree of experience a developer has with development of the system under scope.

Since GenDeMoG is generic with respect to the software systems under analysis its applicability is tied to the amount of language definitions readily available. Especially GPL definitions can be reused across projects. Hence we aim for providing definitions for languages such as Groovy, JSP, and FTL.

6 Related Work

The closest work to GenDeMoG is [20], which presents the PAMOMO tool. PAMOMO uses patterns on models for defining traces or constraints between Ecore-based models. PAMOMO only allows for pattern definition, whereas GenDeMoG supports modeling

entire software projects, where the used languages, components, contained artifacts, and the dependency patterns are provided together. On top of that, we provide experimental data: we assess the tools performance applying it to a mature system, and analyzing the data that it can provide. No experimental data on a realistic case study is available in [20]. It would be interesting to investigate how PAMOMO patterns scale compared to dependency patterns in our CDM.

Macromodels [29] have been proposed to capture the meaning of relationships between models. The reference key relationship captured in the dependency graph of GenDeMoG (Fig. 5) could be defined in macromodels as well. Contrary to macromodels, we do not aim at assigning different types to dependencies. We believe, plain reference key relations are the most generic dependencies. Therefore, only those should be supported by a generic tool, such as GenDeMoG.

Similarly to macromodels, GenDeMoG allows to relate models that present different views of the system to each other. We provide evidence for this in Sect. 4. Both macromodels and megamodels [26] provide a framework, which allows for manual specification of different relations. We provide automatic mining facilities to automatically reveal dependencies.

The AM3 tool (<http://eclipse.org/gmt/am3/>, [26]) allows capturing and handling of traceability links between models across languages. A megamodel contains links between interrelated model elements and models. Our dependency model (Fig. 5) resembles megamodels in the sense that it is noninvasive: it captures dependency information without modifying the artifacts in question. Megamodels have been designed with DSLs in mind. GenDeMoG strives to support both DSLs and GPLs, recognizing that a truly heterogeneous system, like OFBiz, contains all kinds of artifacts. Again, we extend [26], by considering a substantial realistic case study. On the other hand it would be interesting, to see whether there are any performance gains in integration of AM3 into GenDeMoG.

Mahé [30] uses megamodels in reverse engineering for an existing software system, *TopCased*. Unlike, GenDeMoG, which is generic, his tool is geared towards specific kind of patterns and platforms. Since GenDeMoG is parameterized with models it allows for a more precise and concrete software system definition.

Favre et. al. describe a scenario of reverse engineering a software system written in C and COBOL [19]. They deploy the Obeo Reverse engineering tool that treats source code as models. They create a single model for the entire software system. This model is subsequently transformed, e.g., to a visual representation of the system. The main commonality between Favre et. al., the *Reuseware* framework [22], the *MoDisco* [17] toolkit, and GenDeMoG is that code artifacts are abstracted to Ecore-based models and further transformations are applied to these models and not directly to the source code. Reuseware and MoDisco both provide a meta-model for the Java GPL. We decided to implement and deploy our own Ecore-based Java model using XText. We did not reuse MoDisco's Java model since we are interested in a model per compilation unit and MoDisco generates one model per project from the abstract syntax tree. Our Java model is grammar based, i.e., a parser is generated out of a grammar that builds a Java model independently from the Java abstract syntax tree. We did not chose EMFText's Java model [21], since it could not be used to read the OFBiz code base.

GenDeMoG and *SmartEMF* [23] both abstract XML-based DSLs to Ecore models. They both use the same case study—the OFBiz project. However, SmartEMF requires using Prolog to express inter-model dependencies, and it does not handle reference to GPLs. Also we did not focus on customizations of OFBiz applications, but more on OFBiz applications seen as component-based systems.

Mélusine [18] is a DSL composition environment, which captures the forward-development of a software system. In contrast, GenDeMoG is concerned with existing, perhaps legacy, heterogeneous software systems, where languages are already inter-related and the hidden dependencies need to be revealed.

7 Conclusion

We have presented GenDeMoG, a generic generator-generator which allows for the non-invasive description of component-based software systems and patterns which describe language structures that result in inter-component dependencies. In particular, GenDeMoG allows for the definition of dependency patterns between artifacts in a heterogeneous system, including dependencies across boundaries between GPLs and DSLs. GenDeMoG is available online, and as a generic tool, it can be readily instantiated for other projects.

Furthermore, we have conducted an experiment, applying GenDeMoG to an industrial strength, mature case study, the OFBiz project. We are not aware of any previously published experiment that characterizes adverse impact of inter-model references on architecture in systems based on multiple modeling languages. The experiment has confirmed the informal impression, that it is difficult to manipulate OFBiz components—this is likely caused by a quite tight and circular coupling between core application components. It has also confirmed that a large number of implicit dependencies exist in the system, even though, the internal component specification mechanism of OFBiz, does support explicit specification of such. This is an indicator, that, perhaps, expectations that developers would maintain such dependencies manually are futile.

Acknowledgments. We would like to thank Peter Sestoft for the help on formalizing GenDeMoG’s architecture description.

References

1. ATL - A Model Transformation Technology. <http://www.eclipse.org/atl/> (Jan 2011)
2. AtlantEcore Zoo. <http://www.emn.fr/z-info/atlanmod/index.php/Ecore> (Jan 2011)
3. EMFText Concrete Syntax Mapper. <http://www.emftext.org/index.php/EMFText> (Jan 2011)
4. Expression Language Documentation. http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.xpand.doc/help/r10_expressions_language.html (Jan 2011)
5. FreeMarker. <http://freemarker.org/> (Jan 2011)
6. Groovy - An agile dynamic language for the Java Platform. <http://groovy.codehaus.org/> (Jan 2011)
7. JavaServer Pages Overview. <http://www.oracle.com/technetwork/java/overview-138580.html> (Jan 2011)
8. Model To Model (M2M). <http://www.eclipse.org/m2m/> (Jan 2011)

9. MoDisco. <http://www.eclipse.org/MoDisco/> (Jan 2011)
10. OFBiz The Apache Open for Business Project. <http://ofbiz.apache.org/> (Jan 2011)
11. Package org.eclipse.emf.ecore. <http://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org.eclipse.emf.ecore/package-summary.html> (Jan 2011)
12. Xpand. <http://wiki.eclipse.org/Xpand> (Jan 2011)
13. Xtend documentation. http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.xpand.doc/help/Xtend_language.html (Jan 2011)
14. Xtext - Language Development Framework. <http://www.eclipse.org/Xtext/> (Jan 2011)
15. Bergmann, G., Horváth, A., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental Evaluation of Model Queries over EMF Models. In: Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I. pp. 76–90. MODELS'10, Springer-Verlag, Berlin, Heidelberg (2010)
16. Bézivin, J., Gerbé, O.: Towards a Precise Definition of the OMG/MDA Framework. In: Proceedings of the 16th IEEE international conference on Automated software engineering. pp. 273–. ASE '01, IEEE Computer Society, Washington, DC, USA (2001)
17. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: a Generic and Extensible Framework for Model Driven Reverse Engineering. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. pp. 173–174. ASE '10, ACM, New York, NY, USA (2010)
18. Estublier, J., Vega, G., Ionita, A.D.: Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications. In: MODELS. pp. 69–83 (2005)
19. Favre, J.M., Musset, J.: Rétro-ingénierie dirigée par les métamodèles : Concepts, Outils, Méthodes (June 2006)
20. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-modelling: From Theory to Practice. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS (1). Lecture Notes in Computer Science, vol. 6394, pp. 376–391. Springer (2010)
21. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: SLE. pp. 374–383 (2009)
22. Henriksson, J., Johannes, J., Zschaler, S., Aßmann, U.: Reuseware – Adding Modularity to your Language of Choice. In: Proc. of TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (2007)
23. Hessellund, A., Czarnecki, K., Wasowski, A.: Guided Development with Multiple Domain-Specific Languages. In: MODELS. pp. 46–60 (2007)
24. Howell, R.: Apache OFBiz Development: The Beginner's Tutorial. Packt Publishing (2008)
25. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation (1993)
26. Jouault, F., Vanhooff, B., Bruneliere, H., Doux, G., Berbers, Y., Bezivin, J.: Inter-DSL Coordination Support by Combining Megamodeling and Model Weaving. In: Proceedings of the 2010 ACM Symposium on Applied Computing. pp. 2011–2018. SAC '10, ACM, New York, NY, USA (2010)
27. Jouault, F., Bézivin, J., Team, A.: KM3: A DSL for Metamodel Specification. In: In proc. of 8th FMOODS, LNCS 4037. pp. 171–185. Springer (2006)
28. Oei, J.L.H., van Hemmen, L., Falkenberg, E., Brinkkemper, S.: The Meta Model Hierarchy: A Framework for Information Systems Concepts and Techniques (1992)
29. Salay, R., Mylopoulos, J., Easterbrook, S.: Using Macromodels to Manage Collections of Related Models. In: Proceedings of the 21st International Conference on Advanced Information Systems Engineering. pp. 141–155. CAiSE '09, Springer-Verlag, Berlin, Heidelberg (2009)
30. Vincent, M., Jouault, F., Bruneliere, H.: Megamodeling Software Platforms: Automated Discovery of Usable Cartography from Available Metadata

10

Tengi Interfaces for Tracing between Heterogeneous Components – GTTSE'11 (Paper C)

Tengi Interfaces for Tracing between Heterogeneous Components

Rolf-Helge Pfeiffer and Andrzej Wąsowski

IT University of Copenhagen, Software Development Group,
`{ropf,wasowski}@itu.dk`

Abstract. Contemporary software systems comprise many heterogeneous artifacts; some expressed in general programming languages, some in visual and textual domain-specific languages and some in ad hoc textual formats. During construction of a system diverse artifacts are interrelated. Only few formats, typically general programming languages, provide an interface description mechanism able to specify software component boundaries. Unfortunately, these interface mechanisms can not express relations for components containing heterogeneous artifacts.

We introduce *Tengi*, a tool that allows for the definition of software components containing heterogeneous artifacts. Tengi interfaces link components containing different textual and visual software development artifacts ranging from high-level specification documents to low-level implementation documents. We formally define and implement Tengi interfaces, a component algebra and operations on them and present a case study demonstrating Tengi's capabilities.

1 Introduction

Contemporary software systems are constructed out of a multitude of *heterogeneous* artifacts. A *software system* (...) consists of a number of separate programs, configuration files, (...) system documentation [21]. These artifacts contain information at different abstraction levels, in various languages and may be tied to different development phases. Still, they form a single whole and thus each of them provides a different view on parts or aspects of the systems. The development artifacts are related either by directly referencing each other or by referring to the same aspect of a system. Some of these relations may be explicit. Source code in a general purpose language usually contains explicit references to other software components or methods called. Other relations may be implicit. For example visual models and code generated from them are both descriptions of the same system aspect at different abstraction levels, but the detailed relation is hidden in the code generator. Some artifact relations can even remain completely undocumented, only stored in human memory. For instance requirements documents are sometimes directly translated to source code without recording any traces from them. Explicit or not, software developers continuously have to reason about and navigate across such relations, and this creates difficulties. For

example [15] points that it is a major challenge in the Linux kernel project, to maintain consistency between the kernel variability model and the source code.

This difficulty calls for investigating language oblivious tools that allow specifying components comprising heterogeneous artifacts, including definition of links across languages and formats, and allowing monitoring of, and navigation along, such links. The challenge in design of such tools lies in the tension between the generic and the specific. Heterogeneous components, and even more so relations between them, are often domain specific—intrinsically hard to support with generic tools. In this paper we take up the challenge of constructing such a generic tool, which is able to capture domain-specific component relations. To do so, we address the two questions: how to specify component boundaries for heterogeneous components? And how to technically link the components to these specifications?

Component boundaries can be specified by interfaces, which are abstract descriptions of the way in which a component relates to its context. We consider anything from files to folder structures as components. Artifacts in software development are files or multiple-files that are used together.

We present *Tengi*¹, a toolkit for defining, reusing, and relating software components by means of specifying interfaces for artifacts. Artifacts can be expressed in various languages on different levels of abstraction, ranging from high-level specification documents to low-level implementation documents and from development artifacts expressed in textual as well as in visual languages. *Tengi*, implemented as an Eclipse plug-in, extends numerous Eclipse editors with an ability to define *ports* on the edited artifacts. Further, it provides a language for specifying dependencies between these ports as interface specifications resembling *contracts*. Operators are provided for automatic checking of component compatibility and refinement and for composition of components.

Let us illustrate the problem of interrelated heterogeneous artifacts and *Tengi*'s use with a small example. Figure 2 shows a requirements document for an aspect of a simple application, implemented using Java classes (not shown yet). How do we record the knowledge that this specification fragment is implemented exactly by the three classes? *Tengi* provides a traceability mechanism based on a simple component algebra. Instead of explicitly declaring traces between the requirements document and the Java classes, with *Tengi*, a user can define ports in any documents (also free text documents like the one in Fig. 2). These ports are available in *Tengi* interfaces. Links or traces are realized by the algebra operations on such interfaces. A *Tengi* interface for the requirements document in Fig. 2 would provide a port for a certain requirement and Java classes implementing this requirement would require this port in a *Tengi* interface.

We use Eclipse to implement *Tengi*, as Eclipse is a prime representative of modern Integrated Development Environments (IDE). However, neither the problem, nor the principal solution discussed in this paper is Eclipse specific.

¹ *Tengi*, Icelandic for *interface*, was chosen to avoid conflicts with all other kind of *interfaces* appearing frequently in computer science.

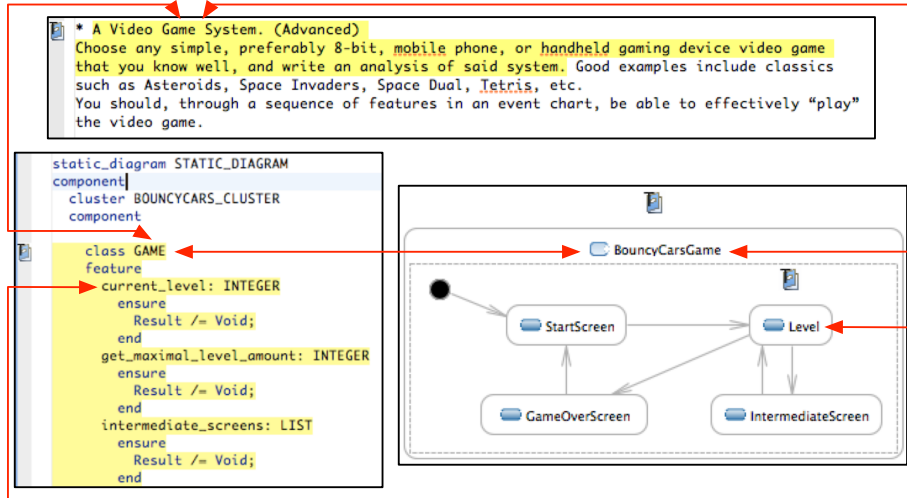


Fig. 1. Examples of software system artifacts: a requirements document fragment on top, fragment of an analysis document in formal BON (bottom left), and a UML state machine (bottom right). Concepts referring to each other are illustrated with red lines.

We proceed by motivating our work in Sect. 2 with an example of a heterogeneous software system. This system is also used for the case study in Sect. 5, which illustrates how to apply Tengi to development of a heterogeneous software system. Section 3 introduces Tengi’s component algebra followed by detailing Tengi internals in Sect. 4. We finish with a discussion of Tengi, related work and conclusion in Sections 6–8.

2 Running Example

We use a small system as our running example, also for the case study in Sect. 5. The system is a clone of a Commodore 64 video game, *Bouncy Cars* (<http://www.gb64.com/game.php?id=1049>). It was developed as an exercise in a graduate course on modeling at IT University of Copenhagen [1]. The task was to specify and implement an automatically verifiable, small-sized, object-oriented (OO) application. The system is specified using the BON method [22]. BON supports informal and formal, textual and visual specification of structural and behavioral properties of OO-systems. Visual BON is similar to UML diagrams, including constraints not unlike OCL constraints.

Our version of Bouncy Cars is an example of a heterogeneous software system. It comprises artifacts in several languages, at different levels of abstraction:

- A requirements document. A regular text file containing the exercise task in natural language.

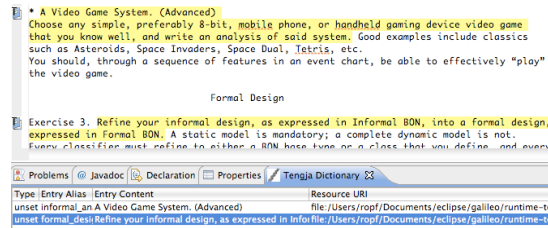


Fig. 2. The requirements document `assignment.txt` with two marked ports in the document and the Tengja dictionary (below).

```
TENGI assignment ENTITY "assignment.txt" [
  IN: {}; CONSTRAINT: true;
  OUT: { informal.analysis, formal.design };
  CONSTRAINT: informal.analysis & formal.design;
] {
  LOCATOR informal.analysis IN "assignment.txt"
    OFFSET 6692 LENGTH 179;
  LOCATOR formal.design IN "assignment.txt"
    OFFSET 7112 LENGTH 106;
}
```

Fig. 3. Interface for the document shown in Figure 2.

- A high-level analysis document. This is an informal system specification in informal textual BON.
- More concrete design documents. There are design documents in formal textual and visual BON giving system design in formal textual BON. Formal BON is refined from the former informal BON specification. Furthermore, a UML state machine specifies the system’s behavior. The UML diagram was not strictly necessary, but we have used it to replace the standard BON event chart in order to expand the number of involved languages.
- Implementation artifacts. Multiple JML annotated Java classes [13] implement the system specification.

The Bouncy Cars example contains artifacts in natural language and in six software languages. The requirements and high-level analysis documents are more abstract than design documents and implementation artifacts. Figure 1 shows three artifacts: a fragment of the requirements document in natural language in the top part of the figure; a fragment of an analysis document in formal textual BON in the bottom left part; a UML state machine specifying behavior in the bottom right part. The three artifacts describe different views on the system, at different abstraction levels. All three artifacts are implicitly interrelated. They refer to shared concepts from different view points. For example, the requirements document, the design documents in formal BON and the UML state machine, all refer to a concept “game”. Furthermore, both the formal BON and the UML state machine artifact refer to the concept “level”. Figure 1 illustrates these relations by red arrows between the shared concepts.

The main challenge in development of heterogeneous systems is caused by implicit nature of relations across artifacts and languages. They exist in human-mind, the mind of the developers, but they are not explicitly available for computers to reason about. Imagine that a new Bouncy Cars developer deletes the `GAME` class in the formal BON specification in Fig. 1. The system is now incomplete and other colleagues who require this class for their work will face errors. For instance, code generators consuming the BON specification will produce incorrect results. These errors could be avoided, if suitable warning messages about the impact of changes were produced early on. This however

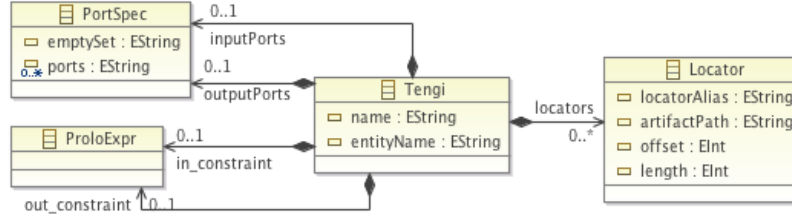


Fig. 4. Excerpt of the meta-model of the Tengi interface DSL.

requires making cross-language relations explicit and using tools to reason about them.

In this paper we set out to address this issue by investigating and implementing interfaces which allow for linking or tracing information across components containing heterogeneous artifacts.

3 Tengi Concepts

This section introduces the notions used in Tengi and Tengi’s component algebra. Two artifacts are *heterogeneous* if they are instances of different meta-models or if there exists no meta-model to describe them (the terms meta-model and language grammar are used synonymously, since they can be mapped to each other in the considered domain [3]). For example, a program artifact in Java and one in C# are heterogeneous, but also a UML class diagram and an arbitrary visual domain-specific language (DSL) are heterogeneous. In particular, there exist development artifacts that are heterogeneous to others due to a lack of a meta-model, e.g., simple text files.

3.1 Tengi Interfaces

We consider anything from files to folder structures as components. We specify component boundaries by Tengi interfaces. *Interfaces* are abstract descriptions of the way in which a component relates to its context. In Tengi interfaces this relation is expressed using ports, which could be anything from communication channels to cross-file references. Tengi interface *ports* are just abstract names that can be related to each other and to the artifacts. Tengi interfaces consider static, development-time properties of components only.

Tengi provides an interface description DSL for heterogeneous artifacts corresponding to the meta-model in Fig. 4. In the following we illustrate an example for such an interface and provide a formal definition.

Figure 3 shows an example for a Tengi interface for the required tasks of Fig. 2. This interface simply specifies two ports in `assignment.txt`, which correspond to the requirement of informal analysis and a formal design. Both of them are output ports, meaning that they are provided to the components context. Furthermore, any concretization of this interface, will have to provide informal analysis and

formal design; pointing to the locations in its components where these ports are realized. We chose to avoid constructing more complicated interfaces, for the sake of simplicity of the example. Ports, classified into inputs and outputs, provide an alias to a corresponding location. They characterize what information is provided by a component (output ports) or what information is required from the environment (input ports). Ports in the meta-model are represented by class `PortSpec` and the division into input and output ports is manifested by the containment relations `inputPorts` and `outputPorts`, see Fig. 4. Semantically, ports are Boolean variables. Assignment of true to a port means that it is 'present', otherwise it is 'absent'. *Constraints*, implemented by `PortSpec` in Fig. 4, are propositional statements that raise the expressiveness of an interface. The default constraint is `true` which for outputs means that nothing is guaranteed to be provided, and for inputs that nothing is required by the component. Both input and output ports can be constrained, see containment relations `in_constraint` and `out_constraint` in Fig. 4. A *locator* links a port to a physical location in the file system. A physical location is specified by a path to a file, an offset and the length of the marked information, see class `Locator` in Fig. 4.

Tengi relies on physical locations for the following reasons: (i) Since we provide interfaces for heterogeneous artifacts, we want the locators to be as general as possible. Physical locations are advantageous due to their meta-model independence. That is, new languages can be used with Tengi without modifying it. (ii) It is important that Tengi indicates the locators visually, raising developer's awareness of important dependencies. This is naturally done with physical locators. (iii) Furthermore, Tengi allows for the evolution of artifacts referred by locators. For example a locator can be moved, if the file containing it has been edited. This is now automatically supported for artifacts using text editors. We intend to investigate technologies that would support other evolution scenarios. Since locators relate to physical locations in files, Tengi interfaces can be considered lexical interfaces.

Definition 1. $\mathcal{T} = (I, O, \varphi, \psi)$ is an interface iff I is a set of input ports, O is a set of output ports and $I \cap O = \emptyset$; φ is a propositional constraint over I (required), which constrains the valid input port combinations; and ψ is a propositional constraint over O (provided), which constrains the valid output port combinations. Denote the set of all ports as $P = I \cup O$.

3.2 Operations on Tengi Interfaces

Composition We say that interfaces $\mathcal{T}_1 = (I_1, O_1, \varphi_1, \psi_1)$ and $\mathcal{T}_2 = (I_2, O_2, \varphi_2, \psi_2)$ are *composable* iff $I_1 \cap I_2 = O_1 \cap O_2 = \emptyset$. Composeable interfaces (and thus their components) can be composed. The interface of the *composition* is defined as an interface $\mathcal{T} = \mathcal{T}_1 \oplus \mathcal{T}_2 = (I, O, \varphi, \psi)$, where $I = I_1 \cup I_2 \setminus (O_1 \cup O_2)$ and $O = O_1 \cup O_2$. The intuition is that all ports provided (outputs) by \mathcal{T}_1 and \mathcal{T}_2 remain provided by the composition \mathcal{T} , but the required inputs that are provided within the composition itself are no longer required—thus the set difference in computing the input set. The constraints over input and output ports are given by

(i) $\varphi = \exists(l_1 \cup l_2) \cap O. \varphi_1 \wedge \varphi_2$ and (ii) $\psi = \forall l. (\varphi_1 \wedge \varphi_2) \rightarrow (\psi_1 \wedge \psi_2)$ where the existential elimination of a variable $x \in X$ from formula φ over variables X is the formula $\exists x. \varphi = \varphi[0/x] \vee \varphi[1/x]$, which extends to $\exists A. \varphi = \exists x_1. \dots \exists x_n. \varphi$ for a set of variables $A = \{x_1, \dots, x_n\} \subseteq X$. Dually the universal elimination of x from ψ is the formula $\forall x. \psi = \psi[0/x] \wedge \psi[1/x]$, generalizing to $\forall A. \psi = \forall x_1. \dots \forall x_n. \psi$ for the same set of variables A . Intuitively, the first point above means that inputs required by the components are still required by the composition, except for the part of the constraint, which has been satisfied. Point two above states that the component might provide any combination of outputs such that regardless of what inputs are given (that satisfy the required constraint) this combination still can be delivered. Two interfaces are *compatible* if their output constraint ψ is satisfiable. This corresponds to a requirement that a precondition of a procedure is consistent. We only require satisfiability (and not validity) in order to achieve an optimistic notion of composition [4], in which a component is useful as long as there exists a context, with which it is consistent. When composing two interfaces their locator lists are simply concatenated. Tengi implements composition using an *Xpand* [2] template, so by composing the syntactical representations of interfaces.

Subtyping (or *refinement*) is a binary relation that allows comparing interfaces, in a fashion that is similar to object oriented generalization hierarchy. We say that \mathcal{T}_1 is a subtype of \mathcal{T}_2 iff: (i) $l_1 = l_2$ and $O_2 = O_1$ and (ii) $\varphi_1 \rightarrow \varphi_2$ and $\psi_2 \rightarrow \psi_1$. Presently, checks of propositional statements in Tengi are implemented using binary decision diagrams (BDD) [5]. The subtyping definition is somewhat rigid in that it requires that both interfaces completely agree on their input and output alphabets. This is not a limitation. If we want to place a subtype interface in a context of the supertype, we basically need to add extra constraints setting the unused inputs and outputs of the context to false.

Conformance checking is a check of an interface against one or more development artifacts. More precisely, all development artifacts are checked if they provide the information specified in the corresponding Tengi interfaces. An interface conforms to the corresponding artifacts iff for all the locators exists a marker on the appropriate file with the appropriate physical locations.

This component algebra, albeit simple, exhibits all crucial properties that are expected of such: **(i)** The composition operator is associative and commutative. **(ii)** The composition operator is optimistic [4] **(iii)** The refinement relation is a preorder (reflexive and transitive). **(iv)** Composition satisfies independent-implementability [4], i.e., it is possible to replace an interface by any of its refinements in any context, without breaking compatibility (as long as this refinement does not introduce clashing names—a technicality caused by the fact that all port names are global).

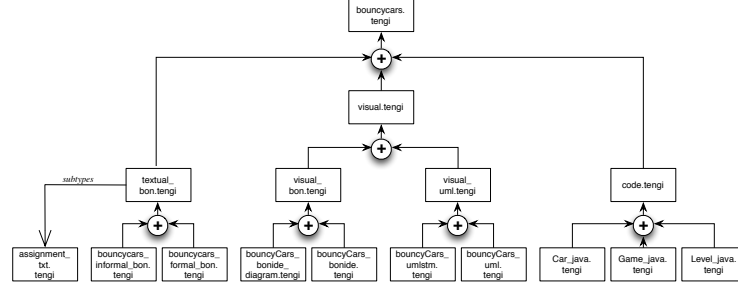


Fig. 6. Interfaces for all the artifacts in the case study project.

In general, it is not trivial to provide physical file locators for elements of visual languages.

Tengi supports computing physical locators for visual model elements automatically, using its traceability component *Tengja*² [17]. With Tengja it requires just a button click to move from a marked element to the persistent models opened in text editors, with the highlighted text corresponding to the original model element. This functionality is instantly available for all DSLs defined with Ecore, and all GMF and EMF generated DSL editors. Tengja establishes the connections, the traceability links between model elements in visual syntax and their corresponding serialization syntax and highlights these elements.

But how does Tengja bridge gap between the visual layout representation, its visual concrete syntax, and persistent textual representation, the serialization syntax?

Technically, Tengja is an extension to Eclipse, which recovers the links between the abstract and concrete syntax and the serialization syntax of models by observing the persistence mechanism. Since Eclipse’s standard persistence mechanism obscures traces, and since we aim at a reusable and non-invasive tracing toolkit, we settle on observing the standard persistence mechanism with an aspect, recording the context elements and linking them to the generated syntax. The aspect observes the top-most traversing method and its callees in `org.eclipse.emf.ecore.xmi.impl`. It observes the sequence of model elements that get treated in the control-flow of these methods, and keeps track of start and stop positions in the generated stream of text in serialization syntax for a model element. Subsequently, it maps model elements to indices in the generated serialization stream. Thereby, we can trace each model element to its textual representation and establish an explicit mapping between them. The mapping is then exposed to the development environment via the Tengja dictionary and can be used in Tengi interfaces.

Tengja allows to mark arbitrary model elements in Ecore-based visual models and to navigate from the respective element to all related other model elements and textual representations in abstract syntax, visual concrete syntax, and serialization syntax, and furthermore, to persist those connections or traceability

² Tengja, Icelandic *connect*, was chosen to avoid conflicts with “connects”, “connections”, and “connectors” appearing frequently in MDE literature.

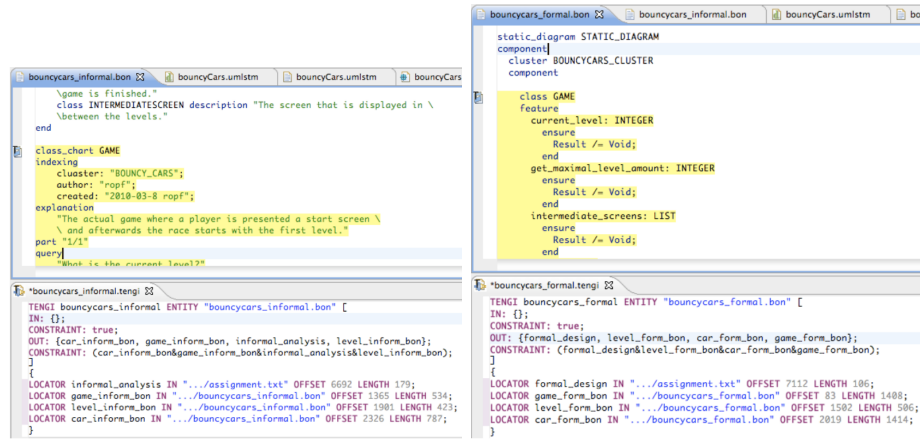


Fig. 7. Excerpt of the analysis document in informal BON (bouncycars_informal.bon) with a marked port on top and below the corresponding Tengi interface low (bouncycars_informal.tengi). **Fig. 8.** Excerpt of the design document in formal BON (bouncycars_formal.bon) with a marked port on top and below the corresponding Tengi interface low (bouncycars_formal.tengi).

links in a global locator dictionary. To define locators in Tengi interfaces this dictionary can be used to drag single entries into the interface definition.

5 Case Study

This section demonstrates how *Tengi* is used in a project containing multiple heterogeneous artifacts, how the Tengi interfaces are defined, and what are the results of applying operators to them. We use the Bouncy Cars project introduced in Sect. 2. Notably, we successfully apply Tengi to textual and visual languages and editors, developed independently of this work by other authors.

Figure 6 presents the overview of the entire project using the composition structure of Tengi interfaces. Rectangles represent Tengi interfaces. The interfaces in the bottom row correspond directly to the individual artifacts of the kinds listed above. We construct the abstract interface specification for the entire *Bouncy Cars* project using stepwise bottom-up composition with the \oplus composition operator introduced in Sect. 3. The Tengi interfaces for basic components (files) are presented as follows: Figure 3 shows the interface for the requirements document `assignment.txt`, itself presented in Fig. 2; Interfaces for the informal and formal textual BON specifications are found in the bottom of Fig. 7 and in Fig. 8 respectively; In Fig. 13 interfaces for the visual BON model and its corresponding data model are shown, see Sect. 4; The interface for the UML state machine is in Fig. 10; and Fig. 11 shows interfaces for the Java classes `Car.java`, `Level.java` and `Game.java`. All file paths in interfaces in this paper are abbreviated to avoid clutter. Complete model files are available at www.itu.dk/people/ropf/src/tengi.

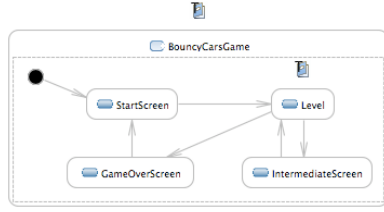


Fig. 9. UML state machine model `bouncyCars.umlstm` with two elements marked as ports, which appear in the Tengi dictionary

```

Tengi visual.uml ENTITY
  "(bouncyCars.uml.tengi+bouncyCars.umlstm.tengi)"[
    IN: { }; CONSTRAINT: true;
    OUT: { level.uml.data, game.uml.vis, level.uml.vis };
    CONSTRAINT: level.uml.data & game.uml.data
      & level.uml.vis & game.uml.vis;
  ] {
    LOCATOR level.uml.data IN "bouncyCars.uml" OFFSET 1313 LENGTH 79;
    LOCATOR game.uml.data IN "bouncyCars.uml" OFFSET 975 LENGTH 1469;
    LOCATOR game.uml.vis IN "bouncyCars.umlstm" OFFSET 335 LENGTH 5333;
    LOCATOR level.uml.vis IN "bouncyCars.umlstm" OFFSET 2385 LENGTH 941;
  }

```

Fig. 10. Composition of interfaces for the UML state machine and its corresponding data model (Fig. 9).

All basic components listed above provide views on the same domain, the Bouncy car game, from the point of view of different abstraction levels. That is, they all contain pieces of information that are related to each other. For example, all of the basic components care about a “game” that contains multiple “levels” and some of them tell something about a “car”. Similarly, the state `Level` in `bouncyCars.umlstm` and the class `Level` in `bouncyCars.bonide_diagram` refer to each other, but there is no explicit link that allows for automatic reasoning over such relations. Tengi interfaces establish such a link.

Let us examine a bit closer the interfaces of files `bouncyCars.formal.bon` and `bouncyCars.umlstm` (Fig. 8–9). The interfaces are shown in figures 8 and 10 respectively. The first one states that the component `bouncycars.formal.bon` provides, among others, a port `level_form_bon` that refers via its locator to the specification of a class `Level`. The Tengi interface for the UML state machine (Fig. 10) requires, amongst others, the formal specification of `Level` in BON (`level_form_bon`), to provide the state `Level` via two new ports `level_uuml_data` and `level_uuml_vis`. These are then used to trace the refinement further to Java implementation in other interfaces.

The Tengi interface `textual_bon.tengi` is a simple example of refinement (subtyping) of `assignment_txt.tengi`. Both interfaces provide the ports `informal_analysis` and `formal_design`, the former, since it corresponds to the high-level requirements document, more abstract the latter more concrete. This means that `textual_bon.tengi` provides both the informal analysis and the formal design and explicitly indicates, by means of locators, where these are placed in the model.

The composition of all interfaces in the case study results in the synthesized interface presented in Fig. 14. The overall interface shows no input and thus no constraints on inputs. This is expected as the entire system is supposed to be complete, and should not require anything. We also remark that the output constraint warrants satisfaction of `informal_analysis` and `formal_design`, which can be traced all the way back to the initial requirement.

This case study demonstrates that Tengi allows defining interfaces and thereby components for heterogeneous development artifacts (here free text files, GMF and EMF models, and Java source code), and further to process such interfaces

```

TENGI Game ENTITY "Game.java" [
  IN:
    { game.uml.data, game.uml.vis, game.bon.data,
      game.bon.vis };
  CONSTRAINT:
    ( game.uml.data & game.uml.vis & game.bon.data &
      game.bon.vis );
  OUT: { game.java };
  CONSTRAINT: game.java;
] {
  LOCATOR game.uml.data IN "bouncyCars.uml" OFFSET 975 LENGTH 1469;
  LOCATOR game.uml.vis IN "bouncyCars.umlstm" OFFSET 335 LENGTH 5333;
  LOCATOR game.bon.data IN "bouncyCars.bonide" OFFSET 1302 LENGTH 2100;
  LOCATOR game.bon.vis IN "bouncyCars.bonide.diagram" OFFSET 19177 LENGTH 31651;
  LOCATOR game.java IN "Game.java" OFFSET 94 LENGTH 2228;
}

TENGI Level ENTITY "Level.java" [
  IN: { level.uml.data, level.uml.vis, level.bon.data, level.bon.vis };
  CONSTRAINT: (level.uml.data & level.uml.vis & level.bon.data & level.bon.vis);
  OUT: { level.java };
  CONSTRAINT: level.java;
] {
  LOCATOR level.uml.data IN "bouncyCars.uml" OFFSET 1313 LENGTH 79;
  LOCATOR level.uml.vis IN "bouncyCars.umlstm" OFFSET 2385 LENGTH 941;
  LOCATOR level.bon.data IN "bouncyCars.bonide" OFFSET 274 LENGTH 1023;
  LOCATOR level.bon.vis IN "bouncyCars.bonide.diagram" OFFSET 630 LENGTH 18540;
  LOCATOR level.java IN "Level.java" OFFSET 43 LENGTH 1511; }

```

Fig. 11. Tengi interfaces for the Java classes

using appropriate interface operations. The interface specifications, particularly interface's provisions and requirements, not only define components, but also provide traceability links by marking port's locations explicitly and interrelating ports using the constraints and component algebra operators.

In this section we have constructed the Tengi interface in a bottom-up fashion, starting with the interfaces for basic components. This is not generally required, as Tengi allows definition of components of any granularity.

6 Discussion

Currently, Tengi allows for the following:

- Defining Tengi interfaces using a textual DSL. The tool provides an appropriate editor with syntax highlighting, live validation, and code completion.
- Applying operations to Tengi interfaces, i.e., composition, subtype checking, compatibility checking, and conformance checking.
- Establishing links between visual model elements and their serialization syntax and organizing them in a global dictionary.
- Highlighting of information which is referred by Tengi locators in textual and graphical editors (except for tree viewers).

Tengi itself relies on Eclipse's model-driven software development tools. For example, the interface editor was generated using *Xtext*. That is, Tengi interfaces are internally represented by Ecore-based models. The composition operation is implemented via an *Xpand* template. *Xtext* and *Xpand* are both parts of the Eclipse Modeling Project [6]. Furthermore, interface operations are implemented using Binary Decision Diagrams (BDD's), in particular the JavaBDD [23] library, for the representation of the port specification constraints.

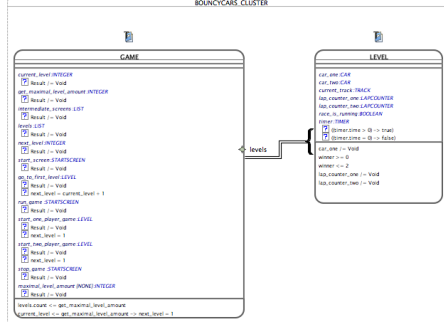


Fig. 12. A visual BOM model for the Game entity which contains levels.

```

TENGI bouncyCars.bonide_diagram ENTITY
    "bouncyCars.bonide_diagram.tengi" [
        IN: { game.bon_data, level.bon_data };
        CONSTRAINT: (game.bon_data & level.bon_data);
        OUT: { game.bon_vis, level.bon_vis };
        CONSTRAINT: (game.bon_vis & level.bon_vis);
    ] { LOCATOR game.bon_vis IN "bouncyCars.bonide_diagram"
        OFFSET 19177 LENGTH 31651;
        LOCATOR game.bon_data IN "bouncyCars.bonide"
            OFFSET 1302 LENGTH 2100;
        LOCATOR level.bon_vis IN "bouncyCars.bonide_diagram"
            OFFSET 630 LENGTH 18540;
        LOCATOR level.bon_data IN "bouncyCars.bonide"
            OFFSET 274 LENGTH 1023; }

TENGI bouncyCars.bonide ENTITY "bouncyCars.bonide.tengi" [
    IN: { }; CONSTRAINT: true;
    OUT: { game.bon_data, level.bon_data };
    CONSTRAINT: (game.bon_data & level.bon_data);
    ] { LOCATOR game.bon_data IN "bouncyCars.bonide"
        OFFSET 1302 LENGTH 2100;
        LOCATOR level.bon_data IN "bouncyCars.bonide"
            OFFSET 274 LENGTH 1023; }

```

Fig. 13. Interfaces for the visual model and its data model (Fig. 12)

The most advanced technically part of Tengi is its traceability mechanism, *Tengja*, that allows linking physical locations in files to model elements in modeling editors by applying suitable aspects to Eclipse editors. *Tengja*, described in more detail in a preliminary version of this work [17], modifies the standard serialization mechanism of Eclipse using aspect oriented programming. The *Tengja* aspect observes model serialization to establish physical positions of model elements in files in a meta-model independent manner. Thus it is not required that users of Tengi manually change modeling and programming editors to allow for visualization of ports.

Tengi is generally applicable to development projects that are executed in the Eclipse IDE. However, any other modern IDE with support for visual models could have been chosen to serve as platform for Tengi. As mentioned earlier, Tengi is able to deal with all textual development artifacts as well as visual models that are EMF/GMF based. To our understanding, this covers the most important artifacts in current software development projects. Supporting new artifact types would require the extension of the Tengi tool to deal with the artifact's specific editor, since Tengi distinguishes and handles artifacts based on their specific editor.

Tengi interfaces are separated, i.e., non-invasive, to the corresponding development artifacts. We could have investigated an invasive approach. That would mean that information that should appear in development artifacts would be directly marked within the development artifact. We decided against this approach to make the use of Tengi optional and ease adoption in legacy projects. Further, non-invasive component definition approaches can be researched more easily since existing projects do not need to be inherently modified. A drawback of choosing a non-invasive approach is that it requires the use of additional tools in the development process, here it is the use of Eclipse IDE with our plugins.

As described in Sect. 3, Tengi allows for the specification of ports for arbitrary information in development artifacts. It might be a shortcoming that such ports

```

TENGI bouncycars
  ENTITY "Car.java+Game.java+Level.java+bouncyCars.bonide_diagram.tengi+bouncyCars.bonide.tengi+
    bouncyCars.uml.tengi+bouncyCars.umlstm.tengi+bouncycars.formal.bon+bouncycars.informal.bon"
[
  IN: { }; CONSTRAINT: true;
  OUT: {car_java, game_java, level_java, game_bon_vis, level_bon_vis,
    game_bon_data, level_bon_data, level_uml_data, game_uml_data, game_uml_vis,
    level_uml_vis, formal_design, level_form_bon, car_form_bon, game_form_bon,
    car_inform_bon, game_inform_bon, informal_analysis, level_inform_bon};
  CONSTRAINT: car_java & game_java & level_java & game_bon_vis & level_bon_vis &
    game_bon_data & level_bon_data & level_uml_data & game_uml_data & game_uml_vis &
    level_uml_vis & formal_design & level_form_bon & car_form_bon & game_form_bon &
    car_inform_bon & game_inform_bon & informal_analysis & level_inform_bon;
] {
  LOCATOR car_java IN "bouncycars/Car.java" OFFSET 65 LENGTH 2471;
  ...
  LOCATOR car_inform_bon IN "bouncycars.informal.bon" OFFSET 2326 LENGTH 787;}

```

Fig. 14. The interface synthesized for the BouncyCars project.

are presently untyped and that it is thereby possible to construct Tengi interfaces which relate information that either should not be related. On the other hand, we think that untyped ports are advantageous since they do not restrict developers in the specification of interfaces and allow to apply Tengi interfaces in various settings and environments and under various requirements. For example, with untyped ports it is possible that in one development project Tengi interfaces relate only documentation artifacts that relate whole chapters to each other, whereas another development projects relates only method names of Java classes to each other.

7 Related Work

The composition operators in Tengi's algebra is a simplified and regularized version of the algebra presented in [11], originally inspired by the input/output interfaces in [4]. Unlike in [11], there is no concept of meta-interfaces in Tengi, since Tengi regards all software development artifacts as first level artifacts. Also this version of the component algebra does not reason about internal dependencies between outputs and inputs within the component.

We have settled on a simple, propositional specification language that can be efficiently treated using state of the art technologies like SAT-solving or BDDs. It was not our objective to create a very rich component algebra. One starting point to get an overview of this research area is the anthology by Liu and Jifeng [14], which discusses languages beyond propositional logics.

Static interrelations of heterogeneous software development artifacts are currently not widely discussed. The work of Henriksson et al. [9] is very close to ours. They provide an approach to add modularity to arbitrary languages by enriching their grammars or meta-models with variation points. That is they provide an *invasive* modularization support. Also Heidenreich et al. [8] take a similar route. Both works require an extension of a language's specifications to support modularity. First, the described mechanism is language focused, i.e., each new language's grammar needs to be modified before supporting modularization support, and second the described approach is invasive in the sense that no

separate interfaces are constructed but the artifacts itself define their provisions and requirements.

Current traceability solutions like macromodels [19], mega-models [12], trace models [7], and relation models [16] rely on an explicit model containing traces between different model elements. Such explicit models can be regarded as composed or “wired” interfaces where the trace ends are ports of interfaces. Differently, to Tengi all these solutions interrelate models, whereas Tengi abstracts even more by concentrating on visual and textual artifacts in their textual representation. Similarly, SmartEMF [10] checks cross-reference constraints to support developers and cross-references may be regarded as interface ports of implicit interfaces. The present paper can be seen as a generalization of [10] in the (specific) sense that Tengi could also be used to address the same problem.

8 Conclusion and Future Work

This paper presented Tengi a tool that allows for the construction of components of heterogeneous development artifacts using interfaces. Tengi interfaces rely on ports to physical locations. Combined with the presented component algebra, such ports describe relations between heterogeneous artifacts themselves. The tool provides a textual DSL for defining interfaces for heterogeneous software development artifacts, an appropriate editor including syntax highlighting, live validation, and code completion, and operations on the interfaces. Furthermore, the tool includes Tengja, a mechanism for connecting visual model elements with their serialization syntax and thereby enabling their integration into a global, IDE-wide, locator dictionary, so that they can be used in Tengi interfaces. The tool is integrated into the Eclipse IDE as a plug-in. To demonstrate the abilities and advantages of our tool we provided a case study, that applies Tengi in the development process of a small sized software system.

In future we will continue developing Tengi. We want to investigate use of structured locators. We intend to use query languages and express locators as queries to particular information. This is not trivial, since we would still like to support evolution of development artifacts with interfaces, which requires being able to evolve queries in parallel. It is much simpler to track evolution of physical location, than of complex structures defined by queries. We will address this issue by investigating heterogeneous development artifacts with respect to commonalities and differences in their structure. This will result in more development artifacts being usable in Tengi and a standard mechanism for registering new development artifacts to Tengi. We consider evaluating the tool in a real-world software development scenario to understand its impact on developers and on the quality of software produced.

Acknowledgements. The assignment of Fig. 2 is due to Joe Kiniry, who also introduced Pfeiffer to BON. We thank Ralph Skinner for developing a GMF-based development environment for BON [20], and for supporting us in its use. We also thank the GTTSE reviewers for their constructive comments on earlier versions of this paper.

References

1. Advanced Models and Programs, Course Homepage. www.itu.dk/research/pls/wiki/index.php/AMP-Spring2010 (2010)
2. Xpand. <http://wiki.eclipse.org/Xpand> (May 2010)
3. Alanen, M., Porres, I.: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Tech. rep., Turku Centre for Computer Science (2003)
4. Alfaro, L., Henzinger, T.A.: Interface Theories for Component-Based Design. In: EMSOFT (2001)
5. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 8, 677–691 (1986)
6. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language Toolkit. Addison-Wesley (2009)
7. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-modelling: From theory to practice. In: MoDELS (1). pp. 376–391 (2010)
8. Heidenreich, F., Johannes, J., Zschaler, S.: Aspect Orientation for Your Language of Choice. In: Workshop on Aspect-Oriented Modeling (AOM at MoDELS) (2007)
9. Henriksson, J., Johannes, J., Zschaler, S., Aßmann, U.: Reuseware - Adding Modularity to Your Language of Choice. *Journal of Object Technology* 6(9) (2007)
10. Hessellund, A., Czarnecki, K., Wąsowski, A.: Guided Development with Multiple Domain-Specific Languages. In: MoDELS'07 (2007)
11. Hessellund, A., Wąsowski, A.: Interfaces and Metainterfaces for Models and Metamodels. In: MoDELS'08. pp. 401–415. Springer-Verlag, Berlin, Heidelberg (2008)
12. Jouault, F., Vanhooft, B., Bruneliere, H., Doux, G., Berbers, Y., Bezivin, J.: Inter-DSL Coordination Support by Combining Megamodeling and Model Weaving. In: Proceedings of the 2010 ACM Symposium on Applied Computing (2010)
13. Leavens, G.T., Cheon, Y.: Design by Contract with JML (2004)
14. Liu, Z., Jifeng, H. (eds.): Mathematical Frameworks for Component Software: Models for Analysis and Synthesis. Springer (2007)
15. Lotufo, R., She, S., Berger, T., Czarnecki, K., Wąsowski, A.: Evolution of the Linux kernel variability model. In: SPLC'1. LNCS, vol. 6287. Springer (2010)
16. Pfeiffer, R.H., Wasowski, A.: Taming the Confusion of Languages. In: Proceedings of the 7th European Conference on Modelling Foundations and Applications (2011)
17. Pfeiffer, R.H., Wąsowski, A.: An Aspect-based Traceability Mechanism for Domain Specific Languages. In: ECMFA Traceability Workshop (2010)
18. Reenskaug, T.M.H.: Models - Views - Controllers. heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf (1979)
19. Salay, R., Mylopoulos, J., Easterbrook, S.: Using Macromodels to Manage Collections of Related Models. In: Proc. of the 21st International Conference on Advanced Information Systems Engineering (2009)
20. Skinner, R.: An Integrated Development Environment for BON. Master's thesis, School of Computer Science and Informatics, University College Dublin (2010)
21. Sommerville, I.: Software Engineering. International Computer Sciences Series, Addison Wesley, Harlow, UK, 8th edn. (2006)
22. Waldén, K., Nerson, J.M.: Seamless object-oriented software architecture: analysis and design of reliable systems. Prentice-Hall, Inc. (1995)
23. Whaley, J.: JavaBDD Project Homepage. javabdd.sourceforge.net/ (Mar 2012)

TexMo: A Multi-Language Development Environment

Rolf-Helge Pfeiffer and Andrzej Wąsowski

IT University of Copenhagen, Denmark
`{ropf,wasowski}@itu.dk`

Abstract. Contemporary software systems contain a large number of artifacts expressed in multiple languages, ranging from domain-specific languages to general purpose languages. These artifacts are interrelated to form software systems. Existing development environments insufficiently support handling relations between artifacts in multiple languages. This paper presents a taxonomy for multi-language development environments, organized according to language representation, representation of relations between languages, and types of these relations. Additionally, we present TexMo, a prototype of a multi-language development environment, which uses an explicit relation model and implements visualization, static checking, navigation, and refactoring of cross-language relations. We evaluate TexMo by applying it to development of a web-application, JTrac, and provide preliminary evidence of its feasibility by running user tests and interviews.

1 Introduction

Maintenance and enhancement of software systems is expensive and time consuming. Between 85% to 90% of project budgets go to legacy system operation and maintenance [6]. Lientz et. al. [19] state that 75% to 80% of system and programming resources are used for enhancement and maintenance, where alone understanding of the system stands for 50% to 90% percent of these costs [25].

Contemporary software systems are implemented using multiple languages. For example, PHP developers regularly use 1 to 2 languages besides PHP [1]. The situation is even more complex in large enterprise systems. The code base of OFBiz, an industrial quality open-source ERP system contains more than 30 languages including General Purpose Languages (GPL), several XML-based Domain-Specific Languages (DSL), config files, property files, and build scripts. ADempiere, another industrial quality ERP system, uses 19 languages. ECommerce systems Magento and X-Cart utilize more than 10 languages each.¹ Systems utilizing the model-driven development paradigm additionally rely on multiple languages for model management, e.g., meta-modelling (UML, Ecore, etc.) model transformation (QVT ATL, etc.), code generation (Acceleo, XPand, etc.), and model validation (OCL, etc.).²

¹ See ofbiz.apache.org, adempiere.com, magentoocommerce.com, x-cart.com

² See uml.org, eclipse.org/modeling/emf, omg.org/spec/QVT, eclipse.org/atl, eclipse.org/acceleo, wiki.eclipse.org/XPand, omg.org/spec/OCL respectively.

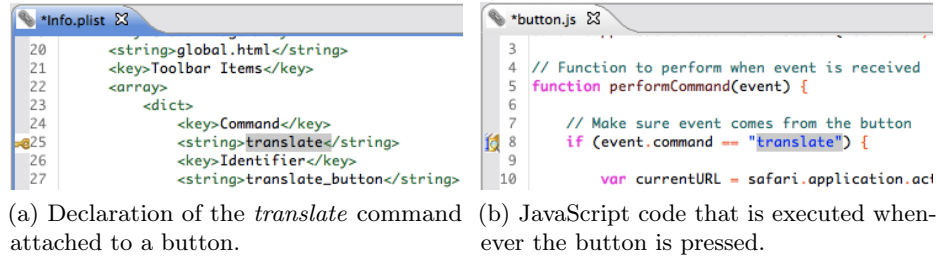


Fig. 1: Declaration of a command and its use.

We call software systems using multiple languages, *Multi-Language Software Systems (MLSS)*. Obviously, the majority of modern software systems are MLSSs.

Development artifacts in MLSS can be models, source code, property files, etc. To simplify presentation, we refer to all these as *mograms* [18]. Mograms in MLSS are often heavily interrelated. For example, OFBiz contains many hundreds of relations across its languages [23,13]. Unfortunately, relations across language boundaries are fragile. They are broken easily, as development environments neither visualize nor statically check them.

Consider the following scenario. For simplicity of presentation we use a small example. Our work, though, is not tight to a particular selection of languages, or the particular example system.

Example Scenario. Bob develops a *Safari* web browser *extension*. The extension contributes a button to Safari’s menu bar. Pressing the button translates the current web-page to English using *Google translate* and presents it in a new tab. Browser extensions are usually built using HTML, CSS and JavaScript. Bob’s extension consists of three source code files: *Info.plist*, *button.js*, and *global.html*.

Plist files serve as an interface for the extension. They tell Safari what the extension contributes to the UI. In Bob’s extension, the *Plist* file contains the declaration of a **translate** command attached to a toolbar button (Fig. 1a). *JavaScript* code contains logic attached to buttons, menus, etc. Bob’s *button.js* forwards the current URL to Google’s translation service whenever the corresponding button is pressed (Fig. 1b). Every extension contains a *global.html* file, which is never displayed. It contains code which is loaded at browser start-up or when the extension is enabled. It is used to provide code for extension buttons, menus, etc. Bob’s *global.html* file (not shown here) contains only a single script tag pointing to *button.js*.

In Fig. 1a the **translate** command for the button is defined. Fig. 1b shows how the **translate** command is used in *button.js* in a string literal. This is an example of a *string-based reference* to *Info.plist*. Such string-based references are common in development of MLSSs.

Now, imagine Bob renaming the command in *Info.plist* from **translate** to its Danish equivalent **oversæt**. Obviously, the browser plugin will not work anymore

since the JavaScript code in *button.js* is referring to a non-existing command. Symmetrically, the reference is broken whenever the “`translate`” string literal is modified in the *button.js* file, without the corresponding update to *Info.plist*. \square

Existing Integrated Development Environments (IDE) do not directly support development of MLSSs. IDEs do not visualize cross-language relations (markers left to line numbers and gray highlighting in Fig. 1). Neither do they check statically for consistency of cross-language relations, or provide refactorings across mograms in multiple languages. We are out to change this and enhance IDEs into *Multi-Language Development Environments (MLDE)*.

This paper introduces a taxonomy of design choices for MLDEs (Sec. 2). The purpose of this taxonomy is twofold. First, it serves as requirements list for implementing MLDEs, and second it allows for classification of such. We argue for the validity of our taxonomy by a survey of related literature and tools.

As the second main contribution, the paper presents TexMo (Sec. 3), an MLDE prototype supporting textual GPLs and DSLs. It implements actions for *visualization* of, *static checking* of, *navigation* along, and *refactoring* of inter-language relations, and facilities to declare inter-language relations. Additionally, TexMo provides standard editor mechanisms such as syntax highlighting. We position TexMo in our taxonomy and evaluate it by applying it to development of an MLSS and user tests followed by interviews.

2 Taxonomy of Multi-Language Development Environments

Popular IDEs like Eclipse or NetBeans implement separate editors for every language they support. A typical IDE provides separate Java, HTML, and XML editors, even though these editors are used to build systems mixing all these languages. Representing languages separately allows for an easy and modular extension of IDEs to support new programming languages. Usually, IDEs keep an *Abstract Syntax Tree (AST)* in memory and automatically synchronize it with modifications applied to concrete syntax. IDE editors exploit the AST to facilitate source code navigation and refactorings, ranging from basic renamings to elaborate code transformations such as *method pull ups*.

Inter-language relations are a major problem in development of MLSS [23,13,12]. Since they are mostly implicit, they hinder modification and evolution of MLSS. An MLDE is an IDE that addresses this challenge by not only integrating tools into a uniform working experience, as IDEs do, but also by integrating languages with each other. MLDEs support across language boundaries the mechanisms implemented by IDEs for every language separately.

We surveyed IDEs, programming editors³, and literature to understand the kind of development support they provide. We realized that 4 features, that

³ IDEs: Eclipse, NetBeans, IntelliJ Idea, MonoDevelop, XCode. Editors: MacVim, Emacs, jEdit, TextWrangler, TextMate, Sublime Text 2, Fraise, Smultron, Tincta, Kod, gedit, Ninja IDE. (See project websites at: www.itu.dk/~ropf/download/list.txt)

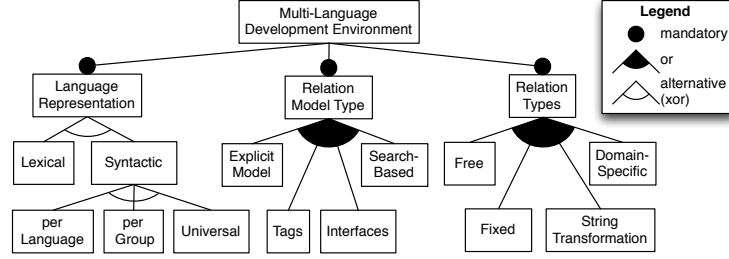


Fig. 2: Taxonomy for multi-language development environments.

visualization, navigation, static checking, and refactoring are implemented by all IDEs and by some programming editors. Thus, in order to support developers best, MLSS need to consider delivering these features across language boundaries as their essential requirements:

1. *Visualization.* An MLDE has to highlight and/or visualize inter-language references. Visualizations can range from basic markers, as for instance in the style of Fig. 1 to elaborate visualization mechanisms such as treemaps [7].
2. *Navigation.* An MLDE has to allow navigating along inter-language relations. In Fig. 1, the developer can request to automatically open *button.js* and jump to line 8, when editing *Info.plist*. All surveyed IDEs allow to navigate source code. Further, IDEs allow for source code to documentation navigation, a basic multi-language navigation.
3. *Static Checking.* An MLDE has to statically check the integrity of inter-language relations. As soon as a developer breaks a relation, the error is indicated to show that the system will not run error free. All surveyed IDEs provide static checking by visualizing errors and warnings.
4. *Refactoring.* An MLDE has to implement refactorings, which allow easy fixing of broken inter-language relations. Different IDEs implement a different amount of refactorings per language. Particularly, rename refactorings seem to be widely used in current IDEs [21,31].

To address these requirements one needs to make three main design decisions: **a)** *How to represent different programming languages?* **b)** *How to inter-relate them with each other?* **c)** *Using which kind of relations?*

Systematizing the answers to these questions led us to a domain model characterizing MLDEs. We present this model in Fig. 2 using the feature modeling notation [5,16]. An MLDE always represents programs based on the their language (*Language Representation*). Furthermore, an MLDE has to represent inter-language relations (*Relation Model Type*). This feature is essential for augmenting an IDE to an MLDE. Finally, an MLDE associates types to inter-language relations (*Relation Types*). An IDE first becomes an MLDE if it supports inter-language relations, i.e., as it implements an instance of this model.

The following subsections detail and exemplify the fundamental MLDE characteristics of our taxonomy. References to the surveyed literature are inlined.

2.1 Language Representation Types

We consider two main types of language representation, lexical and syntactic language representation. The former always works on an artifact directly without constructing a more elaborate representation, whereas the latter is always based on a richer data-structure representing mograms in a certain language. Syntactic language representation can represent mograms per language, per language group, or universally.

Lexical Representation. Most text editors, such as EMacs, Vim, and jEdit, implement lexical representation. Mograms are loaded into a buffer in a language agnostic manner. Syntax highlighting is implemented solely based on matching tokens. Due to lack of sufficient information about the edited mogram such editors provide limited support for static checking, code navigation, and refactoring.

Syntactic Representation. Per Language. Typical IDEs represent mograms in any given language using a separate AST, or a similar richer data structure capturing a mogram's structure; for instance Eclipse, NetBeans, etc. Unlike lexical representation, a structured, typed representation allows for implementation of static checking and navigation within and between mograms of a single language but not across languages. The advantage using per language representation, compared to per language group and universal representation, is that IDEs are easily extensible to support new languages.

Using models to represent source code is getting more and more popular⁴. This is facilitated by emergence of language workbenches such as EMFText, XText, Spoofox, etc.⁵ The MoDisco [4] project, a model-driven framework for software modernization and evolution, represents Java, JSP, and XML source code as EMF models, where each language is represented by its own distinct model. These models are a high-level description of an analyzed system and are used for transformation into a new representation. The same principle of abstracting a programming language into an EMF model representation is implemented in JaMoPP [11]. Similarly, JavaML [3] uses XML for a structural representation of Java source code. On the other hand, SmartEMF [12] translates XML-based DSLs to EMF models and maps them to a Prolog knowledge base. The EMF models realize a per language representation. Similarly, we represent OFBiz' DSLs and Java using EMF models to handle inter-component and inter-language relations [23].

Syntactic Representation. Per Language Group. A single model can represent multiple languages sharing commonalities. Some languages are mixed or embedded into each other, e.g., SQL embedded in C++. Some languages extend others, e.g., AspectJ extends Java. Furthermore, languages are often used together, such as JavaScript, HTML, XML, and CSS in web development. Using a per language

⁴ Language workbenches mostly use modeling technology to represent ASTs. Therefore, we use the terms AST and model synonymously in this paper.

⁵ See www.languageworkbenches.net for the annual language workbench competition.

group representation allows increased reuse in implementation of navigation, static checks and refactoring in MLDEs, because support for each language does not need to be implemented separately.

For example, the IntelliJ IDEA IDE (jetbrains.com/idea), supports code completion for SQL statements embedded as strings in Java code. X-Develop [28,27] implements an extensible model for language group representation to provide refactoring across languages. AspectJ’s compiler generates an AST for Java as well as for AspectJ aspects simultaneously. Similarly, the WebDSL framework represents mograms in its collection of DSLs for web development in a single AST [8]. *Meta*, a language family definition language, allows the grouping of languages by characteristics, e.g., object-oriented languages in *Meta(oopl)* [14]. The Prolog knowledge base in [12] can be considered as a language group representation for OFBiz’ DSLs, used to check for inter-language constraints.

Syntactic Representation. Universal. Universal representations use a single model to capture the structure of mograms in any language. They can represent any version of any language, even of languages not invented yet. Universal representations use simple but generic concepts to represent key language concepts, such as blocks and identifiers or objects and associations. A universal representation allows the implementation of navigation, static checking, and refactoring only once for all languages. Except for TexMo, presented in Sec. 3, we are not aware of any IDE implementing a universal language representation.

The per group and the universal representations are generalizations of the per language representation. Both represent multiple languages in one model. Generally, there are two opposing abstraction mechanisms: *type abstraction* and *word abstraction* [29]. Type abstraction is a unifying abstraction, whereas word abstraction is a simplifying abstraction.

For example, both Java and C# method declarations can include modifiers, but the set of the actual modifiers is language specific. The *synchronized* modifier in Java has no equivalent in C#. Under the type abstraction, Java and C# method declarations can be described by a *Method Declaration* type and an enumeration containing the modifiers. In contrast, under the word abstraction, Java and C# method declarations would be described by a common simple *Method Declaration* type that neglects the modifiers. Obviously, in the type abstraction Java and C# method modifiers are distinguishable, whereas in the more generic word abstraction this information is lost.

Type abstraction is preferable for per group representations. Word abstraction is preferred for universal representations. The choice of abstraction influences the specificity of the representation, affecting the tools. Word abstractions are more generic than type abstractions. For instance, more cross-language refactorings are possible with the per group representation, while the refactorings in the systems relying on the universal representation automatically apply to a wider class of languages.

2.2 Relation Model Types

Software systems are implemented using multiple mograms. At the compilation stage, and often only at runtime, a complete system is composed by relating all the mograms together. Each mogram can refer to, or is referenced by, other mograms. An MLDE should maintain information about these relations. We observe four different techniques to express cross-language relations:

Explicit model. For example, *mega-models* [15], *trace models* [22,9], *relation models* [23], or *macromodels* [24]. All these are models linking distributed mograms together.

Tags. Hypertext systems, particularly HTML code links substructures or other artifacts with each other by tags. Tags define anchors and links within an artifact [10]. Hypertext systems interpret artifacts, anchors, and links. first after interpretation a link is established.

Interfaces. Interfaces are anchors decoupled from artifacts. An interface contains information about a development artifact's contents and corresponding locations. For example, OSGi manifest files or model and meta-model interfaces describe component and artifact relations [13].

Search-based. There is no persistent representation of relations at all. Possible relation targets are established after evaluating a search query. Search-based relations are usually used to navigate in unknown data. For example, in [30] relations across documents in different applications are visualized on user request by searching the contents of all displayed documents.

2.3 Relation Types

Here we elaborate on relations between mograms in different languages. Since we consider only textual languages all the following relation types relate strings.

Free relations are relations between arbitrary strings. They rely solely on human interpretation. For example, natural language text in documentation can be linked to source code blocks highlighting that certain requirements are implemented or that a programmer should read some documentation. Steinberger et. al. describe a visualization tool allowing to interrelate information across domains, even across concrete syntaxes [26]. Their tool visualizes relations between diagrams and data.

Fixed relations: Relations between equal strings are fixed relations. Fixed relations occur frequently in practice. For example, the relation between an HTML anchor declaration and its link is established by equality of a tag's argument names. Figure 1 shows an example of a fixed relation across language boundaries.

Waldner et. al. discuss visualization across applications and documents [30]. Their tool visualizes relations between occurrences of a search term matched in different documents.

String-transformation relations are relations between similar strings, or functionally related strings. For example, a Hibernate configuration file (XML) describes how Java classes are persisted into a relational database. The Hibernate framework requires that a field specified in the XML file has a corresponding get and set method in the Java class. A string `fieldName` in a Hibernate configuration file requires a getter with name `getFieldName` in the corresponding Java class. Depending on the direction, a string-transformation relation either attaches or removes get and capitalizes or decapitalizes `fieldName`.

Domain-Specific Relations (DSRs) are relations with semantics specific to a given domain or project. DSRs are always typed. Additionally, DSRs can be free, fixed or string-transformation relations. For example, a requirements document can **require** a certain implementation artifact, expressing that a certain requirement is implemented. At the same time, some Java code can **require** a properties file, meaning that the code will only produce expected results as soon as certain properties are in place. We consider any relation type hierarchy domain-specific, e.g., trace link classification [22].

The first three relation types, free, fixed, and string-transformation relations are untyped. They are more generic than DSRs, since they only rely on physical properties of relation ends. Fixed, string-transformation, and domain-specific relations can be checked automatically, which allows to implement tools supporting MLSS development, such as error visualization and error resolution.

3 TexMo as an MLDE Prototype

TexMo⁶ addresses the requirements listed in Sec. 2 and it implements an instance of our MLDE taxonomy. TexMo uses a *key-reference* metaphor to express relations. In the example of Fig. 1, the command *declaration* takes the role of a *key* (Fig. 1a) and its uses are *reference* (Fig. 1b). TexMo relations are always many-to-one relations between *references* and *keys*. We summarize how TexMo meets the requirements presented in Sec. 2:

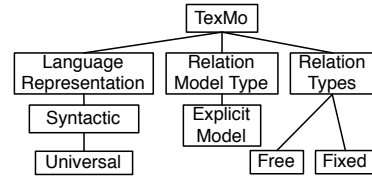


Fig. 3: The feature model instance describing TexMo in our taxonomy of MLDEs.

1. *Visualization.* TexMo highlights keys and references using gray boxes, see line 25 in Fig. 1a and line 8 Fig. 1b. Keys are labeled with a key icon and references are labeled by a book icon; see Fig. 1 left to line numbers. Inspecting markers reveals detailed information, e.g., how many references in which files refer to a key, see Fig. 4b.

⁶ TexMo's source code including the text model and the relation model is available online at: www.itu.dk/~ropf/download/texmo.zip

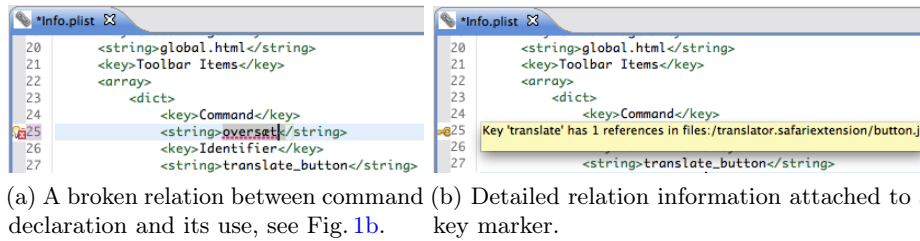


Fig. 4: Visualization and information for inter-language relations.

2. *Navigation.* Users can navigate from any reference to the referred key and from a key to any of its references. Navigation actions are called via the context menu.
3. *Static checking.* Fixed relations in TexMo's relation model (RM) are statically checked. Broken relations, i.e., fixed relations with different string literals as key and reference, are underlined red and labeled by a standard error indicator in the active editor, see Fig. 4a.
4. *Refactoring.* Broken relations can be fixed automatically using quick fixes. TexMo's quick fixes are key centric rename refactorings. Applying a quick fix to a key renames all references to the content of the key. Contrary, applying a quick fix to a reference renames this single reference to the content of the corresponding key.

On top of these multi-language development support mechanisms, TexMo provides syntax highlighting for 75 languages. GPLs like Java, C#, and Ruby, as well as DSLs like HTML, Postscript, etc. are supported. Standard editor mechanisms like undo/redo are implemented, too.

Universal Language Representation. The Text Model. TexMo implements a universal language representation since such an MLDE is easily applicable for development of any MLSS.

All textual languages share a common coarse-grained structure. The text model (Fig. 5), an AST of any textual language, describes blocks containing paragraphs, which are separated by new lines and which contain blocks of words. Words consist of characters and are separated by white-spaces. The only model elements containing characters are word-parts, separators, white-spaces, and line-breaks. Blocks, paragraphs, and word blocks describe the structure of a mogram. Separators are non-letters within a word, e.g., '/', '.', etc., allowing represent of typical programming language tokens as single words.

TexMo treats any mogram as an instance of a textual DSL conforming to Fig. 5. For example, a snippet of JavaScript code `if(event.command ==`, line 8 in Fig. 1b, looks like: `Block(Paragraph[WordBlock(Word[WordPart("if"), SeperatorPart(content:("("), WordPart("event"), SeperatorPart("."), WordPart("command"), WhiteSpace(" ")]), ...])` (using Spoofox [17] AST notation).

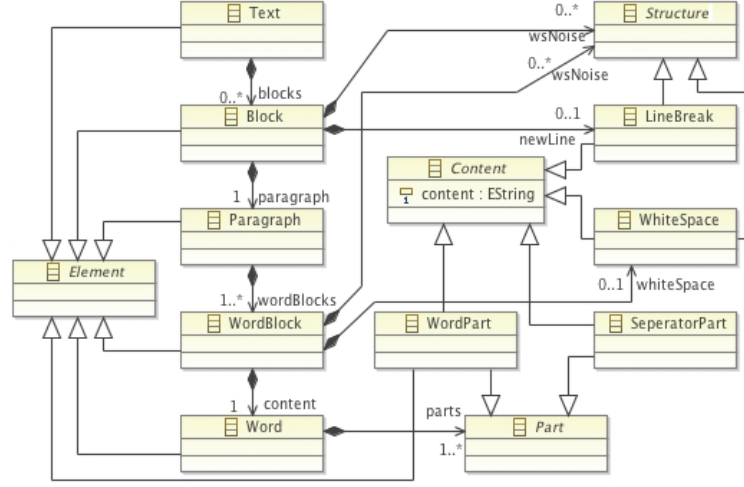


Fig. 5: The open universal model for language representation.

An Explicit Relation Model. TexMo uses an instance of the Relation Model (RM) presented in Fig. 6 to keep track of relations between multi-language mogram code. Our RM allows for relations between mogram contents (**ElementKey** and **ElementReference**), between mogram contents and files (**Artifacts**) or components (**Components**), and between files and components. This allows for example to express relations in case mogram code requires another file, which occurs frequently, e.g., in HTML code.

The RM instance is kept as a textual artifact. The textual concrete syntax is not shown here, since the RM is not intended for human inspection. TexMo automatically updates the RM instance whenever developers modify interrelated mograms. That is, TexMo supports evolution of MLSS. Currently, the RM is created manually. TexMo provides context menu actions to establish relations between keys and references. Future versions of TexMo will integrate pattern based mining mechanism [23,9] to supersede manual RM creation.

Relation Types. TexMo’s RM currently implements fixed and explanatory relations. Explanatory relations are free relations in our taxonomy. Keys and references of fixed relations contain the same string literal. Figure 1 shows a fixed relation and Fig. 4 shows a broken fixed relation. Explanatory relations allow to connect arbitrary text blocks with each other, for example documentation information to implementation code.

4 Evaluation

In this section we discuss TexMo’s applicability. First, we evaluate TexMo’s language representation mechanism, i.e., its representation of mograms as text

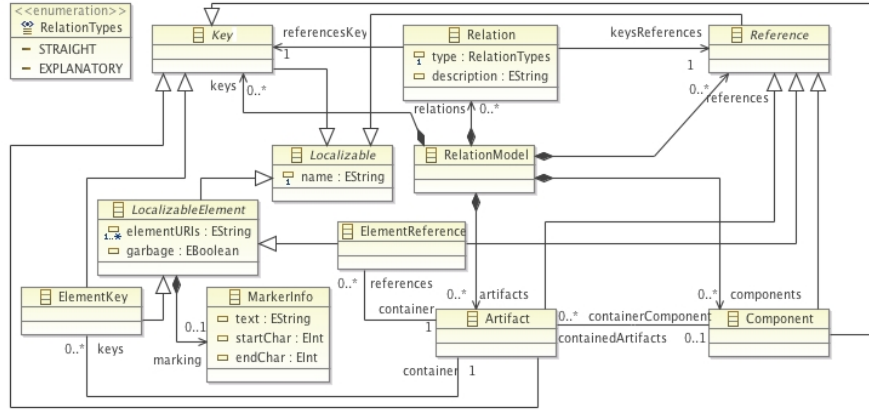


Fig. 6: TexMo's explicit relation model.

models. Second, we provide preliminary evidence on the feasibility of TexMo by testing user acceptance. Furthermore, we discuss applicability of TexMo's relation model with respect to keeping inter-language relations while testers are using TexMo.

The subject used for this evaluation is the open-source web-based bug-tracking system, JTrac. JTrac's code base consists of 374 files. The majority of files, 291, contain source code in Java (141), HTML (65), property files (32), XML (16), JavaScript (8), and 29 other source code files such as Shell scripts, etc. Similar to many web-applications, JTrac implements the model-view-controller (MVC) pattern. This is achieved using popular frameworks: Hibernate (hibernate.org) for OR-Mapping and Wicket (cwiki.apache.org/WICKET/) to couple views and controller code. The remaining 83 files are images and a single jar file. We did not consider these files in our evaluation since they do not contain information in a human processable, textual syntax. Clearly, JTrac is an MLSS.

4.1 Universal Language Representation

To evaluate TexMo's universal language representation, we manually opened all 291 mograms with the TexMo editor to check if a correct text model can be established. By correct we mean that any character and string in a source code artifact has a corresponding model element in the text model, which in turn allows the RM to interrelate mograms in different languages. The files used are available at: www.itu.dk/~ropf/download/jtrac_experiment.zip.

We concluded that all 291 source code files can be opened with the TexMo editor. For all files a correct text model has been established.

4.2 User Test

To test user acceptance, we let 11 testers perform three typical development tasks. The testers included 4 professional developers, 3 PhD students, and 4 undergraduate students, with median 3 years of working experience as software developers.

Using only a short tutorial, which explains TexMo’s features the testers had to work on the JTrac system. First, they had to find and remove a previously injected error, a broken fixed relation. Second, they had to rename a reference and fix the now broken relation. Third, they had to replace a code block, which removes two keys. We captured the screen contents and observed each tester. After task completion, each tester filled out a questionnaire. Questions asked for work experience, proficiency in development of MLSS using Java, HTML, and XML. Additionally, two open questions on the purpose of the test and on the usefulness of TexMo were asked. After the completion of questionnaires we had a short, open discussion about TexMo where we took notes on tester’s opinions.

We conclude that the testers understand and use MLDE concepts. Seven testers applied inter-language navigation to better understand the source code, i.e., to inspect keys and references whenever an error was reported. Furthermore, another seven used rename refactorings to securely evolve cross-language relations in JTrac. All testers were able to find all errors and to fix them. In the following we quote a selection of the testers arguing about usefulness of TexMo (we avoid quoting complete statements for the sake of brevity). Their statements indicate that visualization, static checking, navigation and refactoring across language boundaries are useful and that such features are missing in existing IDEs.

Q: *“Do you think TexMo could be beneficial in software development? Why?”*

A₁: *“TexMo’s concepts are really convincing. I would like to have a tool like this at work.”*

A₂: *“Liked the references part and the checking. Usually, if you change the keys/references you get errors at runtime [which is] kind of late in the process.”*

A₃: *“[TexMo] improves debugging time by keeping track of changes on source code written in different programming languages that are strongly related. I do not know any tool like this.”*

A₄: *“I see [TexMo] useful, especially when many people work on the same project, and, of course, in case the projects gets big.”*

A₅: *“I did development with Spring and a tool like TexMo would solve a lot of problems while coding.”*

A₆: *“In large applications it is difficult to perform renaming or refactoring tasks without automated tracking of references. . . . If there would be such a reference mechanism between JavaScript and C#, it would save us a lot of work.”*

A₇: *“[TexMo] solves [a] common problem experienced when software project involves multiple languages.”*

Robustness of the Relation Model. To run the user test and to demonstrate that the RM can express inter-language relations in an MLSS, we established a RM relating 9 artifacts containing 51 keys, 87 references, via 87 fixed relations with each other. The RM relates code in Java, HTML, and properties files with each

other. We did not aim for a *complete* RM, since we focus on demonstrating TexMo’s general applicability. After the testers had finished their development tasks, we inspected the RMs manually to verify that they still correctly interrelate keys and references.

We conclude that TexMo’s RM is robust to modifications of the MLSS. After modification operations, all relations in the RM correctly relate keys and references across language boundaries.

A common concern of the testers related to replacing a code block containing multiple keys with a new code block, where TexMo complains about a number of created dangling references in corresponding files. We did not implement a feature to automatically infer possible keys out of the newly inserted code, since we consider this process impossible to automate completely.

4.3 Threats to Validity

The code base of JTrac might be too small to allow to generalize that any textual mogram in any language can be represented using TexMo’s text model. However, we think that nearly 300 source code files in 15 languages gives a rather strong indication. The RM used for the user tests might be too small and incomplete. We were not interested in creating a complete RM, but only concerned about its general applicability.

To avoid direct influence on the testers in an oral interview, we used a written questionnaire. All quotes in the paper are taken from this written data.

5 Related Work

Strein et. al. argue that contemporary IDEs do not allow for analysis and refactoring of MLSS and thus are not suitable for development of such. They present X-Develop an MLDE implementing an extensible meta-model [28] used for a syntactic per language group representation. The key difference between X-Develop and TexMo is the language representation. TexMo’s universal language representation allows for its application in development of any MLSS regardless of the used languages. Similarly, the IntelliJ IDEA IDE implements some multi-language development support mechanisms. It provides multi-language refactorings across some exclusive languages, e.g., HTML and CSS. Unlike in TexMo, these inter-language mechanisms are specific to particular languages since IntelliJ IDEA relies on a per language representation.

Some development frameworks provide tools to enhance IDEs. Our evaluation case, JTrac relies on the web framework Wicket. *QWickie* (code.google.com/p/qwickie), an Eclipse plugin, implements navigation and renaming support between inter-related HTML and Java files containing Wicket code. The drawback of framework-specific tools is their limited applicability. QWickie cannot be used for development with other frameworks mixing HTML and Java files.

Chimera [2] provides hypertext functionality for heterogeneous *Software Development Environments (SDE)*. Different programs like text editors, PDF

viewers and browsers form an SDE. These programs are viewers through which developers work on different artifacts. Chimera allows for the definition of anchors on views. Anchors can be interrelated via links into a hyperweb. TexMo is similar in that models of mograms can be regarded as views where each model element can serve as an anchor for a relation. Chimera is not dynamic. It does not automatically evolve anchors while mograms are modified. Subsequent to modifications, Chimera users need to manually reestablish anchors and adapt the links to it. Contrary, TexMo automatically evolves the RM synchronously to modifications applied to mograms. Only after deleting code blocks containing keys, users need to manually update the dangling references.

Meyers [20] discusses integrating tools in multi-view development systems. One can consider language integration as a particular flavor of tool integration. Meyers describes basic tool integration on file system level, where each tool keeps a separate internal data representation. This corresponds to the per language representation in our taxonomy. Meyers' *canonical* representation for tool integration corresponds to our universal language representation. Our work extends Meyers work by identifying a per language group representation.

6 Conclusion & Future Work

We have presented a taxonomy of multi-language development environments, and TexMo, an MLDE prototype implementing a universal language representation, an explicit relation model supporting free and fixed relations. The taxonomy is established by surveying related literature and tools. We have also argued that implementation of TexMo meets its design objectives and evaluated adequacy of its design. By itself TexMo demonstrates that design of useful MLDEs is feasible and welcomed. We reported very positive early user experiences.

To gather further experience, we plan to extend TexMo with string-transformation and domain-specific relations and compare it to an MLDE using a per language representation. We realized that it is costly to keep an explicit RM updated while developers work on a system, especially the larger a RM grows. Therefore, we will experiment with a search-based relation model. This will also overcome the vulnerability of an explicit RM to changes applied to mograms outside the control of the MLDE.

Note, TexMo's RM does not only allow the interrelation of mograms of different languages but also of mograms in a single language. We do not focus on this fact in this paper. However, this ability can be used to enhance and customize static checks and visualizations beyond those provided by current IDEs without extending compilers and other tools.

While working with TexMo we realized that a universal language representation is favorable if an MLDE has to be quickly applied to a wide variety of systems with respect to the variety of used languages. Furthermore, there is a trade-off between the language representation mechanism and the richness of the tools an MLDE can provide. Basic support, like visualization, highlighting, navigation and rename refactorings, can be easily developed on any language

representation, with very wild applicability if the universal representation is used. More complex refactorings require a per group or a per language representation.

In future we plan to build support to automatically infer inter-language relations. Fixed and string-transformation relations can be automatically established by searching for equal or similar strings. This process is not trivial as soon as a language provides for example scoping. Then inferring inter-language relations has to additionally consider language specific scoping rules. Inferring domain-specific relations has to rely on additional knowledge provided by developers, for example as patterns [23], which explicitly encode domain knowledge. Inferring free relations is probably not completely automatable but relying on heuristics and search engines could result in appropriate inter-language relation candidates.

Acknowledgements: We thank Kasper Østerbye, Peter Sestoft and David Christiansen for discussion and feedback on models for representation of language groups and for feedback on the TexMo prototype. EMFText developers have provided technical support during TexMo’s development. Chris Grindstaff has developed the Color Editor (gstaff.org/colorEditor), parts of which were reused for TexMo’s syntax highlighting. Last but not least, we also thank all the testers participating in the experiment.

References

1. Zend Technologies Ltd.: Taking the Pulse of the Developer Community. static.zend.com/topics/zend-developer-pulse-survey-report-0112-EN.pdf, seen: Feb. 2012
2. Anderson, K.M., Taylor, R.N., Whitehead, Jr., E.J.: Chimera: Hypermedia for Heterogeneous Software Development Enviroments. *ACM Trans. Inf. Syst.* 18 (July 2000)
3. Badros, G.J.: JavaML: A Markup Language for Java Source Code. *Comput. Netw.* 33 (June 2000)
4. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In: *Proc. of the IEEE/ACM International Conference on Automated Software Engineering* (2010)
5. Czarnecki, K., Eisenecker, U.W.: *Generative Programming: Methods, Tools, and Applications* (2000)
6. Erlikh, L.: Leveraging Legacy System Dollars for E-Business. *IT Professional* 2 (May 2000)
7. de Figueiredo Carneiro, G., Mendonça, M.G., Magnavita, R.C.: An experimental platform to characterize software comprehension activities supported by visualization. In: *ICSE Companion* (2009)
8. Groenewegen, D.M., Hemel, Z., Visser, E.: Separation of Concerns and Linguistic Integration in WebDSL. *IEEE Software* 27(5) (2010)
9. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-Modelling: From Theory to Practice. In: *Proc. of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I* (2010)
10. Halasz, F.G., Schwartz, M.D.: The Dexter Hypertext Reference Model. *Commun. ACM* 37(2) (1994)
11. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: *Proc. of the 2nd International Conference on Software Language Engineering (SLE 2009), Revised Selected Papers* (2010)

12. Hessellund, A.: SmartEMF: Guidance in Modeling Tools. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (2007)
13. Hessellund, A., Wąsowski, A.: Interfaces and Metainterfaces for Models and Meta-models. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (2008)
14. Holst, W.: Meta: A Universal Meta-Language for Augmenting and Unifying Language Families, Featuring Meta(oopl) for Object-Oriented Programming Languages. In: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (2005)
15. Jouault, F., Vanhooft, B., Brunelie, H., Doux, G., Berbers, Y., Bezivin, J.: Inter-DSL Coordination Support by Combining Megamodeling and Model Weaving. In: Proceedings of the 2010 ACM Symposium on Applied Computing (2010)
16. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (1990)
17. Kats, L.C.L., Visser, E.: The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. In: OOPSLA (2010)
18. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels (2008)
19. Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. Commun. ACM 21 (June 1978)
20. Meyers, S.: Difficulties in Integrating Multiview Development Systems. IEEE Softw. 8 (1991)
21. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: Proc. of the 31st International Conference on Software Engineering (2009)
22. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering. Softw. Syst. Model. 10 (October 2011)
23. Pfeiffer, R.H., Wasowski, A.: Taming the Confusion of Languages. In: Proceedings of the 7th European Conference on Modelling Foundations and Applications (2011)
24. Salay, R., Mylopoulos, J., Easterbrook, S.: Using Macromodels to Manage Collections of Related Models. In: Proc. of the 21st International Conference on Advanced Information Systems Engineering (2009)
25. Standish, T.A.: An Essay on Software Reuse. IEEE Trans. Software Eng. (1984)
26. Steinberger, M., Waldner, M., Streit, M., Lex, A., Schmalstieg, D.: Context-Preserving Visual Links. IEEE Transactions on Visualization and Computer Graphics (InfoVis '11) 17(12) (2011)
27. Strein, D., Kratz, H., Lowe, W.: Cross-Language Program Analysis and Refactoring. In: Proc. of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (2006)
28. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An Extensible Meta-Model for Program Analysis. IEEE Trans. Softw. Eng. 33 (September 2007)
29. Wagner, S., Deissenboeck, F.: Abstractness, Specificity, and Complexity in Software Design. In: Proc. of the 2nd International Workshop on the Role of Abstraction in Software Engineering (2008)
30. Waldner, M., Puff, W., Lex, A., Streit, M., Schmalstieg, D.: Visual Links Across Applications. In: Proc. of Graphics Interface (2010)
31. Xing, Z., Stroulia, E.: Refactoring practice: How it is and how it should be supported — an Eclipse case study. In: Proc. of the 22nd IEEE International Conference on Software Maintenance (2006)

12

Cross-language Support Mechanisms Significantly Aid Software Development – MODELS'12 (Paper E)

Cross-Language Support Mechanisms Significantly Aid Software Development

Rolf-Helge Pfeiffer and Andrzej Wąsowski

IT University of Copenhagen, Denmark
{ropf,wasowski}@itu.dk

Abstract. Contemporary software systems combine many artifacts specified in various modeling and programming languages, domain-specific and general purpose as well. Since multi-language systems are so widespread, working on them calls for tools with cross-language support mechanisms such as (1) visualization, (2) static checking, (3) navigation, and (4) refactoring of cross-language relations. We investigate whether these four mechanisms indeed improve efficiency and quality of development of multi-language systems. We run a controlled experiment in which 22 participants perform typical software evolution tasks on the JTrac web application using a prototype tool implementing these mechanisms. The results speak clearly for integration of cross-language support mechanisms into software development tools, and justify research on automatic inference, manipulation and handling of cross-language relations.

1 Introduction

Developers building contemporary software systems constantly deal with multiple languages at the same time. For example, around one third of developers using the Eclipse IDE work with C/C++, JavaScript, and PHP, and a fifth of them use Python besides Java [1]. PHP developers regularly use one to two languages besides PHP [2]. Developers of large enterprise systems face a particularly complex challenge. For instance, OFBiz, an industrial quality open-source ERP system combines more than 30 languages including General Purpose Languages (GPLs), several XML-based Domain-Specific Languages (DSLs), along with configuration files, property files, and build scripts. ADempiere, another industrial quality ERP system, uses 19 languages. The eCommerce systems Magento and X-Cart utilize more than 10 languages each.¹

We call systems using multiple languages, *Multi-Language Software Systems (MLSSs)*. Obviously, the majority of modern software systems are MLSSs.

To demonstrate how disturbing development of MLSSs is, let's consider an example extracted from JTrac, an open-source, web-based bug-tracking system. JTrac's login page (Fig. 1) is implemented in three source code files in three different languages. The login page itself is described in HTML (Lst. 3), displayed messages are given in a properties file (Lst. 1), and the logic evaluating a login is

¹ See ofbiz.apache.org, adempiere.com, magentocommerce.com, x-cart.com


```

1 login.title = JTrac Login
2 login.home = Home
3 login.loginName = Login Name / email ID
4 login.password = Password
5 login.rememberMe = remember me
6 login.submit = Submit
7 login.error = Bad Credentials

```

Listing 1: A properties file excerpt.

```

1 private class LoginForm extends StatelessForm {
2     private String loginName;
3     private String password;
4     public String getLoginName() {
5         return loginName;
6     }
7     public String getPassword() {
8         return password;
9     }

```

Listing 2: Java login logic excerpt.

```

1 <table class="jtrac">
2 <tr>
3 <td class="label"><wicket:message key="login.loginName" /></td>
4 <td colspan="2"><input wicket:id="loginName" size="35" /></td>
5 </tr>
6 <tr>
7 <td class="label"><wicket:message key="login.password" /></td>
8 <td colspan="2"><input type="password" wicket:id="password" size="20" /></td>
9 <td align="right">
10 <input type="submit" wicket:message="value:login.submit" />
11 </td>
12 </tr>

```

Listing 3: A fragment of the HTML code describing JTrac’s login page.

described in Java (Lst. 2). The HTML code describes the structure of the login page and its contents—how the input fields for login and password insertion are laid out and how they are ordered. Since JTrac is built using the web-development framework *Wicket*, the HTML code contains wicket identifiers, which serve as anchors for string generation or behavior triggering, see lines 3, 4, 7, 8, and 10 in Lst. 3. The properties provide certain messages for the login page. For instance, the property on line 3 in Lst. 1 provides the message string for line 3 of the HTML code. The Java code (Lst. 2) provides authentication logic. Most of this code is not shown here, to conserve space. In order, to correctly invoke the Java code, the field names (lines 2–3), the corresponding get methods (lines 4 and 7), and the set methods (not shown), must use the same name as the wicket identifiers on lines 4 and 8 in Lst. 3.

Now, imagine that a developer renames the string literal `login.loginName` on line 3 in the HTML code to `login.loginID`. Obviously, the relation between the properties file (line 3) and the HTML file is now broken. In effect, the message asking for a login name is not displayed correctly anymore, see Fig. 2. The mistake is only visible at runtime. Observe that such small quiet changes of behavior can easily be missed by testers. Similarly, renaming the string literal `loginName` on line 4 in Lst. 3 to `loginID` breaks a relation to the field `loginName` in the Java file (affects lines 2, 4, and 5 in Lst. 2). The effect of this change is even more serious since JTrac crashes with an error page, see Fig. 3.

We believe that development of MLSSs could be significantly improved if Integrated Development Environments (IDEs) included support for multi-language development, known from single languages, such as (i) visualization (ii) static checking for consistency, (iii) navigation and (iv) refactoring of cross-language relations. In the remainder we refer to these four mechanisms as *Cross-*

Fig. 1: Error-free login page.

Fig. 2: Login page with a broken relation between HTML and property code.

Unexpected RuntimeException

```
WicketMessage: Unable to find component with id 'loginID' in [MarkupContainer [Component ic
[markup = file:/Volumes/Data/JTrac_Experiment/runtime_workspace/jtrac/target/jtrac/WEB-INF
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
<html>
  <head>
    <title wicket:id="title"> </title>
    <link rel="stylesheet" type="text/css" href="resources/jtrac.css"/>
    <link rel="shortcut icon" type="image/x-icon" href="favicon.ico"/>
  </head>
  <body>
    <span wicket:id="individucl"> </span> <br/>
    <table width="100%" class="jtrac alt">
```

Fig. 3: Login page with a broken relation between HTML and Java code.

Language Support (CLS) mechanisms. In this paper, we address the following research question on CLS:

Do Cross-Language Support mechanisms improve developer's understanding of the system and reduce the number of errors made at development time?

To investigate this question we run a controlled experiment in which 22 participants perform typical development and customization tasks on JTrac, a representative MLSS.

It is well known that maintenance and customization of software systems is expensive and time consuming. Between 85% to 90% of project budgets go to legacy system operation and maintenance [5]. Lientz et al. [12] state that 75% to 80% of system and programming resources are used for extensions and maintenance, where alone understanding of the system stands for 50% to 90% percent of these costs [18]. The results of our experiment demonstrate (i) that developers using CLS mechanisms find and fix more errors in a shorter time than those in the control group, (ii) that they perform development tasks on language boundaries more efficiently, and (iii) that even unexperienced developers provided with CLS perform similarly or better than experienced developers in developing MLSSs. Clearly, the integration of CLS into IDEs and development tools would contribute to reducing the high cost of software maintenance and evolution. These results confirm the importance of research on interrelating models and modeling languages, such as trace models [7,14], multi-modeling [8], mega-models [9], macromodels [17], and relation models [15,16]. Additionally, the results motivate research on automatic inference of cross-language relations.

The JTrac system plays a role of the experimental unit in our setup. We use a prototype development editor, TexMo, as the experimental variable, by enabling and disabling its cross-language support. JTrac and TexMo are presented in Sect. 2. Section 3 describes our methodology and the setup of the experiment. We analyze the results in Sect. 4, discuss threats to validity in Sect. 5 and related work in Sect. 6. We conclude in Sect. 7.

Experiment artifacts referred in this paper are available online at itu.dk/people/ropf/download/Experiment.zip. The archive contains TexMo’s source code, the JTrac instance used for the experiment, all documents, questionnaires, answers, and statistics. Screen captures are available on request, as they take up a lot of space.

2 Technical Background

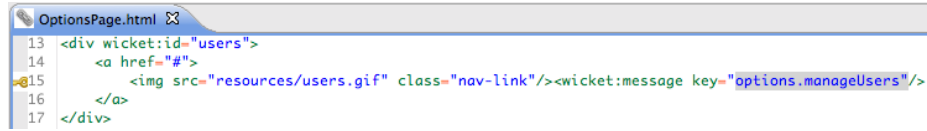
2.1 JTrac: An MLSS Representative

We use the open-source web-based bug-tracking system JTrac as an experimental unit in our experiment. JTrac’s code base contains 374 files of which the majority (291) contain code: Java (141), HTML (65), property files (32), XML (16), JavaScript (8), and 29 other source code files such as Shell scripts, XSLT transformations, etc. The remaining 83 files are images such as “.png”, “.gif” and a single jar file. Most of the property files are used for localization of system messages. The XML files are used for various purposes, for example to give an object-relational mapping describing how to persist business objects. As many other web-applications, JTrac implements the model-view-controller pattern. This is achieved using popular frameworks: Hibernate (hibernate.org) for object-relational mapping and Wicket (wicket.apache.org) to couple views and controller code. Clearly, JTrac is a MLSS.

2.2 TexMo: A Multi-Language Programming Environment

TexMo is a prototype of a Multi-Language Development Environment [16] developed by Pfeiffer. It is an editor that allows to interrelate source code in multiple languages. TexMo uses a *relation* metaphor. Relations are defined between *references* and *keys*. A key is a fragment of code that introduces an identifiable object, a concept, etc. A reference is a location in code that relates to a key. Relations are always many-to-one between *references* and *keys*. TexMo addresses MLSSs development by implementing the CLS mechanisms as follows:

1. *Visualization*. TexMo highlights keys and references in gray. See reference from l.143 in Fig. 4b to l.15 in Fig. 4a. Keys are labeled with a key icon and references are labeled by a book icon; see Fig. 4, left to line numbers. Inspecting markers reveals further details, such as how many references and in which files refer to a key.
2. *Navigation*. Users can access the key from any of its reference and navigate from a key to any of its references. Navigation is activated via a context menu.

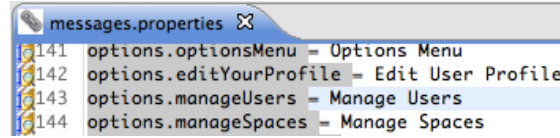


```

13 <div wicket:id="users">
14   <a href="#">
15     <wicket:message key="options.manageUsers"/>
16   </a>
17 </div>

```

(a) HTML code, which fills a message given by a property name.



```

141 options.optionsMenu = Options Menu
142 options.editYourProfile = Edit User Profile
143 options.manageUsers = Manage Users
144 options.manageSpaces = Manage Spaces

```

(b) A properties key options.manageUsers.

Fig. 4: Declaration of a Wicket id and its use.

3. *Static checking.* TexMo statically checks cross-language relations. Broken relations are underlined red and labeled by a standard error indicator, see Fig. 5.
4. *Refactoring.* Broken relations can be fixed automatically by applying quick fixes. TexMo’s quick fixes are key centric rename refactorings. Applying a fix to a key renames all references to the content of the key. Dually, applying a quick fix to a reference renames this single reference to point to its key.

TexMo is an Eclipse plugin. It uses a universal model for representation of any textual language. That is, any source code file is an instance of an EMF-based DSL, which relies on the physical structure of its text. Code is represented as paragraphs, words, parts of words, characters, and special characters like dots or semicolons. This universal representation of source code permits the use of a universal *relation model*, to track relations across different programming artifacts, to link arbitrary information across language boundaries and to synchronize these relations whenever programming artifacts are modified by developers. Further information about TexMo is available in [16].

3 The Experiment

We run a controlled experiment with 22 participants divided into two groups. The control group A performs the tasks using TexMo with CLS disabled. The treatment group B uses TexMo with all four CLS mechanisms enabled.

3.1 Hypotheses

We refine the initial research question into five specific hypotheses:

- H1. *Developers using CLS find and fix more errors than the developers in the control group.* H1 aims to capture the effectiveness of CLS. Since developers get more support by the IDE guiding to problems and offering possible solutions, we expect them to find and fix more errors.

- H2. *Using CLS does not have negative impact on speed of work.* Since CLS provides more information that need to be processed by developers, it could take longer working with CLS than without it (due to information flooding).
- H3. *The least experienced developers using CLS perform better than the most experienced developers in the control group.* Since we expect experienced software developers to perform better than non-experienced developers, it is interesting to investigate how close non-experienced developers can be brought to the quality and performance of experienced ones by just offering CLS. Note, that we refer to general experience in software engineering not experience related to the experimental unit JTrac.
- H4. *Developers using CLS locate errors in source code, whereas developers in the control group identify effects of errors.* We expect developers in the treatment group, those using CLS, to describe errors on a different level of abstraction. They will locate errors, i.e., which code constructs in relation with others are responsible for erroneous behavior, whereas developers in the control group will identify effects of errors, i.e., the erroneous behavior of the system. This would mean that developers using CLS have a deeper understanding of the implementation of the system under development.
- H5. *Developers use CLS mechanisms.* We expect developers offered CLS mechanisms to actually use them voluntarily.

3.2 Experiment Design

We use the terminology of Juristo and Moreno [10, Chpt. 4.2] in our description.

The Experimental Unit. JTrac is a representative of a MLSS. It uses more than 5 languages and with its size of nearly 300 source code files it is sufficiently large to not be easily understandable by the experiment subjects within the given time.

The Experimental Variable. We used TexMo as an IDE with CLS. We are not aware of any other tool supporting the four CLS mechanisms simultaneously. Other existing tools either only provide CLS for particular pairs of languages like IntelliJ IDEA (jetbrains.com/idea), are no longer available, like X-Develop [20], or they do not implement all four mechanisms simultaneously. Also, since TexMo is an Eclipse extension it allows the participants to work in a familiar environment.

Factors. We follow a single-factor with two alternatives experiment design. The factor alternatives are TexMo with visualization, navigation, static checking and refactoring of cross-language relations disabled and the full-featured TexMo as described in Sect. 2. Group B uses the full-featured TexMo and the control group, Group A, uses the restricted TexMo. The latter simulates using a modern IDE.

The Response Variables. We have four response variables representing all quantitative outcomes: number of found errors, number of fixed errors, and the times for finding and fixing errors.

The Pre-Experiment. Before the actual experiment we ran a pre-experiment with five participants, three using the full-featured TexMo editor and two using the control group version. The purpose of the pre-experiment was to check if the experiment tutorials, task descriptions, and objects are consistent, correct and can be understood. In response to the results of the pre-experiment we have fixed incorrect file paths, typos, and wrong line numbers in the task document, and we improved error markers in the TexMo editor. The participants of the pre-experiment have not been used in the main experiment to avoid learning effects. The results of the pre-experiment are not included in the statistics below.

The Pre-Questionnaire. To avoid bias in the distribution of participants in two groups with similar technical experience, we let everyone answer a short questionnaire prior to the actual experiment. We asked 13 yes-no questions about the technical experience of participants: did they develop web-applications before and whether they know and used the web-application frameworks Wicket (wicket.apache.org) or Spring (springsource.org), the object relational mappers Cayenne (cayenne.apache.org) or Hibernate (hibernate.org), the IDEs VisualStudio (microsoft.com/visualstudio) or Eclipse (eclipse.org).

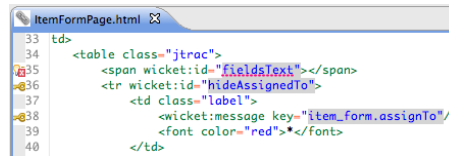
Only Wicket, Hibernate, and Eclipse are used in the experiment but we asked for alternative technologies to minimize the risk that a participant tries to learn about an important technology before the actual experiment.

The Experimental Subject. This experiment is conducted with 22 experimental subjects falling into four major categories: software professionals along with PhD, MSc, and undergraduate students at The IT University of Copenhagen.

The youngest participant is 18 and the oldest is 48, average age is around 29 years, median 28. Nineteen participants report that they have been working as professional software engineers for at least half a year, with maximum of 13 years (average work experience: around 3 years, median 3 years). Two PhD and one graduate student have no experience as professional software engineers.

We distributed the subjects in two groups, one per factor alternative. The distribution was solely based on technological experience reported in a pre-questionnaire, described above. From the 22 participants, 19 reported to have experience with web-application development, 1 already used Wicket, 5 used Hibernate, and 20 have experiences using the Eclipse IDE. Participants were assigned randomly to distribute them equally according to their experience. In Group A, 10 persons have experience developing web-applications, none of them used Wicket before, 2 of them used Hibernate, and 9 used the Eclipse IDE. Similarly, in Group B, 9 persons have developed web-applications before, 1 of them used Wicket, 3 of them Hibernate, and 9 of them Eclipse.

The demographic characteristics of the sample were established using a post-questionnaire (see below). Group A's average age is 29 years, with a median of 28, average work experience is 3.67 years with a median of 3 years. For Group B the age average is 28.64 years with a median of 30, and average work experience is 3.22 years with a median of 3 years.

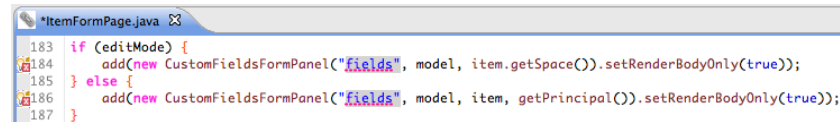


```

33  td>
34      <table class="jtrac">
35          <span wicket:id="fieldsText"></span>
36          <tr wicket:id="hideAssignedTo">
37              <td class="label">
38                  <wicket:message key="item_form.assignedTo"/>
39                  <font color="red">*</font>
40              </td>

```

(a) Declaration of the fieldsText id attached to a span tag.



```

183  if (editMode) {
184      add(new CustomFieldsFormPanel("fields", model, item.getSpace()).setRenderBodyOnly(true));
185  } else {
186      add(new CustomFieldsFormPanel("fields", model, item.getPrincipal()).setRenderBodyOnly(true));
187  }

```

(b) Java code that fills a panel to the span HTML element

Fig. 5: Declaration of a Wicket id and its use.

The Tutorials. At the beginning of the experiment each participant received a tutorial explaining how to compile, start, and stop JTrac. Group B received an extended version explaining the cross-language mechanisms of TexMo. To reduce bias, all features are described using an example in a different domain than the one used for the experiment—the development of a Safari browser extension.

The Tasks. The subjects were asked to perform three tasks representing typical development and customizations tasks. Each task had to be completed, including a brief per task questionnaire, within the 10 minutes. After 8 minutes the participant was reminded that only two minutes were left. After 10 minutes the participants were asked to proceed to the next task. We recorded screen contents of subjects solving the tasks.

Task 1. The participants received an instance of JTrac in which a cross-language relation was broken. Figure 5 shows the error: we renamed fields to fieldsText, the wicket:id attribute in a span tag of the HTML code in line 35. This string literal serves as a key for two references in the corresponding Java code. The renaming leads to a runtime error whenever a new issue report is added to the system.

The participants were asked to locate the error in the source code, name all files which contribute to the error, and to fix the error. The error can be fixed by renaming the key fieldsText to fields or conversely by renaming the references from fields to fieldsText. We considered both solutions as valid fixes.

Task 2. We asked to rename the property options.manageUsers in line 143 Fig. 4b to options.manageAllUsers. This renaming breaks a cross-language relation between a properties file and HTML code. The system will still run error free but a message next to an icon on JTrac’s administration page is not displayed anymore.

The participants were asked to name all files contributing to the newly introduced error and to fix the error. We recognize both renaming options.manageUsers to options.manageAllUsers in the HTML code and reverting the change applied to the properties file as valid solutions.



Fig. 6: Declaration of a Wicket id and its use.

```

1 <tr>
2   <td class="label"><wicket:message key="login.loginName"/></td>
3   <td colspan="2"><input wicket:id="loginName" size="35"/></td>
4 </tr>

```

Listing 4: HTML code replacing lines 20 to 23 in Fig. 6.

Task 3. The participants were asked to replace a block of code. Figure 6 shows the HTML code of JTrac’s login page. Lines 20–23 implement a table row displaying labeled input fields. Line 21 contains a key `login.loginName` and line 22 contains another key `loginName`. A property file providing the text labels refers the `login.loginName`. The `loginName` key is referred from a Java class that evaluates user’s input.

The participants were asked to replace the code block in lines 20–23 with the HTML code given in Fig. 4. Replacing this block removes two keys and breaks several references across three files in different languages. We asked the participants to name all files containing dangling references and to explain how to fix the problem.

The Post-Questionnaire. The post-questionnaire gathered both qualitative and quantitative data, mostly about the demographics: age, length of professional experience, size of developed systems, experience in web-development, familiarity with IDEs, whether they tried to learn technologies mentioned in the pre-questionnaire. Some of the questions overlapped with the pre-questionnaire, to verify consistency, or to check for temporal changes. We also asked for participant experienced problems working with TexMo, and whether TexMo could be beneficial for software development, to collect feedback about our tooling.

4 Results

H1. Developers using CLS find and fix more errors than the developers in the control group. We distinguish between *locating* an error and observing its *effect*. A participant locates an error if she properly names all files contributing to an error and navigates to corresponding lines within the code. She only observes the effect of an error if she runs the application and identifies erroneous behavior.

The results are summarized in Tab. 1. All developers in Group B successfully locate errors in all tasks. Only one developer in Group A locates the error in Task 1, five locate the error introduced in Task 2, and none is able to locate the errors in Task 3. Four developers in Task 1 and five developers in Task 3 managed to partly locate errors, indicating some files contributing to an error but not all.

	Task 1		Task 2		Task 3		Average	
	A	B	A	B	A	B	A	B
error located	9.09%	100%	45.45%	100%	0%	100%	18.18%	100%
error effect located	45.45%	n/a	36.36%	n/a	90.9%	n/a	57.57%	n/a
error fixed	0%	100%	45.45%	100%	qualitative		22.72%	100%

Table 1: Success rate per task (n/a=not applicable). Each group has 11 members.

Tasks 1 and 2 ask the participants to fix the errors. In Task 3 the participants explain how to fix the problem. This is why Tab. 1 contains no success rates for fixing errors for Task 3 (the last row). All members of Group B fix the error in Task 1, compared to none in Group A. In Tasks 2 and 3, a substantially larger fraction of participants fixes the errors in Group B than in Group A. On average Group B is around five times more effective in locating errors than Group A and nearly four times better in fixing errors than Group A. We conclude that CLS significantly improves effectiveness of locating and fixing errors.

H2. Using CLS does not have negative impact on speed of work. We measure the time to locate errors (Group B) or observe effects of errors (Group A) and the subsequent times to fix identified errors (both groups). The results per task are illustrated in Fig. 7. We only report time for participants completing a task, at least partly within the given time.

Group B finds and fixes errors faster than Group A, in Tasks 1 and 2. For Task 3 Group A is slightly faster than Group B. But remember that we give the time to observe an error’s effect for Group A and the time to locate an error for Group B. To fix the the members of the control group would still need to locate it.

In Task 1 (column 1 in Fig. 7) only six participants in the control group locate the error and none of them is able to fix it. Consequently, there is no corresponding box-plot in the second column of Fig. 7. In Task 2, only five participants in Group A succeed to locate and fix the error. For Task 3 ten Group A participants locate the error. All eleven participants in Group B locate and fix all the errors in all tasks (100% success rate). For Task 3, Group A members are slightly faster. This is because the observable error effect appears directly on the login page and is easy to find. Still, members of Group A are not able to find all files contributing to the error, see Tab. 1.

Since Group B is always similarly fast (Task 3) or faster (Tasks 1 and 2) than Group A, we conclude that CLS does not have negative impact on effectiveness.

H3. The least experienced developers using CLS perform better than the most experienced developers in the control group. We ordered participants in both groups based on age, professional experience, experience in engineering of large software systems and web-applications, and the size of developed systems. In our sample, high experience correlates with age, work experience, experience in development of large systems, and with the sizes of systems developed. We

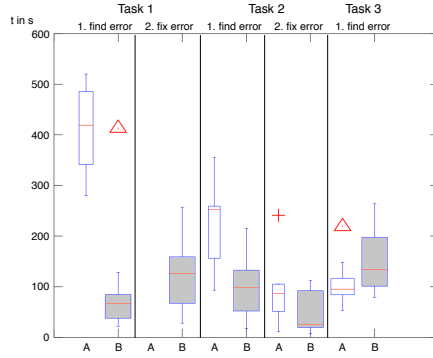


Fig. 7: Time to find and fix errors per group and task in seconds.

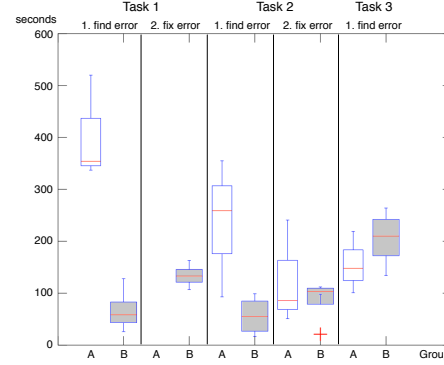


Fig. 8: Time to find and fix errors for the most experienced third of Group A and the least experienced in Group B.

compare the four most experienced developers in Group A with the four least experienced in Group B. Figure 8 illustrates the time used per task. We give the time until a participant observed the effect of an error for Group A and the time to locate an error for Group B. Only three of the selected four members in Group A contribute data to the analysis, since one of the participants did not finish the tasks within the allotted ten minutes.

Clearly, the least experienced members of Group B are faster in locating errors than the most experienced members of the control group, in Tasks 1 and 2. Again, in Task 3 the error is easily observable directly on the login page. Group A members are slightly faster in finding the effect but do not find all files contributing to the error.

For errors which are not easily observable, developers exploiting CLS are faster in finding and fixing errors than developers without them, despite disadvantageous difference in reported experience.

H4. Developers using CLS locate errors in source code, whereas developers in the control group identify effects of errors. Members of Group B always locate errors successfully, see Tab. 1. Only one participant in Group B decided to start JTrac but did not even look at it. Significantly less members of Group A locate errors. Usually, they do observe the effects, and only subsequently they search for error locations if time is left. They rely on text search within the code base for locating the errors.

Members of Group B reason right from the beginning about abstract structures of the implementation, rather than merely observing effects of errors. This increases their effectiveness as indicated by the following quote from the post-questionnaire: *I liked the references part and the checking. Usually, if you change the keys/references you get errors at runtime which is kind of late in the process.* At the same time members of Group A are often not aware, that errors are caused by broken cross-language relations, as seen from their task notes. They either

	Task 1	Task 2	Task 3
read markers	100%	100%	100%
used navigation	63.63%	72.72%	18.18%
used refactoring	63.63%	45.45%	qualitative

Table 2: Rate of Group B participants using CLS per task.

do not locate the error, or admit that they do not know the reason, or simply repeat the error message from the running system. Clearly, members of Group B work on a higher cognitive level than members of Group A.

H5. Developers use CLS mechanisms. All participants in Group B actually use visualizations. They actively investigate error markers by hovering the mouse cursor over them to get more detailed error descriptions. Over 50% of participants for Tasks 1–2 use navigation and automatic refactoring, see Tab. 2. Most do not use navigation, when replacing a code block, since they just deleted the keys, which they could use as navigation start points. Those participants who used navigation did so by undoing the changes and calling navigation from the old keys. This indicates need for new user interface design that would allow accessing deleted relations in a natural manner. No participants used automatic refactoring for Task 3, since TexMo does not implement automatic inference of possible keys out of the newly inserted code.

Furthermore, members of Group A complain that there is no static checking for the errors created when breaking cross-language relations. They expect this feature from an IDE searching for error markers or warnings. It is *difficult to identify the errors [and] . . . to navigate through the source code structure*. Contrary, Group B members not only do use CLS mechanisms, but also admit that *[TexMo] solves [a] commonly experienced problem when software project involves multiple languages*. These results strengthen our believe that higher speed and success rate in Group B is not accidental, but indeed caused by the availability of CLS mechanisms in their version of TexMo.

5 Threats to Validity

To ensure that the results and conclusions in Sect. 4 are statistically sound, we test hypotheses H1 to H3 statistically. Hypothesis H4 relies on qualitative data and hypothesis H5 only observes behavior of Group B, the treatment group. We apply a χ -test to sample data for hypothesis H1 and Student’s t-test to sample data of hypotheses H2 and H3. The effective null-hypothesis for every test is that there is no difference between the experimental factor’s alternatives ($\mu_A = \mu_B$), so CLS mechanisms do not aid software developers measurably.

We reject the null-hypothesis for H1, as all p -values are below significance level (0.05), meaning that for all tasks the alternative providing CLS has a significant

impact on developers. For Tasks 1–3 developers in the treatment group perform significantly better than in the control group.

Testing H2 and H3, results in a statistically significant performance gain for the treatment group to locate errors, except if the errors are easily observable. However, performance is not statistically significantly better for fixing errors if we apply the test to the part of the control group that succeeded (no time to fix the error is available for the subjects who failed). Applying the t-test assuming time larger than 10 minutes for those participants who did not complete the tasks, confirms a significant performance improvement when fixing errors using CLS in Tasks 1–2.

All statistical test data is available in the online appendix.

Internal Threats to Validity. The extended tutorial, explaining TexMo’s features, might have caused a learning effect on members of Group B. They might have been more aware of cross-language relations. We believe that these effects are sufficiently minimized by choice of an example from a completely different domain. Also we assumed that in a standard development scenario, the developers would be aware of CLS support, either through reading manuals or by observing user interface visualizations. A tutorial might have helped them to use them faster in the beginning, which is justified within a frame of a short experiment task. Undoubtedly, they would be able to use the CLS mechanisms even more fluently, if they applied them in a daily work.

Arguably, the sample sizes for H1 tests are very small, while the χ -test is best applied for larger frequencies [10]. We used it, mostly to get a feeling for the data and to give an indication for a trend. Extending the experiments with more participants will have to prove this trend. Similarly, sample sizes pose a threat to validity when testing H2 and H3 with t-tests; in particular, testing H3 where three data points of Group A are compared to four data points of Group B is questionable. Note though, that comparing to similar experiments in related work [21,19] our sample size is large. Indeed this is the largest controlled experiment about CLS mechanisms, that we are aware of.

It can be questioned if times for locating errors (Group B) are at all comparable with times for just observing their effects (Group A). We believe that this is not a problem since for the control group the time to observe the effect is a lower bound for the time to locate an error. So we compare an optimistic under approximation with complete time, and Group B still performs favorably.

External Threats to Validity. We ran a blind experiment. We tried to minimize bias of the participants by relying on written questionnaires and provided only minimal help on request. Typical help was to point the participants to the appropriate Ant task to compile and run JTrac.

There is a risk that participants could have learned about technologies after answering the pre-questionnaire. In the post-questionnaire we re-evaluate the known technologies and note that only four participants learned about a previously unknown technology. Two of them studied Cayenne and Spring respectively, which poses no threat as they are not used in the experiment. Another two learned

about Wicket and Hibernate. Since they fall in two separate groups we do not think that this poses a threat to our grouping.

If our subjects were JTrac experts, they would be able to apply the fixes faster and the disparity would likely be smaller. However, the task of changing unknown code is a common scenario, so the results are valuable.

The factor alternative for control group, with disabled CLS, is not a plain Eclipse. TexMo does not implement all features of Eclipse editors. In particular it does not implement all the keyboard shortcuts. To allow for comparability of results we decided to use the restricted TexMo in the control group, so that the same functionality is available to both groups (besides CLS). We do not think that this has a significant impact on the results. TexMo does provide syntax highlighting and redo/undo support. We believe that industrial strength implementation of CLS mechanisms in would only improve the already promising results of this experiment.

We established the cross-language relation model for JTrac manually. It relates 9 artifacts containing 51 keys, 87 references, via 87 relations with each other. Our model does not contain false positives, which could have been the case, if it was established automatically.

6 Related Work

Mens et al. [13] identify support for multi-language systems as a major challenge for software evolution. They postulate investigating techniques that are as language independent as possible and providing real-world validation and case studies on industrial software systems as valuable.

Chimera [3] provides hypertext functionality for heterogeneous Software Development Environments (SDE). It allows for the definition of anchors that can be interrelated via links into a hyperweb. Chimera supports navigation along the links. The authors claim that developers in an industrial context appreciate using such links while working. Our paper confirms this belief through a controlled experiment, not provided in [3].

Others agree [11], that multi-language systems pose a real problem in maintenance and evolution. The authors of [11] focus on the process of understanding of such systems, trying to improve it with a graph-based query mechanism to find and understand cross-language relations. Their tool is used in industry, but no empirical data on its effectiveness is available. Our experiment results indicate that these techniques are likely very effective, too.

SourceMiner [6] is an IDE providing advanced software visualizations such as tree maps to aid program understanding. The paper does not present any empirical data. It would be interesting to combine SourceMiner with TexMo to measure if these visualizations improve development of MLSSs beyond the CLS mechanisms studied here.

Since the experimental unit JTrac is based on Wicket. We could have chosen *QWickie* (code.google.com/p/qwickie) as a factor alternative. QWickie is an Eclipse plugin, implementing navigation and renaming support between inter-related

HTML and Java files containing Wicket code. We favored TexMo, since we wanted to allow for rerunning the experiment on other experimental units. TexMo is not bound to a particular framework like Wicket.

Visualization mechanisms for relations across heterogeneous concrete syntaxes are studied in the Human-Computer Interaction community. In [21,19] relations across documents in different applications are visualized by links on user request. Visual links are lines crossing application windows. Waldner et al. [21] study if visualization of links between related information in several browser windows is beneficial for understanding scattered information. They run an informal user evaluation with seven participants concluding that *Visual links prevent the user from having to search information manually ... thereby limiting the error probability induced by overseeing information and the effort for the user.* A similar but more formal experiment with 18 participants on visual links is reported by Steinberger et al. [19]. They argue that visual search across different views is a typical task of knowledge workers, which has to be supported by tools. They demonstrate that context preserving visual links are beneficial when searching for interrelated information. Our experiment confirms usefulness of explicit visualization, even though TexMo uses a different visualization scheme.

Chen and coauthors [4] name modern MLSSs, such as Hibernate and Spring “polyglot frameworks”. They implement rename refactorings between Java source code and XML configuration files. Unfortunately, they do not provide any experimental data confirming usefulness of such refactorings. Our experiment shows that a substantial amount of developers use such refactorings when provided.

7 Concluding Remarks & Future Work

In this paper we report a controlled experiment evaluating cross-language support mechanisms. The result is, that visualization, static checking, navigation, and refactoring when offered across language boundaries are highly beneficial. CLS mechanisms perceptibly improve effectiveness of developers working on MLSS. Users of CLS are more effective than the control group with respect to both error rate and productivity (working speed). Furthermore, we show that CLSs are actually used by developers and that they improve understanding of complex, unknown multi-language source code.

In future, we plan to replicate our experiment on larger samples to increase confidence in the presented results. Furthermore, we plan to enhance TexMo with more CLS mechanisms, in particular with more elaborate cross-language refactorings, in order to be able to evaluate a broader range of support functions. Ultimately, the present and future experiments will direct our efforts on developing a new generation of development environments.

References

1. THE OPEN SOURCE DEVELOPER REPORT – 2010 Eclipse Community Survey. eclipse.org/org/press-release/20100604_survey2010.php, seen: Mar. 2012

2. Zend Technologies Ltd.: Taking the Pulse of the Developer Community. static.zend.com/topics/zend-developer-pulse-survey-report-0112-EN.pdf, seen: Feb. 2012
3. Anderson, K.M., Taylor, R.N., Whitehead, Jr., E.J.: Chimera: Hypermedia for Heterogeneous Software Development Enviroments. *ACM Trans. Inf. Syst.* 18 (July 2000)
4. Chen, N., Johnson, R.: Toward Refactoring in a Polyglot World: Extending Automated Refactoring Support across Java and XML. In: *Proceedings of the 2nd Workshop on Refactoring Tools* (2008)
5. Erlikh, L.: Leveraging Legacy System Dollars for E-Business. *IT Professional* 2 (May 2000)
6. de Figueiredo Carneiro, G., Mendonça, M.G., Magnavita, R.C.: An experimental platform to characterize software comprehension activities supported by visualization. In: *ICSE Companion* (2009)
7. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-Modelling: From Theory to Practice. In: *Proc. of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I* (2010)
8. Hesselund, A.: Domain-Specific Multimodeling. Ph.D. thesis, IT University of Copenhagen (2009)
9. Jouault, F., Vanhooff, B., Bruneliere, H., Doux, G., Berbers, Y., Bezivin, J.: Inter-DSL Coordination Support by Combining Megamodeling and Model Weaving. In: *Proceedings of the 2010 ACM Symposium on Applied Computing* (2010)
10. Juzgado, N.J., Moreno, A.M.: *Basics of software engineering experimentation*. Kluwer (2001)
11. Kullbach, B., Winter, A., Dahm, P., Ebert, J.: Program Comprehension in Multi-Language Systems. In: *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)* (1998)
12. Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. *Commun. ACM* 21 (June 1978)
13. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: *Proceedings of the Eighth International Workshop on Principles of Software Evolution. IWPSE '05* (2005)
14. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering. *Softw. Syst. Model.* 10 (October 2011)
15. Pfeiffer, R.H., Wasowski, A.: Taming the Confusion of Languages. In: *Proceedings of the 7th European Conference on Modelling Foundations and Applications* (2011)
16. Pfeiffer, R.H., Wasowski, A.: TexMo: A Multi-Language Development Environment (2012), under submission, www.itu.dk/~ropf/download/texmo.pdf
17. Salay, R., Mylopoulos, J., Easterbrook, S.: Using Macromodels to Manage Collections of Related Models. In: *Proc. of the 21st International Conference on Advanced Information Systems Engineering* (2009)
18. Standish, T.A.: An Essay on Software Reuse. *IEEE Trans. Software Eng.* (1984)
19. Steinberger, M., Waldner, M., Streit, M., Lex, A., Schmalstieg, D.: Context-Preserving Visual Links. *IEEE Transactions on Visualization and Computer Graphics (InfoVis '11)* 17(12) (2011)
20. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An Extensible Meta-Model for Program Analysis. *IEEE Trans. Softw. Eng.* 33 (September 2007)
21. Waldner, M., Puff, W., Lex, A., Streit, M., Schmalstieg, D.: Visual Links Across Applications. In: *Proc. of Graphics Interface* (2010)

13

The Design Space of Multi-language Development Environments – SoSyM'13 (Paper F)

The Design Space of Multi-language Development Environments

Rolf-Helge Pfeiffer and Andrzej Wąsowski

IT University of Copenhagen, Process and System Models Group
`{ropf,wasowski}@itu.dk`

Abstract. Non-trivial software systems integrate many artifacts expressed in multiple modeling and programming languages. However, even though these artifacts heavily depend on each other, existing development environments do not sufficiently support handling relations between artifacts in different languages.

By means of a literature survey, tool prototyping and experiments, we study the design space of multi-language development environments (MLDEs)—tools that consider cross-language relations as first artifacts. We ask: what is the state of the art in the MLDE space? What are the design choices and challenges faced by tool builders? To what extent are MLDEs desired by users, and what aspects of MLDEs are particularly helpful?

Our main conclusions are that (a) cross-language relations are ubiquitous and troublesome in multi-language systems, (b) users highly appreciate cross-language support mechanisms of MLDEs and (c) generic MLDEs clearly advance the state of the art in tooling for language integration. The technical artifacts resulting from this study include a feature model of the MLDE design space, a data set of harvested cross-language relations in a case study system (JTrac) and two MLDE prototypes, TexMo and Coral, that implement two radically different choices in the design space.

1 Introduction

Contemporary software systems are implemented using multiple programming and modeling languages. Today, even simple applications employ more than one language. For instance, PHP developers tend to use a language or two besides PHP itself [1], or around one third of developers using the Eclipse IDE work with C/C++, JavaScript, and PHP besides Java and a fifth of them use Python besides Java [2]. For large enterprise systems the number of languages can be measured in dozens. The Apache Open For Business (OFBiz)¹, an industrial quality open-source ERP system, integrates artifacts in more than 30 languages, including general-purpose languages (GPLs), several XML-based domain-specific languages (DSLs), configuration files, properties files and build scripts. A competing ERP project,

¹ <http://ofbiz.org>, see also [47] on use of DSLs in OFBiz.

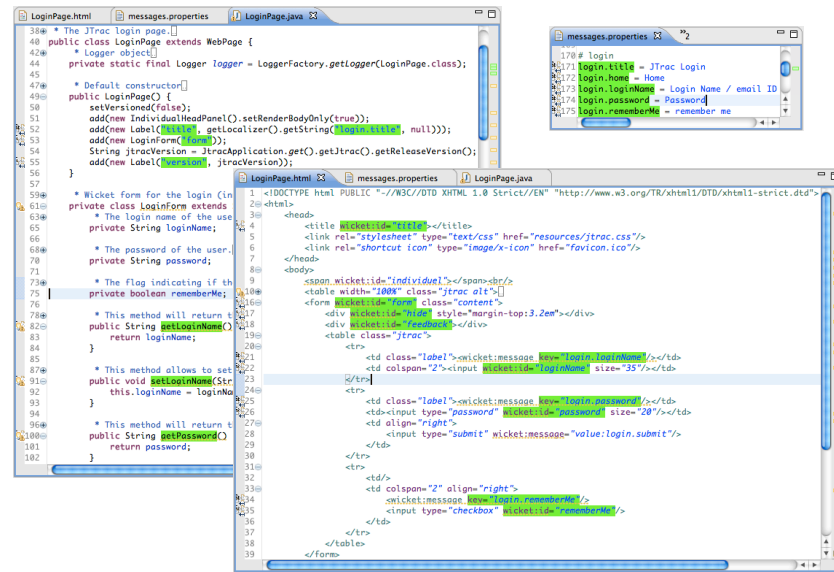


Fig. 1: Mograms in three languages describing JTrac’s login page shown in Coral user interface

ADempiere², uses 19 languages. The eCommerce systems Magento³ and X-Cart⁴ utilize more than 10 languages each. Systems constructed, utilizing the model-driven development paradigm are likely to consist of even more languages: languages for metamodeling (Ecore, KM3⁵ etc.), modeling (DSLs, UML, CVL⁶), validation (OCL, EVL⁷ etc.), model-to-model transformation (QVT, ATL⁸ etc.), code generation (Acceleo⁹ XPand¹⁰ etc.) and scripting (MWE2¹¹ etc.).

There are many good reasons to combine multiple languages into a single system. Domain-specific languages are developed in order to bring the implementation code closer to domain abstractions, to better exploit the knowledge of subject matter experts, and to boost productivity [26]. Usually more than one language is needed, since non-trivial systems span multiple problem domains and multiple technical spaces [46]. Existing domain-specific and general purpose languages are brought into the development in order to reuse existing frameworks,

² <http://www.adempiere.com>

³ <http://www.magentocommerce.com>

⁴ <http://www.x-cart.com>

⁵ <http://wiki.eclipse.org/KM3>

⁶ <http://www.viabilitymodeling.org>

⁷ <http://www.eclipse.org/epsilon/doc/evl/>

⁸ <http://eclipse.org/atl>

⁹ <http://eclipse.org/acceleo>

¹⁰ <http://wiki.eclipse.org/XPand>

¹¹ <http://help.eclipse.org/helios/topic/org.eclipse.xtext.doc/help/MWE2.html>

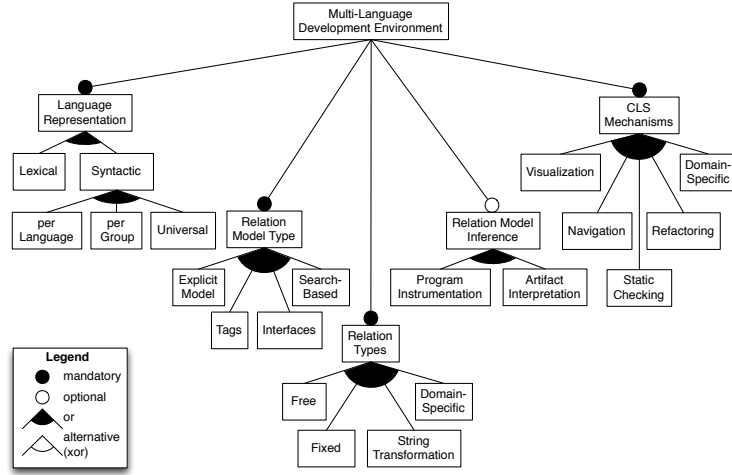


Fig. 2: Taxonomy for multi-language development environments

tools and technology stacks [18]. Moreover, modern systems are rarely standalone and increasingly integrate with other systems, that requires use of interface mechanisms and integration of their languages [65].

The heterogeneity of software systems is thus not accidental, but deliberate, and we expect it to stay. In this paper, we call such heterogeneous composite systems *multi-language (software) systems*. Obviously, as indicated above, the vast majority of modern software systems are multi-language systems.

A typical multi-language system contains many diverse *development artifacts* such as models, source code, properties files, etc. To simplify presentation, we refer to all these as *mograms* [56] in this paper.

Mograms are often heavily interrelated. For example, OFBiz contains hundreds of relations across mograms in different languages [49, 72]. Arguably, relations across language boundaries are fragile. They are broken easily during development, as programming environments do not check them statically, nor do they visualize them. We illustrate the problem with a simple scenario, adapted from [73].

Example. JTrac¹² is an open-source multi-language web-based bug tracking system. JTrac’s login page is implemented using mograms in three different languages. The login page is described in HTML (Fig. 1, bottom). Message strings are stored in a properties file (Fig. 1, top right). The logic is specified in a Java class (top left).

The HTML code specifies the structure of the page and its contents: the actual fields for login and password and their order. Since JTrac is built using the Apache Wicket¹³ web-development framework, the HTML code contains some

¹² <http://www.jtrac.info/>

¹³ <http://wicket.apache.org/>

Wicket identifiers, which allow other mograms to insert strings or behavior at indicated locations. These identifiers can be found in the `LoginPage.html` file, highlighted in lines 4, 16, 17, 18, 22, 26 and 35 in the figure above. The properties file defines the contents of messages on the login page. The Java code provides logic for evaluating a login (authentication). Observe that both the Java code and the properties file refer to the same Wicket identifiers that were used in the HTML file.

Imagine that a developer renames the string literal `login.loginName` in line 21 in Fig. 1 to `login.loginID`. Obviously, the relation between the properties file (l. 173) and the HTML file is now broken, leaving a dangling reference. In effect the message asking for a login is not displayed anymore. Similarly, changing the string literal `loginName` (l. 22 of the HTML file) to `loginID` would break the relation with the `loginName` field of the Java class, affecting lines 82 and 91—Wicket requires existence of accessor methods for its identifiers. This change has a serious effect: JTrac would not function anymore, throwing a runtime exception instead.

Existing Integrated Development Environments (IDE) do not directly support development of multi-language systems. They do not visualize cross-language relations, unlike in Fig. 1, where markers next to line numbers and green highlighting indicate the relations. IDEs lack static checking for consistency of cross-language relations. They cannot offer refactorings encompassing mograms in different languages.

A special class of IDEs, the *Multi-language Development Environments (MLDEs)*, aim at addressing these shortcomings, by providing cross-language support mechanisms (CLS mechanisms). In the past we have built several tools in this space. With this paper we want to document our experience, by exploring the requirements and the design space for MLDEs along three research questions:

1. What is the state of the art in development of MLDEs?
2. What are the design choices and challenges faced by developers (vendors) of MLDEs?
3. To what extent are MLDEs desired by users, and what aspects of MLDEs are particularly helpful?

The paper provides the following contributions:

1. To address the first question we perform a literature survey documenting the main design choices for many MLDEs and related tools (Sect. 2). We summarize the knowledge in a taxonomy of MLDEs, presented as a feature model. The model contains both the defining requirements for MLDEs and the variability in their implementation.
2. To address the second question we provide independent implementations of two radically different instances of the above design space: the Coral and TexMo MLDEs (Sect. 3). These two implementations show the challenges faced by developers of different classes of MLDEs. They also materialize two, so far unavailable, solutions with respect to the design space. We discuss our

experience with both tools, which we gained by applying them to a multi-language case study. We analyze the differences between them qualitatively. We also use the developed tools to harvest a subset of actual cross-language relations in a case study system (JTrac), reporting the density of relations, which clearly cannot be effectively handled without tool support. In this way, we learn storage and performance requirements on MLDEs, caused by size of models and the relations (Sect. 4.1).

3. To address the third question, we approach the communities of users and experts with two experiments addressing the need for, and usefulness of, MLDEs. First, we run an experiment with TexMo, involving developers, who evolve a case study system with and without help of the CLS mechanisms (Sect. 4.2). Second, we survey the community of language developers to evaluate the current practice in language integration (Sect. 4.3).

These technical developments are followed by a discussion of related work (Sect. 5) and conclusion (Sect. 6).

The main conclusions from our case studies and experiments are that (a) cross-language relations are ubiquitous and troublesome in multi-language systems, (b) users highly appreciated cross-language support mechanisms of MLDEs and (c) generic MLDEs like TexMo and Coral can clearly advance the state of the art in tooling for language integration. An important aspect of both TexMo and Coral are that they are generic—they do not depend on any particular languages being related, and thus can be adapted to many frameworks and ecosystems, benefiting not only JTrac, but any multi-language software system. We believe that these conclusions are interesting both for tool builders and for researchers in multi-modeling.

An earlier version of this work appeared in [74]. We also adapt some elements from [73]. In this expanded version, the literature survey has been revised and extended. The implementation of the Coral MLDE, the comparison of Coral with TexMo, and two of the experiments (Sections 4.1 and 4.3) are entirely new.

2 Taxonomy of MLDEs

Programming and modeling languages can hardly be considered in isolation of the system allowing their interpretation—a human mind or a computing system (an interpreter, compiler, data visualizer, etc.). Cross-language relations do not exist in isolation either. They are a manifestation of implicit rules in the underlying interpreting system. We call this underlying set of rules a *framework*. Frameworks could be object-oriented frameworks, but could also be other contexts, as indicated above. Different frameworks give rise to different relations for the same languages.

In the example of Fig. 1, the application server interprets the Java, HTML, and property files. The semantic rules underlying the web-application framework Wicket establish the cross-language relations between the files.

The popular integrated development environments (IDEs), like Eclipse or NetBeans, do not capture these implicit underlying relations and they implement separate editors for every supported language, with separate, isolated syntax

representations. A typical IDE provides separate Java, HTML, and XML editors, even though these editors are used to build systems mixing all these languages. Representing languages separately allows for an easy and modular extension of IDEs to support new programming languages. This easy extensibility has most certainly contributed to the growth and widespread adoption of IDEs [38]. Most, IDE editors maintain an *Abstract Syntax Tree (AST)* in memory and automatically synchronize it with modifications applied to concrete syntax. They exploit the AST to facilitate source code navigation and refactorings, ranging from basic renamings to elaborate code transformations such as *method pull ups*.

Implicit cross-language relations are a major problem in development of multi-language systems, obstructing their modification and evolution [45, 49, 72]. Unlike IDEs, which just integrate development tools, a MLDE integrates different languages by relating programs across language boundaries. This way MLDEs are able to address the challenge of modification and evolution of multi-language systems.

We surveyed IDEs, programming editors¹⁴, and literature to understand the kind of development support they provide. We find that four features, *visualization*, *navigation*, *static checking*, and *refactoring* are implemented by all IDEs and by some programming editors. Consequently, MLDEs should consider delivering these very features across language boundaries as an essential requirement. We call these four features cross-language support mechanisms (CLS mechanisms) [73]:

1. *Visualization* of cross-language relations. Visualizations can range from basic markers, for instance in the style of Fig. 1, to elaborate visualization mechanisms such as treemaps [28].
2. *Navigation* of cross-language relations. Navigation would allow the developer to automatically open either *LoginPage.html* and jump to line 4 or *message.properties* and jump to line 171, when editing *LoginPage.java* on line 52 (Fig. 1). All surveyed IDEs allow to navigate source code. Further, IDEs allow for source code to documentation navigation, which is a basic example of cross-language navigation.
3. *Static Checking* of cross-language relations. As soon as a developer breaks a relation, the error is indicated to show that the system will not run error free. All surveyed IDEs provide static checking by visualizing errors and warnings.
4. *Refactoring* and fixing of broken cross-language relations. Different IDEs implement a different amount of refactorings per language. Particularly, rename refactorings seem to be widely supported in IDEs [64, 93].

¹⁴ We examined the following IDEs/editors: Eclipse <http://www.eclipse.org/>, NetBeans <http://netbeans.org/>, IntelliJ Idea <http://www.jetbrains.com/idea/>, MonoDevelop <http://monodevelop.com/>, XCode <https://developer.apple.com/xcode/>, Ninja IDE <http://ninja-ide.org/>, MacVim <http://macvim.org/>, Emacs <http://aquamacs.org/>, TextWrangler <http://www.barebones.com/products/textwrangler/>, TextMate <http://macromates.com/>, Sublime Text 2 <http://www.sublimetext.com/>, Fraise <https://github.com/jfmoy/Fraise>, Smultron <http://sourceforge.net/projects/smultron/>, Tincta <http://mr-fridge.de/software/tincta/index.php>, jEdit <http://jedit.org/>, Kod <http://kodapp.com/>, gedit <http://projects.gnome.org/gedit/>

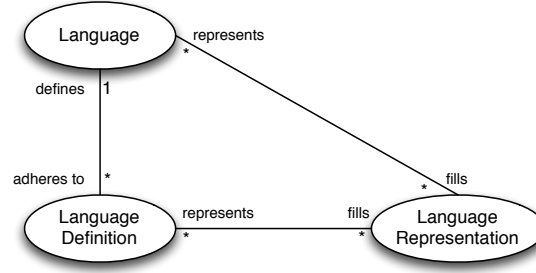


Fig. 3: The concepts of language, language definition, language representation, and their relations

To address the same requirements in an MLDE, in a cross-language fashion, one needs to make three fundamental design decisions:

- How to represent different programming languages?*
- How to relate them?*
- Using what kind of relations?*

Systematizing the answers to these questions led us to a domain model characterizing MLDEs. We present this model in Fig. 2 using the feature modeling notation [20, 54]. The following subsections detail and exemplify the fundamental MLDEs characteristics of our taxonomy. References to the surveyed literature are inlined.

2.1 Language Representation Types

Typically, multi-language systems contain many diverse files such as models, source code, properties files, etc. written in various diverse languages.

Definition 1. *Mograms* are all files that are created, edited, or modified by humans or machines with the purpose to develop, customize, or modify a software system. Such files may contain source code, models, plain text, etc.

In this paper, we use a very broad definition of language.

Definition 2. A textual *language* is a set of sentences. Each *sentence* is a collection of symbols, where *symbols* are usually alphanumerical characters.

Sentences can be fragmented. **Fragments** are just sequences of symbols in a sentence.

We consider any mogram to be a sentence of a language. Note, we believe, that this definition also covers languages with visual concrete syntax. Even if tools present mograms in visual concrete syntax, these artifacts are always persisted in a textual concrete syntax. Consequently, visual concrete syntaxes are only visualizations, i.e., rendered representations, of textual languages.

Definition 3. A *language definition* is a formal way to specify which sentences belong to a language.

Usually, language definitions are given by formal grammars. Here, we consider any computer program that parses mograms as a language definition. Such programs implicitly specify the set of sentences that belong to a language.

In this paper we work with abstractions of languages as we want to work with mograms in different languages generically. So, the central concept to tackle the research questions stated above is **abstraction** of mograms and languages to more abstract representations.

Definition 4. A *language representation* is a data structure specifying the set of abstract concepts of languages and their relations.

A *language representation* is a means to represent sentences of a language. We consider two main types of language representations: *lexical* and *syntactic*. The former represents any mogram of any language as a stream of characters. Whereas, **syntactic language representation**, relies on data structures like trees and graphs to describe concepts and their relations. This work is strongly influenced by the credo “*Everything is a model*” [15]. Often, metamodels are used for specification of syntactic language representations. Abstract syntax trees or metamodels capturing the concepts of a language are examples of syntactic language representations. Syntactic representation can be shared per language, per language group, or universally, as explained in the following.

The concepts *language*, *language definition*, and *language representation* are not independent from each other. Each language has multiple language definitions and multiple language representations. On the other hand, any language definition defines exactly one language, while a language representation may represent many languages. Figure 3 illustrates this ontological disambiguation and the relation of the terms *language*, *language definition*, and *language representation*.

Lexical Representation.

Definition 5. A *lexical language representation*, represents any mogram of any language as a stream of characters.

Most text editors, such as Emacs [81] (without language modes enabled), Vim, and jEdit, implement lexical representations. Mograms are loaded into a buffer in a language agnostic manner. Syntax highlighting is implemented solely based on matching tokens. Similarly, Sufrin et al. [85] formally define commands for text editing separately on top of characters and on top of words and lines. That is, editing commands are formalized on physical properties of a mogram. Editors with lexical language representations provide limited support for static checking, code navigation, and refactoring. This is, due to lack of sufficient information about the edited mogram.

Syntactic Representation. Per Language.

Definition 6. *A syntactic per language representation, represents a single language, which is already defined by another mechanism such as a formal specification, a parser, a metamodel, etc. using data structures like trees or graphs.*

Typical modern IDEs, such as Eclipse or NetBeans, represent mograms in any given language using a separate abstract syntax tree, or a similar richer data structure capturing a mogram’s content. Unlike lexical representation, a structured, typed representation allows for implementation of static checking and navigation within and between mograms of a single language, but not across languages. The advantage of using per language representation, compared to per language group and universal representation, is that modern IDEs are easily extensible to support new languages.

Using models to represent source code is getting more and more popular.¹⁵ This is facilitated by emergence of language workbenches such as EMFText [41], Xtext [27], Spoofax [55], etc.¹⁶ All of these language workbenches rely on models as per language representations.

Also frameworks for refactoring of legacy code exploit per language representations based on models. For example, the MoDisco [17] project, a model-driven framework for software modernization and evolution, represents Java, JSP, and XML source code as EMF models, where each language is represented by its own distinct model. These models are high-level descriptions of an analyzed system and are used for transformation into a new representation. Similarly, the reverse engineering framework BlueAge [14] represents legacy COBOL source code as models, so that model transformations can be employed to modernize legacy COBOL systems. The same principle of abstracting a programming language into an EMF model representation is implemented in JaMoPP [42]. Also, JavaML [13] uses XML for a structural representation of Java source code. On the other hand, SmartEMF [45] translates XML-based DSLs to EMF models and maps them to a Prolog knowledge base. The EMF models realize a per language representation. In our earlier work, we represent OFBiz’ DSLs and Java using EMF models to handle cross-component and cross-language relations [72].

Syntactic Representation. Per Language Group.

Definition 7. *A syntactic per language group representation, represents a group of languages defined by multiple language definitions or represented by multiple per language representations using data structures like trees or graphs.*

A single language representation can represent multiple languages sharing commonalities. Some languages are mixed or embedded into each other, e.g., SQL embedded in C++. Some languages extend others, e.g., AspectJ extends Java.

¹⁵ Language workbenches use modeling technology to represent abstract syntax trees. Therefore, we use the terms AST and model synonymously in this paper, even though this narrows somewhat the traditional meaning of modeling.

¹⁶ See www.languageworkbenches.net for the annual language workbench competition.

Furthermore, some languages are often used together, for instance JavaScript, HTML, XML, and CSS in web development. Using a per language group representation allows increased reuse in implementation of navigation, static checking, and refactoring in MLDEs, because support for each language does not need to be implemented separately.

For example, the IntelliJ IDEA supports code completion for SQL statements embedded as strings in Java code. X-Develop [83,84] implements an extensible model for language group representation to provide refactoring across object-oriented and mark-up languages. AspectJ's compiler generates an abstract syntax tree for Java as well as for AspectJ aspects simultaneously. Similarly, the WebDSL framework represents mograms in its collection of DSLs for web development in a single syntax tree [34]. *Meta*, a language family definition language, allows the grouping of languages by characteristics, e.g., object-oriented languages in *Meta(Oopl)* [50]. The Prolog knowledge base in [45] can be considered as a language group representation for OFBiz' DSLs, used to check for cross-language constraints. The Generic Intermediate Metamodel in [36] is also a per language group representation for models with similar, but changing, metamodels.

Syntactic Representation. Universal.

Definition 8. *A syntactic universal language representation, represents any language defined by any language definition or represented by any language representation using data structures like trees or graphs.*

Universal representations use a single model to capture the structure of mograms in any language. They can represent any version of any language, even of languages not invented yet. Universal representations use simple, but generic, concepts to represent key language concepts, such as blocks and identifiers or objects and associations. A universal representation allows the implementation of navigation, static checking, and refactoring only once for all languages. Research on truly universal language representations is quite scarce as mostly language group representations are suggestive of being universal representations. However, when discussing schemes of tool integration, Meyers [62] mentions the possibility and desirability of a *canonical* representation of mograms. The only IDE (MLDE) implementing a universal language representation known to us is TexMo [74] described in Sect. 3.3.

2.2 Relation Model Types

Software systems are implemented using multiple mograms. At the compilation stage, and often only at runtime, a complete system is composed by relating all the mograms together. Each mogram can refer to, or is referenced by, other mograms. An MLDE should maintain information about these relations. A relation model is a defining feature for MLDEs, that distinguishes them from plain IDEs. We have identified four different techniques to express cross-language relations in MLDEs:

Listing 1.1: An excerpt of an explicit relation model in TexMo.

```

1 RelationModel {
2   Artifact "/jtrac/src/main/java/info/jtrac/wicket/LoginPage.html" {
3     keys 28603127-20aa-41f3-ad36-e6e37849bd10 ...;
4   }
5   ...
6   Artifact "/jtrac/src/main/resources/messages.properties" {
7     references befa04ed-5d54-4183-9dcf-ecd4f378f28d ...;
8   }
9   ...
10  Key "28603127-20aa-41f3-ad36-e6e37849bd10" </jtrac/src/main/java/info/jtrac/wicket/
    LoginPage.html> {
11    ["/@blocks.20/@paragraph/@wordBlocks.2/@content/@parts.3",
12     "/@blocks.20/@paragraph/@wordBlocks.2/@content/@parts.2",
13     "/@blocks.20/@paragraph/@wordBlocks.2/@content/@parts.4"]
14    | "login.loginName" from 905 to 919|
15  }
16  ...
17  Reference "befa04ed-5d54-4183-9dcf-ecd4f378f28d" </jtrac/src/main/resources/messages.
    properties> {
18    ["/@blocks.157/@paragraph/@wordBlocks.0/@content/@parts.1",
19     "/@blocks.157/@paragraph/@wordBlocks.0/@content/@parts.0",
20     "/@blocks.157/@paragraph/@wordBlocks.0/@content/@parts.2"]
21    | "login.loginName" from 5936 to 5950 |
22  }
23  ...
24  Relation 28603127-20aa-41f3-ad36-e6e37849bd10<-befa04ed-5d54-4183-9dcf-ecd4f378f28d[
    FIXED]
25  ...
26 }

```

Explicit Model.

Definition 9. An *explicit relation model* is an artifact, which contains explicit links interrelating fragments of various mograms.

Explicit relation models seem to be the most natural relation representation from a developer's perspective. Alone the survey by Winkler and Pilgrim [92] reports twelve different explicit relation models for capturing traceability information. However, in the following we describe relation models in general, not only trace models. Existing explicit relation models are most often tailored to a particular domain but they share a high degree of commonality. They all express relations by dedicated model elements in separate models linking structures or fragments of mograms.

In different domains and communities, different terminology is used for explicit relation models. The most common names are *megamodels* [18, 53], *trace models* [22, 35, 52, 59, 68, 70], or *macromodels* [77]. Despite their different names, all these models link fragments of distributed mograms together.

Explicit relation models can be seen as graphs whose edges encode relations and whose vertices encode interrelated fragments in mograms. Listing 1.1 illustrates an excerpt of a possible explicit relation model in a textual concrete syntax (as used in TexMo). It shows a relation (line 24) between two fragments of two mograms. Here, the respective fragments are the string literals `login.loginName` on line 21 in HTML and 173 of the properties file in Fig. 1. The fragments

Listing 1.2: An excerpt of a Java class with link tags

```

1 public class LoginPage {
2     private static final Logger logger = ...
3
4     public LoginPage() {
5         setVersioned ( false );
6         add(new IndividualHeadPanel().setRenderBodyOnly(true));
7         add(new Label(@link(in(..../LoginPage.html), target(wicket:title)),
8             getLocalizer().getString("login.title", null)));
9         add(new LoginForm(@link(in(..../LoginPage.html),target(wicket:form))));
10        String jtracVersion = JtracApplication.get().getJtrac().getReleaseVersion();
11        add(new Label("version", jtracVersion));
12    }
13    ...
14 }

```

are identified by uniform resource identifiers (URIs) (lines 11–13 and 18–20 respectively).

Tags. Alternatively, explicit relation models can be represented by tags, similar to HTML link tags. For example, in HTML, link tags can be used to specify relations between fragments of other HTML documents or entire mograms. Such kind of tags are conceivable for non-hypertext systems too.

Definition 10. A *tag-based relation model* marks interrelated fragments directly within heterogeneous mograms. Relations are expressed by link tags, which refer to anchor tags.

Listings 1.2 and 1.3 illustrate a relation model based on tags. The example is based on Fig. 1. The mograms are modified to store anchor tags (**@anchor**) in HTML sources and link tags (**@link**) in the Java sources. Link tags specify relations to the corresponding opposite relation ends marked with anchor tags.

Hypertext systems, link fragments of mograms or complete mograms with each other via tags. For example, in HTML, links are defined by tags [37]. Hypertext

Listing 1.3: An excerpt of HTML code with relation anchor tags

```

1 <html>
2 <head>
3 <title @anchor(wicket:title)></title>
4 <link rel="stylesheet" type="text/css" href="resources/jtrac.css"/>
5 <link rel="shortcut icon" type="image/x-icon" href="favicon.ico"/>
6 </head>
7 <body>
8 ...
9 <form @anchor(wicket:form) class="content">
10 ...
11 </form>
12 ...
13 </body>
14 </html>

```

Listing 1.4: A Tengi interface corresponding to LoginPage.java

```

1 Tengi LoginLogic ENTITY "LoginPage.java" [
2   IN: { loginTitleHTML, loginFormHTML }; CONSTRAINT: loginTitleHTML & loginFormHTML;
3   OUT: { loginTitleJava, loginFormJava}; CONSTRAINT: loginTitleJava & loginFormJava;
4 ]{
5   LOCATOR loginTitleJava IN "LoginPage.java" OFFSET 198 LENGTH 5;
6   LOCATOR loginFormJava design IN "LoginPage.html" OFFSET 278 LENGTH 4;
7 }

```

Listing 1.5: A Tengi interface corresponding to LoginPage.html

```

1 Tengi LoginView ENTITY "LoginPage.html" [
2   IN: { loginTitleJava, loginFormJava}; CONSTRAINT: loginTitleJava & loginFormJava;
3   OUT: { loginTitleHTML, loginFormHTML}; CONSTRAINT: loginTitleHTML & loginFormHTML;
4 ]{
5   LOCATOR loginTitleHTML IN "LoginPage.html" OFFSET 27 LENGTH 17;
6   LOCATOR loginFormHTML design IN "LoginPage.html" OFFSET 244 LENGTH 16;
7 }

```

systems interpret tags within mograms as anchors, and links. After interpretation, a relation is established. HyperPro [66, 69] is a programming environment which treats mograms in a software system as hypertext. That is, mograms can be enriched with tags linking fragments across language boundaries.

DEFT [91], the *Development Environment For Tutorials* relies on tags to specify how different mograms contribute to a document containing a mixture of natural and computer languages constituting a tutorial. In this case, the multi-language system is a document and not a running program.

Reuseware [43, 44], is a composition framework for invasive composition. Components encoding various concerns are defined separately and composed when a system is specified. Both works [43, 44] consider language definitions as components and apply Reuseware to extend languages with certain concepts, such as modularization or aspect-orientation. Reuseware relies on *slots*, *hooks*, and *anchors*, which are all tags defining variation points, i.e., referable fragments, which can be filled or replaced with separately defined fragments.

Kolovos et al. [59] discuss two ways of representing trace links between models. Trace links can either be embedded in the models themselves, e.g., by marking relation ends via tags into the models, or they can be kept as external separate models. The authors propose to use both representations simultaneously and to merge models and trace links from explicit relation models into a tag-based model on user request. The authors reuse UML stereotypes to tag elements in UML models to establish trace links from merged model elements back to their source models.

Interfaces. Relations between fragments of mograms can be explicitly specified in interfaces. Interfaces can be seen as tagged fragments, as in tag-based relation models, which are decoupled from the corresponding mograms.

Listing 1.6: The Wicket library in Coral DSL.

```

1 java { StringReference is org.emftext.language.java.references.impl.StringReferenceImpl ;
2       NamedElementName is org.emftext.language.java.common.NamedElementName; }
3 properties { Key is org.emftext.language.javaproperties.impl.KeyImpl; }
4 html { StringValParameter is html.impl.StringValParameterImpl; }
5
6 string transformation: Key in properties <--> StringValParameter in html with wickedIDslInHTML
7   is info display "A wicketID to property key relation .";
8
9 string transformation: Key in properties <--> StringReference in java with wickedIDslInJava
10  is info display "A wicketID to property key relation .";
11
12 fixed : StringReference :: value in java <--> StringValParameter::value in html with
13       wickedIDslInJavaConstructors
14   is info display "Wicket IDs in Java constructor call .";
15
16 string transformation: NamedElementName in java <--> StringValParameter in html with
17       getterMethods
18   is info display "Wicket IDs require a getter method in Java";
19
20 string transformation: NamedElementName in java <--> StringValParameter in html with
21       setterMethods
22   is info display "Wicket IDs require a setter method in Java";

```

Definition 11. *Interface-based relation models explicitly define fragments and their relations in interfaces. Interfaces are separate artifacts accompanying interrelated mograms.*

Listing 1.4 and Lst. 1.5 illustrates two interfaces for the interrelated Java and HTML mograms of Fig. 1. The interfaces are expressed in the Tengi interface DSL [71]. Tengi interfaces define relation ends in corresponding mograms (ENTITY) as ports (LOCATOR). Out-ports (OUT) specify which relation ends are provided to the environment and in-ports (IN) specify which relation ends are required from the environment. Constraints (CONSTRAINT) specify how mograms are related.

De Alfaro and Henzinger [5] define different kinds of interfaces for component-based software development. Informally, they define an interface model to specify what a components expects from its environment. Based on this work, Hesselund and Wąsowski [49] define interfaces for interrelated models and metamodels to explicitly describe relations between models crossing language boundaries. Compared to the interfaces in [49], OSGi interfaces [61] are more coarse grained. They specify visibility of Java source code organized in packages and other non-source code artifacts, all aggregated in bundles.

Despite their name, Emacs' [81] tags files are actually interfaces. Tag files store a set of tags pointing to mograms or fragments of them. For example, tags point to methods and classes in source code or to chapters and paragraphs in documentation. Tag files do not encode an explicit relation model as relations are established by users navigating on top of tagged information.

Search-based. The three relation models presented so far, directly refer to fragments in mograms. But relations can also be specified indirectly, based on

search queries, which need to be evaluated before relations between concrete fragments can be established. That is, search-based relation models usually do not provide a persistent representation of relations.

Definition 12. *Search-based relation models represent relations between fragments of mograms via queries locating fragments and constraints between the query results, describing the relations themselves. Only after query and constraint evaluation, relation instances are established.*

Listing 1.6 illustrates a search-based relation model. It is expressed in the Coral DSL (see Sect. 3.4), which allows for specification of constraints for cross-language relations. The relation model contains five cross-language relations between Java, HTML, and properties files. The actual constraint is implemented in Groovy. Consider for example the cross-language relation constraint on line 12. It says that a string reference in Java and a parameter in HTML are in relation as soon as their values are identical and the string reference in Java appears in a constructor call.

In search-based relation models, relations between mograms are specified at metalevel. Evaluation of the cross-language relation constraint (line 12) establishes two relations between the fragments `title` (line 52 in Java and line 4 in HTML) and `form` (line 53 in Java and line 16 in HTML) respectively.

Search-based relations are usually used to navigate in unknown data in open systems. For example, in [90] relations across documents in different applications are visualized on user request by searching the contents of all displayed documents. In [23] consistency rules for models in different UML languages are evaluated to find inconsistencies in interrelated models. Hesselund et al. [48] apply code flow analysis to statically check interrelated XML and Java source code. Cross-language relations are formalized into consistency constraints checking properties of abstract syntax trees of parsed XML files and Java source code. PAMOMO [35], utilizes triple graph patterns to define constraints, i.e., relations between models. The tool allows to specify positive and negative patterns. Positive patterns define two conditions, one for each fragment, under which a relation is present. Negative patterns define single constraints for contents forbidden to occur in models. That is, a set of positive patterns constitutes a search-based relation model.

Also GPLs are used to express search-based relation models. For example, in SmartEMF [45] heterogeneous XML models are compiled to Prolog knowledge bases on which cross-language relation constraints, written as Prolog rules, are executed. The Prolog rules encoding constraints constitute a search-based relation model.

Mechanisms for Identification of Interrelated Fragments. As the four examples for the relation models demonstrate, different mechanisms can be utilized to identify related fragments. We observe three different kinds of such mechanisms.

Physical Navigation In case mograms are in a lexical language representation, fragments can be identified by positions in the character stream. For example,

the interface-based relation model in Lst. 1.4 and Lst. 1.5 specifies relation ends by locating fragments via an offset and length in a stream of characters.

Path Navigation Mograms with syntactic language representations allow to identify fragments by path expressions navigating the data structure of the language representation. For example, the explicit relation model in Lst. 1.1 utilizes URIs to specify relation ends in mograms.

Query Evaluation Alternatively, mograms with syntactic language representations allow to identify fragments via queries. For example, the search-based relation model in Lst. 1.6 specifies relations via queries and constraints.

The mechanism to identify interrelated fragments is influenced by the chosen language representation.

2.3 Relation Types

There exist many different types of relations between mograms in literature. However, different types of relations are caused by operations during software development which require the presence of certain mograms and fragments or they produce one fragment out of the other. We observe the following three fundamental types of relations.

Definition 13. *A relation between two fragments f and g in distinct mograms is a **fixed relation**, if $f = g$. It is a **string-transformation relation**, if the two fragments are similar, i.e., if there exists a transformation T , so that $f = T(g)$ and T is not the identity function. It is a **free relation**, if the two fragments are diverse, i.e., if the relation is neither a fixed nor a string-transformation relation.*

Note, this does not mean that all identical fragments of various mograms in a multi-language software system are necessarily related. Fragments of mograms are only related if an operation during software development, for example a compiler, an interpreter, a code generator, etc. requires the presence of fragments f and g in certain mograms or such an operation produces one fragment out of the other.

Free Relations. Free relations rely solely on human interpretation. For example, natural language text in documentation can be linked to source code blocks highlighting that certain requirements are implemented or that a programmer should read some documentation. Steinberger et al. describe a visualization tool allowing to interrelate information across domains, even across concrete syntaxes [82]. Their tool visualizes relations between diagrams and data.

Fixed Relations. Fixed relations occur frequently in practice. For example, the relation between an HTML anchor declaration and its link is established by equality of a tag's argument names. Figure 1 shows an example of a fixed relation across language boundaries (e.g, on lines 53 and 16).

Waldner et. al. discuss visualization across applications and documents [90]. Their tool visualizes relations between occurrences of a search term matched in different documents.

String-transformation Relations appear often in multi-language software system. For example, the Wicket framework requires identifiers in HTML files to have accessor methods in a corresponding Java class. The Wicket identifier `loginName` on line 22 in Fig. 1 requires a method with the name `getLoginName` and `setLoginName` in the corresponding Java class, see lines 82 and 91 in Fig. 1. Depending on the direction, a string-transformation relation either attaches or removes `get/set` and capitalizes or decapitalizes `loginName`.

Domain-Specific Relations. Besides the three fundamental relation types discussed above, relations can be typed with semantics specific to a given domain or project. Additionally, domain-specific relations can be free, fixed or string-transformation relations. For example, a requirements document can *require* a certain implementation mogram, expressing that a certain requirement is implemented. At the same time, some Java code can *require* a properties file, meaning that the code will only produce expected results as soon as certain properties are in place. We consider any relation type hierarchy domain-specific, e.g., trace link classification [70], or typed links as in DOORS.¹⁷

The first three relation types, free, fixed, and string-transformation relations are untyped. They are more generic than domain-specific relations, since they only rely on physical properties of relation ends. Fixed, string-transformation, and domain-specific relations can be checked automatically, which allows to implement tools supporting multi-language system development, such as error visualization and error resolution.

2.4 Inference of Relation Models

Relations and relation models do not necessarily need to be created manually. Instead, they can be inferred automatically or semi-automatically. The inference may either exploit static properties of a system, i.e., its mograms, or its dynamic behavior [33]. By querying the mograms in a code base together with knowledge about language constructs causing relations between mograms, relation models can be inferred out of mograms themselves. Both *model matching* [16, 33, 87, 88] in the model-driven development community and *schema matching* [75, 80] in the database community aims to automatically identify relations between various mograms. In both cases are object-graphs, models and/or metamodels, matched to each other and whenever a certain similarity measure for sub-graphs is fulfilled, relations, mostly trace links, are automatically created. Additionally, schema matching often combines both semantic and structural analysis of the schemas.

If relations are first present at runtime, often trace links, they can be inferred out of programs processing the mograms. That is, relation models can also be *inferred by instrumentation of programs*.

Few programming languages, in particular model transformation languages, provide first class support for traceability. They automatically establish trace

¹⁷ www-01.ibm.com/software/awdtools/doors

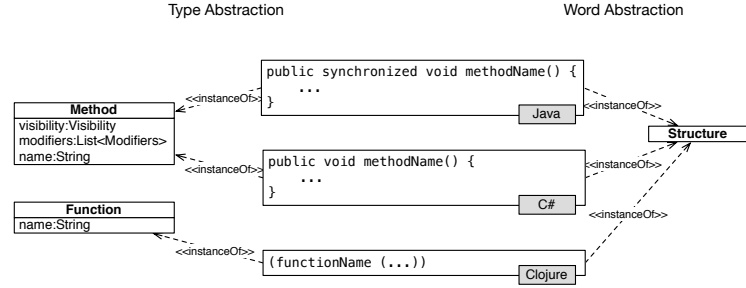


Fig. 4: Type abstraction and word abstraction, two orthogonal abstraction mechanisms.

links between model elements or objects which are in relation because of a transformation directive. For example, Epsilon Transformation Language (ETL) [58] automatically generates a trace model for each model transformation guarded by a post condition. Atlas Transformation Language (ATL) [94] establishes a trace models via a similar mechanism. Also the QVT [67] transformation language has built-in support for traceability [9]. All three languages are rule-based transformation languages, targeting model to model transformations. Model to text transformations can handle traceability similarly. For example, the *MOF Model to Text* transformation language [68], which automatically establishes trace links between model elements and position of text blocks in generated files.

Operations interrelating mograms can be instrumented by other external programs, so that relations are automatically established without modification of the operation. Jouault [52] automatically merges traceability rules into existing ATL transformation rules before their execution populating a trace model. Gammel et al. [32] infer trace links not by instrumentation of transformation code, but by connecting a generic traceability framework to the framework executing the transformation.

3 Implementing MLDEs

In this section we present TexMo (Sect. 3.3) and Coral (Sect. 3.4), two new MLDEs following two radically different design strategies within our taxonomy. But first, we introduce and discuss possible mechanisms of abstraction which are applicable when constructing language representations (Sect. 3.1). Also we discuss qualitatively the impact of design decisions to the created MLDE (Sect. 3.2).

3.1 Creation of Language Representations – Applying Abstraction

Mograms can be, depending on the tool processing them, instances of many languages. For example, a Java 5 program is also a Java 6 program. Independently of tools, mograms can also be represented in many ways. For example, a mogram

containing a program in Java 5 can be represented as instance of the MoDisco Java 5 model [17], as instance of the JaMoPP Java 5 model [42], or as instance of our Java 5 model [72]. All three models are different representations of the same language.

When creating language representations MLDE builders need abstract language concepts into language representations. We observe two orthogonal abstraction mechanisms in modeling. First, *type abstraction*, also referred as *ontological metamodeling* or *logical metamodeling*. Second, *word abstraction*, also referred as *linguistic metamodeling* or *physical metamodeling* [10, 11, 89]. Type abstraction is a unifying abstraction which describes domain concepts along with their properties, whereas word abstraction is a simplifying abstraction, describing structures of sentences or structures of sequences of symbols. According to Colburn [19], the fundamental difference of both abstraction types lies, in relying on *content* or on *form* for abstraction. Any of the two abstractions can be applied at the same time to create any type of language representation.

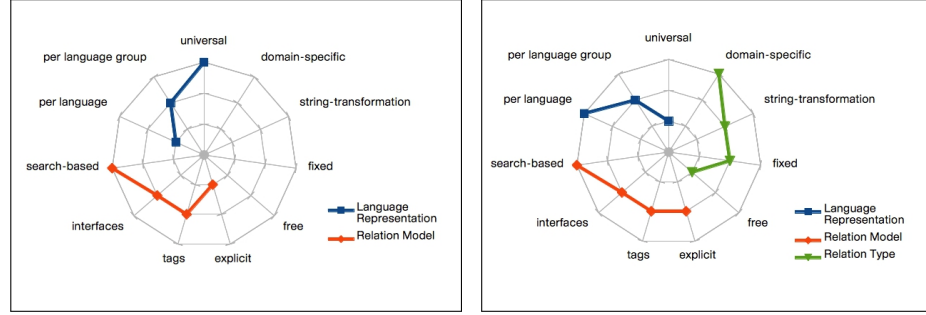
For example, consider Fig. 4, both Java and C# method declarations can include modifiers, but the set of the actual modifiers is language specific. The `synchronized` modifier in Java has no equivalent in C#. Under the type abstraction, Java and C# method declarations can be described by a *Method* type and an enumeration containing the modifiers. In contrast, under word abstraction, Java and C# method declarations could be described by a common simple *Structure* type that neglects the modifiers and universally represents blocks of information. Obviously, in the type abstraction Java and C# methods are distinguishable by their corresponding modifiers, whereas in the more generic word abstraction this information is lost.

The type abstraction is preferable for per language and per language group representations. Word abstraction is preferred for universal representations. Consider the example in Fig. 4, using type abstraction, the concepts of two imperative and one functional language are not easily unifiable, whereas using word abstraction, methods and functions can be abstracted into a single model element such as *Structure*. The choice of abstraction influences the specificity of the representation, affecting the tools. Word abstractions are more generic than type abstractions. For instance, more specific cross-language refactorings are possible when languages are described using type abstraction, while the refactorings in the systems relying on word abstraction automatically apply to a wider class of languages.

Abstraction of arbitrary languages into language representations is a powerful tool as it allows to build generic tools integrating diverse languages with each other.

3.2 Discussion of MLDE Design Choices

Every design decision reflected in the design space of MLDEs (Fig. 2) has a direct impact on the functionality and possible features of the resulting MLDE. In the following we discuss qualitatively the impact of particular decisions across two dimensions: *adaptability* and *feature richness*. We categorize the impact in these



(a) The impact of design choices on adaptability of a MLDEs. Relation types are not included, as they have no impact on adaptability.

(b) The impact of design decisions on richness of functionality of a MLDEs

Fig. 5: The impact of design choices of MLDEs along the relative measures low (inner ring), medium (central ring), or high (outer ring) with respect to adaptability and feature richness.

dimensions using the relative measures low, medium, or high. The purpose of this discussion is to raise awareness towards the impact of design decision using two dimensions as examples.

Adaptability is the ability of an MLDE to be used for development of different heterogeneous multi-language systems. The adaptability of an MLDE depends primarily on the choice of language representation. Since a universal language representation incorporates any used language, it is the best choice when the MLDE should be used for development of various heterogeneous software systems. Consequently, adaptability of a universal language representation is high, see Fig. 5a. Adaptability, decreases with per language group representation, and is even lower for per language representation. In the latter cases any new languages might need to be integrated into the language representations before they can be used in the MLDE. This deficiency is negligible for systems addressing a very stable domain, where the set of languages is known upfront, and it changes rarely.

Explicit relation models have low adaptability. They contain hard links between mogram instances. Tags and interfaces have medium adaptability. They still describe relations on mogram instances but the relation ends are not hard wired. For tags and interfaces relation ends are made explicit, but the relation itself is implicit until an interpreting system establishes them. Search-based relation models demonstrate the highest adaptability since they interrelate mograms at meta-level (language level). Search-based relation models can be reused for development of multi-language system in similar domains.

The choice of relation types supported by a MLDE does not have an impact on its adaptability. Relation types just enrich the relation model with further information. They do not directly refer to any mograms of developed systems.

Richness of Functionality describes the amount of possibly implementable MLDE functionality that leverages the language representation, relation model, and relation types. Such functionality may be elaborate visualizations of interrelated code, versioning of cross-language relations, elaborate cross-language refactorings, etc.

A per language representation has high richness of functionality. Per language representations encode more specific information than the more generic per language group and universal representations. The more specific information is kept in a language representation, the more MLDE functionality is conceivable.

Search-based relation models have high richness of functionality compared to medium richness of functionality for tag-based, interface, and explicit relation models. The former are more generic, since they interrelate mograms at metalevel. But relations established from search-based relation models still contain the same amount of information as relations in the other three relation model types. The more generic a relation model, the wider a MLDE can be applied to various software projects.

Similarly, the more information is kept by relation types, the more functionality is conceivable. Therefore, free relations have low richness of functionality, since they interrelate mograms without indicating the reason for it. Fixed and string-transformation relations have medium richness of functionality, since functionality can leverage the physical properties of the relation ends. Obviously, domain-specific relations have the highest richness of functionality. They keep arbitrary information about the reason for their existence, thus, they allow for MLDEs with rich domain-specific functionality.

3.3 TexMo

TexMo¹⁸ is an MLDE using a universal language representation, with an explicit relation model, and supporting basic types for cross-language relations. As mentioned in Sect. 2, a universal language representation allows to easily deploy TexMo for development of arbitrary multi-language systems relying on textual languages. With TexMo we opt for an explicit relation model since it seems to be the most common design choice from a developers perspective. Alone the survey by Winkler and Pilgrim [92] reports twelve different explicit relation models for capturing traceability information. They are all tailored to a particular solution. We believe that an explicit relation model allows for easy inspection and debugging of encoded relations, since all relations are collected in a central artifact.

TexMo's relation model implements uni-directional relations using a *key-reference* metaphor. For example, `login.title` on line 171 in Fig. 1 is a key in TexMo

¹⁸ <http://www.itu.dk/~ropf/download/texmo.zip>

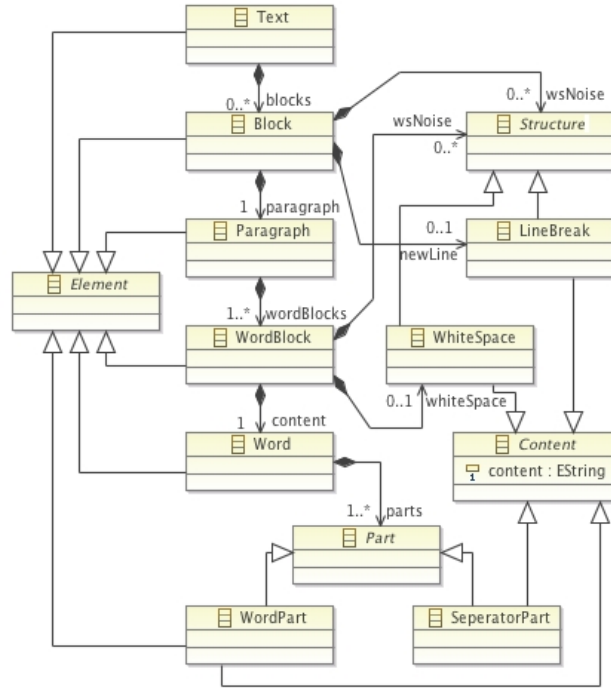


Fig. 6: Text Model – an example of universal language representation as used in TexMo.

and `login.title` on line 52 is a reference in TexMo. TexMo relations are always many-to-one relations between *references* and *keys*. We summarize how TexMo supports the cross-language support (CLS) mechanisms presented in Sect. 2:

1. *Visualization.* TexMo highlights keys and references using gray boxes. Keys are labeled with a key icon and references are labeled by a book icon. Inspecting markers reveals detailed information, e.g., how many references in which files refer to a key.
2. *Navigation.* Users can navigate from any reference to the referred key and from a key to any of its references. Navigation actions are invoked through via the context menu.
3. *Static checking.* Fixed relations in TexMo’s relation model are statically checked. Broken relations, i.e., fixed relations with different string literals as key and reference, are underlined red and labeled by a standard error indicator in the active editor.
4. *Refactoring.* Broken relations can be fixed automatically using quick fixes. TexMo’s quick fixes are key centric rename refactorings. Applying a quick fix to a key renames all references to the content of the key. Contrary, applying a quick fix to a reference renames this single reference to the content of the corresponding key.

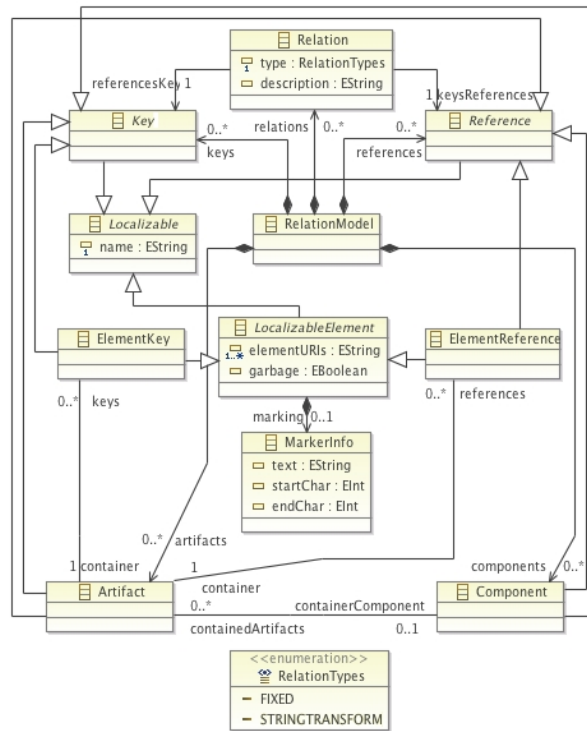


Fig. 7: TexMo's explicit relation model.

On top of these CLS mechanisms, TexMo provides syntax highlighting for 75 languages. GPLs like Java, C#, and Ruby, as well as DSLs like HTML, etc. are supported. Standard editor mechanisms like undo/redo are implemented, too.

Universal Language Representation. Finding a universal language representation, i.e., a representation for any textual language, is challenging since meaningful concepts for relation ends have to be provided. Recall the example from Fig. 4, we have to find a language representation unifying for example, methods for object-oriented languages and functions for functional languages. Now think of how to extend the language representation to include mark-up languages, so that cross-language relations can point to important concepts such as method names, function names, and tag names. Finding a representative abstraction for universal language representation is not easy.

But all textual languages share a common coarse-grained structure. The text model (Fig. 6), an abstract syntax tree¹⁹ of any textual language, describes blocks

¹⁹ The grammar rules for TexMo's universal language representation can be found in the file `TexMo.cs` in the TexMo sources.

containing paragraphs, which are separated by new lines and which contain blocks of words. Words consist of characters and are separated by whitespace. The only model elements containing characters are word-parts, separators, whitespaces, and line-breaks. Blocks, paragraphs, and word blocks describe the structure of a mogram. Separators are non-letters within a word, e.g., `'/'`, `'.'`, etc., allowing representation of typical programming language tokens as single words. Note, that TexMo's universal language representation is only one possible universal language representation.

TexMo treats any mogram as an instance of a textual DSL conforming to Fig. 6. For example, a snippet of Java code `add(new Label("title" ...`, line 52 in Fig. 1, looks like: `Block[Paragraph[WordBlock[Word[WordPart("add"), SeperatorPart(content:"(", WordPart("new"))], WhiteSpace(" ")], WordBlock[Word[WordPart("Label"), SeperatorPart(content:"(", WordPart("title"), SeperatorPart(content:":;"), WhiteSpace(" ")], ...]]]` (using Spoofox [55] AST notation).

Obviously, TexMo's universal language representation model relies on word abstraction, it abstracts over form not over content. This allows for a quite simple language representation model and for automatic generation of a single parser, which parses any textual mogram into an instance of this model. Using type abstraction for language representation would either require a much larger language representation model, unifying language concepts of diverse languages or it would require very sophisticated parsers, which are able to fill instances of this model.

An Explicit Relation Model. TexMo uses an instance of the explicit relation model presented in Fig. 7 to keep track of relations between mograms in different languages. It allows for relations between fragments of mograms (ElementKey and ElementReference), between mograms (Artifacts) or components (Components).

The relation model instance is kept as a textual artifact storing relations between mogram instances. Listing 1.1 illustrates the key-reference relation between the string literal `login.loginName` on lines 21 and 173 in Fig. 1. Relation ends, i.e., interrelated model elements (line 24 Lst. 1.1) are identified by URLs on the language representation model (lines 11-13, 18-20 Lst. 1.1). TexMo automatically updates the relation model instance and the element URLs whenever developers modify interrelated mograms by tracking user input and by reflecting changes it into the relation model. That is, TexMo supports evolution of multi-language systems. So far, the relation model is created manually. TexMo provides context menu actions to establish relations between keys and references. The inference mechanism presented in Sect. 3.4 could be adapted to semi-automatic generation of TexMo's explicit relation model.

Relation Types. TexMo's relation model supports fixed and free relations. Keys and references of fixed relations contain the same string literal. Free relations allow to connect arbitrary text blocks with each other, for example documentation information to implementation code.

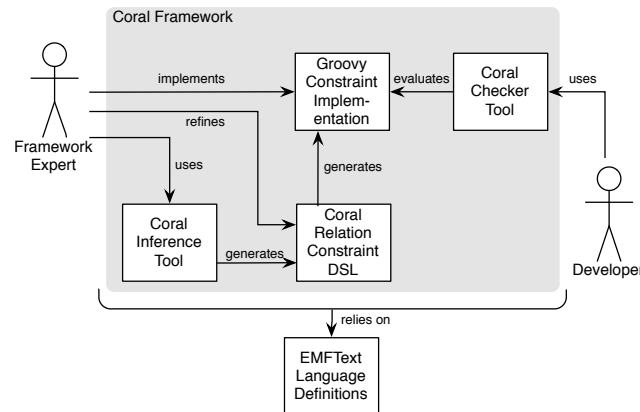


Fig. 8: Coral's architecture and its user groups.

3.4 Coral

Coral²⁰ is an MLDE relying on a per language representation and a search-based relation model, supporting all four relation types. Coral is implemented as an extension of the Eclipse IDE, transforming Eclipse into an MLDE. A search-based relation model allows for high adaptability. Such an MLDE can be parameterized with language representations and libraries containing constraints describing cross-language relations. By parametrization the MLDE can be adapted to development of many kinds of multi-language systems.

The challenge here is to create multiple per language representation models in combination with a search-based relation model. The challenge lies in defining each language in a way that it provides meaningful concepts on which constraints can be expressed, and which are understandable by the constraint developers. Second, a challenge lies in provision of constraints in a generic, reusable manner.

Modern IDEs can be extended to support multiple languages with plug-ins that encode framework-specific knowledge. Such plug-ins exist for most popular application development frameworks, for instance, AspectJ Development Tools,²¹ Spring Tool Suite,²² Hibernate Tools,²³ QWickie an Eclipse plug-in for Wicket,²⁴ etc. The main reason for provision of such tools is to support developers with feedback on cross-language relations. Usually, these tools are not generically parametrizable with language definitions and relation descriptions. One needs to modify the source code of the tools to support new languages. Coral aims at easing adding support for new languages.

²⁰ <http://www.itu.dk/~ropf/coral.html>

²¹ <http://www.eclipse.org/ajdt>

²² <http://www.springsource.org/sts>

²³ <http://www.hibernate.org/subprojects/tools.html>

²⁴ <http://code.google.com/p/qwickie>

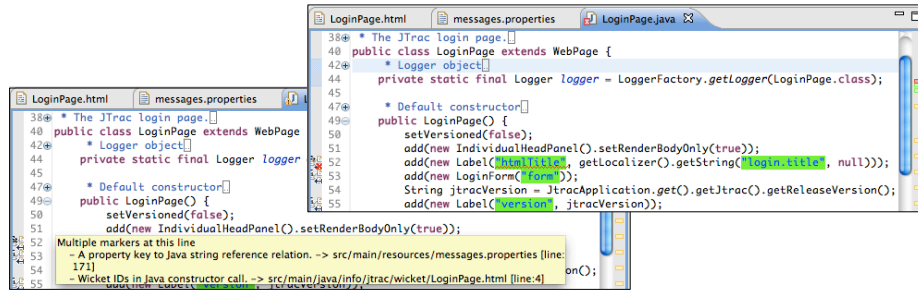


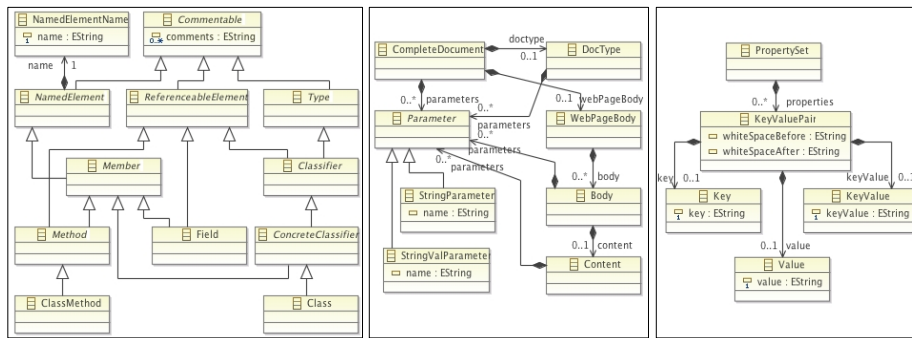
Fig. 9: The CLS mechanisms visualization and static checking in Coral.

Coral supports both, uni-directional and bi-directional relations. In the following, we summarize how Coral realizes the CLS mechanisms presented in Sect. 2:

1. *Visualization.* Coral highlights relation end points using customizable colored boxes, see e.g., line 52 in Fig. 1 and line 171 in Fig. 1. Relation ends are labeled with an icon indicating a relation type, see Fig. 1 left to line numbers. Mouse pointer interaction with the markers allows to reveal detailed information, e.g., the location of the opposite relation end in another file, see Fig. 9 bottom left.
2. *Navigation.* Users can navigate from any relation end to the opposite ends (available via the context menu).
3. *Static checking.* Once established, cross-language relations are statically checked whenever a file is saved. The only unchecked relations are free relations. They do not provide any information that can be used for static checking. Broken relations, i.e., relations not adhering to a constraint specification, are underlined red and labeled by a standard error indicator on the mograms. See Fig. 9 top right.
4. *Refactoring.* Broken relations can be fixed automatically using refactorings. Currently, Coral provides initial basic rename refactorings which rename all opposite relation ends to the content of the relation end to which the refactoring is applied. Coral uses Refactory [12, 76], which supports generic specifications of refactorings. This allows Coral to be easily extended with new kinds of refactorings.

These CLS mechanisms are integrated into Eclipse’s standard editors. Syntax highlighting, editing operations, and keyboard shortcuts are all provided by the host editor and can be used as usual.

Per Language Representation with Models Coral relies on a syntactic per language representation. Figure 10 illustrates excerpts of language representation models for Java (Fig. 10a), HTML (Fig. 10b), and properties files (Fig. 10c). All three language representation models rely on type abstraction. They contain abstractions over a mogram’s contents. The language representations for parametrizing the



(a) An excerpt of the language representation model for Java code. (b) An excerpt of the language representation model for HTML code. (c) The language representation model for properties code.

Fig. 10: Per language representation models for three languages.

Coral framework are generated using EMFText²⁵ [12], a concrete syntax mapper for EMF models. Technically, the Coral framework can be parametrized with other language representation as long as they provide a mapping between the model representation and a mogram's text. At this point, we provide language representations for Java 5.0 (a slightly modified Java model from [42]), XML, Hibernate XML, HTML, properties files²⁶, and for plain text files. These language representations can be downloaded along with the Coral tool. New languages can be easily integrated into Coral. They are just standard Eclipse plug-ins which need to be installed to Eclipse and registered to a Coral plug-in containing the constraint libraries. Note, Coral employs a lazy approach when representing mograms with models. That is, only when static checking and refactoring are invoked, the model representations for the corresponding mograms are present in memory.

Coral DSL. Coral uses a search-based relation model to keep track of relations between multi-language mogram code. The Coral DSL is used to describe cross-language relations as constraints, which interrelate mograms at language level (metalevel). The constraints are kept in library files in Coral DSL.

Listing 1.6 illustrates the Coral DSL. The library contains five constraints, which explicitly encode framework-specific knowledge. The constraints specify how the Wicket web-application framework expects the three languages Java, HTML, and properties files to be interrelated. Constraint libraries form the search-based relation model. The listing starts with a declaration of languages constituting to a relation. Additionally, for each language it is declared which language elements

²⁵ <http://emftext.org>

²⁶ A modified version of http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Properties_Java

contribute in a relation (lines 1-4). Imported language elements can be found in Fig. 10. For Java these are for example element names (`NamedElementName`) and for HTML these are string-value parameters (`StringValParameter`). The Coral DSL allows to name these language elements specifically using the `is` keyword, which maps a name to a Java class representing the language element. Constraint declarations follow these “import” declarations.

Constraints are always typed, such as, `string transformation` or `fixed` (lines 6, 9, 12, 15, 18). A constraint connects two language elements of two distinct languages in a unidirectional (\leftarrow , not shown in the example) or bidirectional (\leftrightarrow) way. Constraints have a severity (`info`, `warning`, `error`) and a message block, whose contents are displayed on established cross-language relations. The constraint logic can be either implemented in an implementation block (not shown in the example) or by provision of a method name referring to a Groovy [21] method implementing the constraint, such as, `wickedIDsInHTML` (the Groovy code is not illustrated here but on the project page²⁷). Method stubs with a complete signature are automatically generated so only the bodies need to be manually implemented.

All constraint libraries, files with a `.coral` extension, reside in a Coral constraint library plug-in. All libraries in this plug-in are automatically evaluated by the Coral framework.

Using the Coral inference tool, see in Sect. 3.4, allows to automatically generate a library with possible constraints.

Relation Types. Coral implements all four relations types of our taxonomy, i.e., fixed, string-transformation, free, and domain-specific relations. Relation ends of fixed relations contain the same string literal and the relation ends of string-transformation relations contain similar string literals. Figure 1 shows fixed and string-transformation relations. For example, a fixed relation between line 52 in `LoginPage.java` and line 16 `LoginPage.html` and a string-transformation relation between line 82 in `LoginPage.java` and line 22 `LoginPage.html`. A broken relation is shown in Fig. 9 line 52 top right. Free relations and domain-specific relations are not shown in our example. They are useful as soon as Coral is deployed in a development project and domain knowledge needs to be captured.

Coral behind the Scenes. Coral automatically establishes cross-language relations when it is parametrized with libraries containing framework-specific constraints and with language representations. Coral consists of three main components, see Fig. 8. First, the Coral DSL allows for declarative specification of constraint libraries. Second, the Coral checker tool, which statically checks mograms of the developed system with respect to the constraint libraries. Third, the Coral inference tool, automatically infers possible constraints from heterogeneous mograms into a library in Coral DSL. Presently, we provide libraries of cross-language constraints for Hibernate and Wicket. More libraries will be available online from the Coral website.

²⁷ <http://www.itu.dk/~ropf/coral.html#Constraints>

The Coral checker operates on constraints compiled into Groovy scripts. Groovy is a dynamic programming language for the JVM. The generated Groovy code serves two purposes. First, it collects all language elements, which potentially participate in a relation. Second, it evaluates each constraint on all possible combinations of language elements. The generated scripts are newly interpreted whenever Coral’s static checking is called. That is, Coral DSL code is highly dynamic, new constraints can be introduced into a library at any time.

The architectural division into Coral inference tool and Coral checker tool is caused by the existence of two distinct user groups. The Coral checker tool targets multi-language system developers. They are MLDE users utilizing the implemented CLS mechanisms. Since the checker tool is only useful when parametrized with constraint libraries, the Coral inference tool supports (framework) developers when creating new constraint libraries. Providing constraint libraries, which explicitly encode cross-language relations, is a formalized way of writing framework documentation.

Development of Cross-Language Relation Libraries. The development of constraint libraries is supported in two ways. Framework developers, who know what kind of constraints their frameworks impose on mograms can implement these constraints directly into Coral libraries. They are supported by automatically generated editors, which provide a model view of the sources. Clicking on mogram code the editor reveals the corresponding language element. The language elements types needed for constraint specifications are easily accessible.

Coral is a new tool. To support its users to create constraint libraries until framework developers provide such libraries, Coral provides an inference tool. The inference tool suggests possible cross-language constraints, which are encoded by used frameworks.

Automatic Inference of Cross-language Relations. The inference process is illustrated for a pair of files in Fig. 11. The inference process on each file pair is divided into three atomic phases, see Fig. 11.

1. *Text Intersection.* The first phase searches for all longest common substrings of two mograms in different languages. This phase can be described as text intersection, where the result is a set of fragments that are shared by two mograms. This phase is language agnostic. It considers input as a text and relies on lexical language representation. Obviously, this interference only produces valuable results, when mogram’s texts are available in unobfuscated form. This first phase will not provide any useful results for running it, on encrypted mograms.
2. *Filtering.* The set of longest common substrings is the input for the second phase. Both mograms are loaded and each file is treated as a model (abstract syntax graph). All longest common substrings which are enclosed entirely by a language element’s attribute in both languages, are potential relation ends. `NamedElementName`, `StringParameter`, or `Key` in Fig. 10 are examples for

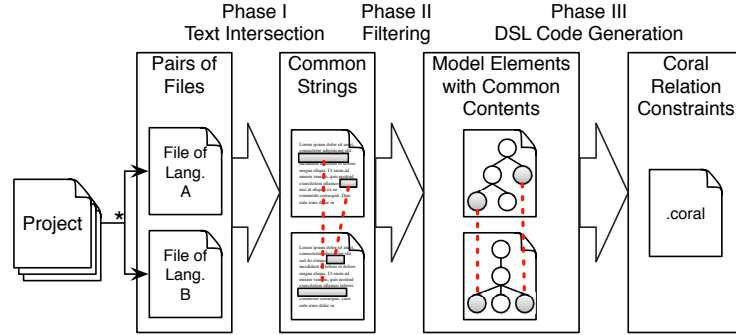


Fig. 11: The four phases to inferring CLRC libraries.

language elements, which potentially enclose a longest common substring in the **name** attribute. Obviously, this phase is not language agnostic anymore.

3. *DSL Code Generation*. The instances of possible cross-language relations from the second phase are abstracted into constraints at metalevel. Consider an example of abstracting fixed relations. We collect all pairs of model elements sharing the longest common substrings within the same attribute. The concrete shared values are discarded, and an equality constraint relating the attributes of the corresponding language elements is generated. The generated library captures the information about the related language elements of the two compared languages.

Even-though, the inference tool is illustrated for a pair of files, it can also operate on single files and on entire projects. When applied to a single file, the inference tool considers all other files in a multi-language system and runs a pair-wise inference. When applied to an entire project, the inference tool runs pair-wise comparisons for all possible combinations of files.

The resulting library usually needs to be refined manually, as it may contain false positives. Consider for example a run of the inference tool on the two files `LoginPage.java` and `LoginPage.html` in Fig. 1. The inferred library would contain a constraint, which establishes a string-transformation relation between the string literal `label` in `<td class="label">` on line 21 and the string literal `Label` on line 52 (the class name of the constructor call). Of course the string literals are similar and they appear in atomic language elements, i.e., they fulfill the requirements of the heuristics of the inference tool, Coral's inference tool will not automatically sort out such false positives. In certain domains they might represent valid cross-language relations. A library developer has to investigate which constraints describe valid cross-language relations and what technology or framework imposes a certain constraint on the source code.

In general, cross-language relations and their constraints cannot be inferred completely automatically. Free relations may relate arbitrary blocks of information. A generic inference tool should not be polluted with domain-specific concepts, which get hard-wired into it. Free and domain-specific relations cannot

be established automatically in a generic manner since there is no generic similarity measure for them. That is, Coral allows for semi-automatic inference of fixed and string-transformation relations.

3.5 Comparison of TexMo and Coral

In this section we compare TexMo and Coral. We rely on three comparison criteria. First, the CLS mechanisms, visualization, navigation, static checking, and refactoring of cross-language relations. Second, the fundamental design choices of or taxonomy, language representation, relation model, relation types, and inference of relation models. Third, the two dimensions, adaptability and richness of functionality.

Both TexMo and Coral implement the same CLS mechanisms, see Tab. 1. Visualization, navigation, and static checking have similar look and feel for the users of the tools. Both MLDEs implement rename refactorings that maintain consistency of relation endpoints. In addition, Coral supports implementation of arbitrary cross-language refactorings through Refactory. For TexMo renaming remains the only feasible refactoring, due to its language agnostic syntax representation, the universal language representation based on word abstraction neglecting types. Coral’s per language representation is based on type abstraction and many diverse refactorings rely on type information. Effectively, Coral supports much richer functionality than TexMo.

Simultaneously, the choice of language representation influences the lower adaptability of Coral, when compared to TexMo. The latter can be immediately used for new multi-language systems. Coral has to be parametrized with new language representations and new constraint libraries to fit a new setting. As soon as a large set of language representations and constraint libraries is available for Coral, this adaptability problem will become much less significant.

The choice of the relation model has an impact too. For example, false positives in automatically inferred relations in an explicit relation model are harder to handle than in a search-based relation model. An explicit relation model interrelates mogram instances with each other, whereas a search-based relation model interrelates mograms on language level. Therefore, when manually adapting automatically inferred relations, in TexMo’s relation model, one has to navigate and master many relation instances, which use cryptic identifiers as relation ends. Whereas, in Coral’s constraint libraries one would manually modify a comparatively small relation model since constraints are expressed at metalevel.

4 Experimental Investigation

In this section we investigate the challenges and motivation for developing MLDEs. First, we demonstrate that implicit relations are ubiquitous and dense, which explains the need for MLDEs and imposes hard performance requirements on them. Second, we approach the users of MLDEs in an attempt to estimate how

	TexMo	Coral
Visualization	✓	✓
Static Checking	✓	✓
Navigation	✓	✓
Refactoring	Rename refactoring	Any refactoring with a Refactory rule
Language Representation	Universal	Per Language
Relation Model	Explicit	Search-Based
Relation Types	Free, Fixed	Fixed, String-Transformation, Free, Domain-Specific
Inference of Relation Model	✗	Artifact Interpretation
Adaptability	high	low/medium
Richness of Functionality	low	high

Table 1: Comparison of the two MLDE prototypes with respect to three criteria.

useful these tools are. Third, we survey the community of language implementation experts, to find out, whether in experts eyes the MLDEs, and especially generically parameterized MLDEs, would be an improvement over the current practice.

4.1 Cross-language Relations in a Typical Multi-language System

We shall now investigate how common cross-language relations are in a typical multi-language system. We find out that these relations are so ubiquitous that they actually pose a performance challenge for tools.

Method. We use Coral inference to automatically establish cross-language relations in JTrac. We obtain two constraint libraries: one containing five constraints for the web-development framework Wicket and another one with five constraints for the persistence framework Hibernate. We address the following questions:

- RQ1** How many cross-language relations exist in a representative multi-language system?
- RQ2** How long does it take Coral to establish cross-language relations?
- RQ3** What is the distribution of cross-language relations in a representative multi-language system?

We used just one iteration of inference and verification to develop the two libraries in this experiment. A complete workspace including the Coral library plug-in and the JTrac sources for reproduction of the experiment are available online.²⁸ We have run the inference on a 2.9GHz Intel i7 Mac Book with 8GB of RAM, of which 4GB were assigned to the Java 6 virtual machine.

²⁸ http://www.itu.dk/~ropf/coral/tech_experiment.zip

Subject. We use JTrac (v2.1.0)²⁹ as the study subject. JTrac’s code base contains 372 files: source code in Java (140 files), HTML (66), property files (30), XML (16), JavaScript (8), and 29 other source code files such as shell scripts, etc. Similar to many web-applications, JTrac implements the model-view-controller (MVC) pattern. This is achieved using popular frameworks: Hibernate³⁰ for OR-Mapping and Wicket to couple views and controller code. The remaining 83 files are graphical images and a single jar file. Coral, and thus this evaluation, does not consider these files since they do not contain information in a human processable, textual syntax, i.e., they are not meant to be text processed by an editor and thus no target for Coral. Clearly, JTrac is a representative of a multi-language system.

Results. The results of measuring the number of cross-language relations established by each constraint, and the time it takes to evaluate a constraint, are summarized in Tab. 2. In JTrac, there are at least 4,941 cross-language relations (question **RQ1**). The Coral tool automatically establishes all of these relations, using just ten constraints distributed over two libraries. It takes in total 1.31 minutes to check all constraints on all possible combinations of files (**RQ2**). A check of a single constraint takes on average 2.27 milliseconds.

Majority (4,741) of cross-language relations in JTrac, are described by only three constraints in the Wicket Coral library. Interestingly, even though the Hibernate OR-mapping is defined in a single file (an XML-based DSL), the five constraints in the Hibernate Coral library still describe 165 relations. The relations are not distributed homogeneously over JTrac’s code base. They form sub-clusters of mograms in the code base. For example, the relations established by the Hibernate Coral library tie together a resource folder containing the Hibernate mapping model and the properties files used for localization with a Java package containing all the Java classes, which form the application’s domain model. The Wicket library contains constraints, which cluster together a resource folder containing the properties files with multiple Java packages. Additionally, the Wicket library heavily interrelates Java and HTML code located in a Java package, which contains the application’s view code.

On average each file participates in more than 13 cross-language relations. Nearly every fourth Java class has references to the Hibernate mapping model and in total about one third of all the mograms (Java, HTML, and properties files) participate in at least one cross-language relation. Clearly, with these many relations being implicit, and unsupported by a development environment, broken relation errors are hard to avoid. However, as the experiment shows, handling this amount of relations is entirely feasible in a MLDE. Consequently, on average it takes just below 30ms to check one mogram, open in an editor, for the relations in which it participates. Standard UI research indicates that response time for visualization of results of computation actions happening without display of any progress indicators should never exceed two seconds [31,63]. So even if the density

²⁹ <http://sourceforge.net/projects/j-trac/files/jtrac/2.1.0>

³⁰ <http://hibernate.org>

of relations would be much higher in other projects, it is very likely that they can be checked within acceptable time.

	Hibernate	Wicket	Total
# cross-lang. relations	165	4,776	4,941
# of checks	700	33,900	34,600
Total time	0.04min	1.27min	1.31min
Average time	3.79ms	2.24ms	2.27ms
# of relations per file			13.21
False Positives	0/165	0/578	

Table 2: Cross-language relations established by Coral inference for JTrac.

We manually verified for false positives within the harvested cross-language relations. For the Hibernate library we checked all 165 established relations between the Hibernate mapping file and thirteen Java classes (complete sample). For the Wicket library we checked 578 random relations out of the established 4,941. These relations involved three properties files, twenty HTML files, and nineteen Java classes. The sample size exceeds ten percent of the all affected sources. We have found no false positives.

Threats to Validity. The declared number of established cross-language relations and the timing results are strict lower bounds, in the sense that JTrac might contain more relations, and more constraints would take longer for evaluation. Our constraint libraries contain basic constraints. Currently, we do not infer complex constraints which for example respects Java’s inheritance mechanism. That is, the established relations are not complete as long as the constraint libraries are not complete. We examined only a subset of established cross-language relations for the Wicket library, for our checks on possible false positive relations. We believe that this subset is representative since it considers a random choice of ten and more percent of the interrelated Java, HTML and properties files.

We provide measurements for checking each constraint. The given values refer to the evaluation of a constraint. We omitted the times of loading the mograms into models (data structures). The latter step is quite costly in comparison to the quick checks. Therefore, an effective, incremental loading strategy will be investigated in a next version of Coral.

4.2 CLS Mechanisms are Beneficial

We conducted an experiment evaluating CLS mechanisms as implemented by TexMo [73, 74], to demonstrate that these mechanisms are actually beneficial when developing multi-language system and that they are appreciated by developers.

We report the results of multi-language software system development supported by the four fundamental CLS mechanisms from a user perspective. Here we focus on reporting qualitative feedback. See [73] for full experiment results and analysis.

Method. We run a single-factor with two alternatives experiment. The factor alternatives are TexMo with the four CLS mechanisms, visualization, navigation, static checking, and refactoring disabled and the full-featured TexMo with CLS mechanisms enabled. A treatment group uses the full-featured TexMo and a control group, uses the restricted TexMo, which simulates multi-language system development using a contemporary IDE. Essentially, the experiment evaluates the four CLS mechanisms but not TexMo itself.

We asked the experiment subjects to perform three tasks representing typical development and customizations tasks on the JTrac system. The first task asks to locate and fix a broken cross-language relation between Java and HTML code. The second task asks for renaming a key in a properties file, what breaks a cross-language relation. The subjects should fix the broken relation. The third task asks to replace a block of code, what breaks multiple cross-language relations. The subjects should explain how to fix the introduced errors. After the task is completed we ask the following question:

RQ4: *Do you think TexMo could be beneficial in software development? Why?.*

Subject. The experiment was conducted with 22 experimental subjects falling into four major categories: software professionals along with PhD, MSc, and undergraduate students at the IT University of Copenhagen. The participants are between 18 and 48 years old, average age is around 29 years, median 28. Nineteen participants are working as professional software engineers for at least half a year, with maximum of 13 years (average work experience: around 3 years, median 3 years). Two PhD and one MSc student have no experience as professional software engineers. The subjects are distributed into two groups, one per factor alternative. Note, in a pre-experiment we had another five subjects, where three were in the treatment group and two in the control group.

Results. Recall the overall research question from Sect. 1: “*To what extent MLDEs are desired by users, and what aspects of MLDEs are particularly helpful?*”. The results of our experiment demonstrate that developers using CLS mechanisms find and fix more errors in a shorter time than those in the control group, that they perform development tasks on language boundaries more efficiently, and that even unexperienced developers provided with CLS perform similarly or better than experienced developers in developing multi-language systems.

In the following we provide answers of subjects in the treatment group to question **RQ4**.

- *TexMo’s concepts are really convincing. I would like to have a tool like this at work.*
- *Liked the references part and the checking. Usually, if you change the keys/references you get errors at runtime [which is] kind of late in the process.*

- *It improves debugging time by keeping track of changes on source code written in different programming languages that are strongly related. I do not know any tool like this.*
- *I see it useful, especially when many people work on the same project, and, of course, in case the projects gets big.*
- *I did development with Spring and a tool like TexMo would solve a lot of problems while coding.*
- *In large applications it is difficult to perform renaming or refactoring tasks without automated tracking of references. ... If there would be such a reference mechanism between JavaScript and C#, it would save us a lot of work.*
- *[TexMo] solves [a] common problem experienced when software project involves multiple languages.*
- *Yes. I do not know enough about web-programming, but the key/ref relationships between HTML and Java seem like a common pitfall to me.*
- *Yes. As code evolves refactoring may be needed. TexMo makes it easy to do so – it's helpful.*
- *Yes. I think when I use Visual Studio for ASP.Net applications, something similar allows me to detect errors when I change a reference name, and there is a dependency from an ASP to a C# file.*
- *Yes. Easy to fix your mistakes.*
- *Yes. Easy markup. A small challenge in understanding the structure of files because of Eclipse.*

The answers of the treatment group subjects to the research question indicate that CLS mechanisms are beneficial and that such features are missing in existing IDEs. Clearly, CLS mechanisms are appreciated by the developers. That is, from a user's perspective it is important to implement them in IDEs (MLDEs). Some developers in the control group were negatively surprised that current IDEs do not provide CLS mechanisms, considering them as something obviously necessary.

4.3 The Language Integration Survey

In the final part of our investigation, we conduct an online survey among language developers, to verify our assumption, that a generically parametrizable MLDE would be welcomed in language development community.

Method. Our survey contains 15 questions in a web-based questionnaire. The survey takes about ten minutes to complete. We ask for example, for how many languages a subject constructed, how the languages are typically used, and how they are typically integrated with other languages. The complete questionnaire bundled with anonymized results is available online³¹. Twelve of the questions and the corresponding results are given in Tab. 3. The remaining questions were cross checking and context questions not used directly in the analysis below. We aim to answer the following research questions with our survey:

RQ5 What are the characteristics of constructed computer languages?

³¹ <http://www.itu.dk/people/ropf/survey.zip>

- RQ6** Do language developers integrate different languages with each other? If so, how?
- RQ7** Are the tools for language integration provided by language developers generic?

Subjects. The survey targets language developers. We distributed the survey widely in the online community through forums and mailing lists of Xtext,³² EMFText,³³ ANTLR,³⁴ JavaCC,³⁵ Parboiled,³⁶ and Pyparsing.³⁷

Xtext and EMFText are EMF-based language workbenches. ANTLR and JavaCC are parser generators. Parboiled and Pyparsing are libraries for development of parsers based on parsing expression grammars, another kind of grammar specification. All tools and frameworks are used for specification and generation of GPLs and DSLs.

Results. The survey was open for 25 days, until October 18, 2012. We have received 25 responses. Unfortunately, due to the open nature of the survey, we cannot estimate the response rate. Table 3 presents the most important results.

Regarding RQ5. An average subject has experience with creating more than fifteen languages (Q I). The majority of subjects (88%, see Q II) mention that they are constructing DSLs for diverse purposes ranging from data modeling, visualization modeling over languages for constraint and check specifications to languages for legacy code replacement and for requirements engineering. Still nearly two-fifth mention that they construct GPLs (Q IV). All constructed languages are applied in development of software systems. This follows from the answers on the usage scenarios of the built languages. These results confirm our claim that current software systems are multi-language systems.

Most subjects indicate that their languages are input to transformations, to code generators, or to interpreters. Around a third admits that they also construct languages which are used stand-alone, i.e., only for communication among human stakeholders. Only around 17% of the respondents say that some of the languages they construct do not have any relations to other languages. The other responses to question Q VI indicate that the majority of the constructed languages participate in cross-language relations.

Regarding RQ6. Interestingly, over two-third (Q VII) of language developers provide tools along with their languages, which check for correct cross-language relations, compared to less than a third, who do not. This high level of appreciation towards cross-language integration was a surprise to us. Around half of the language developers provide tools, which do static checking or compile-time (Q VIII) checking of cross-language relations. Half of the subjects provide IDE

³² www.eclipse.org/forums/index.php?t=thread&frm_id=27

³⁴ antlr-interest@antlr.org

³⁵ users@javacc.java.net

³⁶ users.parboiled.org

³⁷ pyparsing-users@lists.sourceforge.net

Question	Result			
	Average			
Q I) How many languages have you created?	14.92			
	Average of lower bound		Average of upper bound	
Q II) How many computer languages are used in the software projects for which you developed new languages?	3.04		8.48	
Q III) How many frameworks are used in the projects for which you developed new languages?	2.04		9.45	
	DSL		GPL	
Q IV) What is the purpose of the languages you developed?	88%		36%	
	Stand-alone	Transformation/Generation		Interpreted
Q V) What is the usage scenario of the languages you developed?	32%	84%		52%
	Referred	Refer	Referred and Refer	No Relation
Q VI) How do the languages you built interact with other languages in corresponding projects?	28%	40%	44%	16%
	Yes		No	
Q VII) Do you provide tools along with any of your languages which automatically check for correct interrelations to other languages?	68%		32%	
	Development-Time		Compilation	Runtime
Q VIII) When do your tools check for correct interrelations to other languages?	56%		40%	28%
	Yes		No	
Q IX) Are your tools, which check the correctness of language interrelations, generic?	8%		60%	
Q X) Do you provide IDE support for the languages you develop?	52%		16%	
Q XI) Do you incorporate the results of your tools which check for correct interrelations to other languages into the IDE?	44%		8%	
Q XII) Are your tools, which incorporate the results of checking for correct language interrelations into the IDE, generic?	8%		32%	

Table 3: Language integration survey questions and quantitative results.

support for their languages (Q X) and two-fifth of the language developers indicate that the results of these checks are reflected in an IDE (Q XI). Remarkably, a fourth of the language developers neither provides any language integration nor tools to enhance IDEs with language integration knowledge.

Regarding RQ7. Even-though many language developers provide tools which check for correct language interrelation (over two-third, see Q VII), most of these tools are not generic (Q IX and Q XII). That is, whenever a new language is added to the development process or as soon as the patterns of cross-language relations change, the tools have to be modified manually. Note, a generically parameterizable tool, which checks cross-language relations is even more valuable, since around a third of the language developers provide no such tools at all.

The overall conclusion from this survey is that **a)** many computer languages are created (Q I), **b)** languages are in fact interrelated and thus, mograms in various languages are interrelated (Q VI), **c)** the projects using the created languages are multi-language system projects (Q II), which rely on multiple frameworks (Q III). Furthermore, many language developers provide tools which check for cross-language relations. But, most of the provided tools are not generic. That is, whenever a new language is used in multi-language system development the tools have to be adapted to support the changed development architecture. We conclude that generic tools for language integration, such as Coral, are worth to be used for language integration. Such tools only need to be parametrized with language representations and possible constraints. All of the developers indicating that they provide no tools for language integration or that their tools are non-generic, could be efficiently supported by a generically parametrizable MLDE, such as Coral.

Threats to Validity. The main threat to validity of the presented survey is the relatively low number of responses. Informal cross-checking with developers in our network however seems to indicate that these results are agreeable. To minimize the risk of having to few responses we decided to let the survey open for responses. We will update the data on the survey's homepage to reduce this risk.

5 Related Work

5.1 Taxonomy

The IEEE Standard Glossary of Software Engineering Terminology [86], defines traceability as *the degree to which a relationship can be established between two or more products of the development process. . . .* In the context of model-driven development this definition reduces to [3] *. . . any relationship that exists between artifacts involved in the software-engineering life cycle. . . . [Such as] Explicit links or mappings that are generated as a result of transformations. . . , Links that are computed based on existing information, Statistically inferred links.* Our work can be addressing certain kinds of traceability needs, but its objective is broader.

We are concerned with any kinds of relations that are useful to maintain during development process, especially during programming—not just traceability.

The taxonomy supports tool builders in their architectural decisions when heterogeneous mograms on different levels of abstraction should be interrelated. It does not provide an answer for how to obtain complete or semantically correct traceability links.

Winkler et al. [92] present a taxonomy for traceability models in model-driven software development. Similarly to our study, their taxonomy is the result of a survey of related tools and literature. Their taxonomy describes common practices for implementing traceability models. Our perspective is broader and more fundamental. We analyze how to represent related mograms. Traceability links are just one special case of domain-specific relations. Furthermore, we define different other relation model types on top of the explicit relation models listed in [92]. We propose to use several types of language representation models, which allow models to be related in a generic manner.

In [22,25] a traceability meta-language and a traceability scheme are presented. These works abstract over specific traceability models to define a general solution to relate mograms using trace links. In contrast, we do not consider generic descriptions of explicit relation models. We are interested in describing abstractly all possible ways of relating information across languages.

Aizenbud-Reshef et al. [3] survey literature and tools on model traceability. Similar to our taxonomy the authors realize that there are different relation model types. They abstract current relation models into two types. One for tag-based relation models and another one for explicit relation models (to use our terminology). Additionally, they describe the need for differently typed traceability links. Compared to Aizenbud-Reshef study, our taxonomy is more formal and it is more generic in that we focus on how to generally interrelate information in heterogeneous mograms. We identify two more relation model types and, more importantly, we describe the different types of language representations.

Aizenbud-Reshef et al. state the following challenge: *“Tool artifacts may not always have a unique identifier, especially if their granularity is smaller than physically stored artifacts. Technologies such as link anchors and bookmarks can be used to identify such artifacts, but more research is required to make such anchors robust when artifacts are edited, cut, and pasted.”* Both presented MLDEs, TexMo and Coral support these evolution steps. The reduction of high-cost of manual creation and maintenance of traceability links is addressed by Coral’s constraint libraries, specified at the language level.

5.2 Multi-language Development Environments

Strein et al. [84] realize that IDEs do not allow for analysis and refactoring of multi-language system and thus are not suitable for development of such. They present X-Develop a MLDE implementing an extensible metamodel used for a syntactic per language group representation. The key difference between X-Develop and TexMo and Coral is the language representation. TexMo’s universal language representation allows for its application in development of any multi-language

system regardless of the used languages. Coral's per language representation allows for easy extensibility of the MLDE by parametrization with new language representations and corresponding cross-language constraints. In X-Develop one would need to extend the per language group representation and invasively extend the tool to support new cross-language relations.

Similarly to X-Develop, the IntelliJ IDEA IDE implements some multi-language development support mechanisms. It provides multi-language refactorings across some exclusive languages, e.g., HTML and CSS.

Chimera [7] provides hypertext functionality for heterogeneous *Software Development Environments (SDE)*. Different programs like text editors, PDF viewers and browsers form an SDE. These programs are viewers through which developers work on different artifacts. Chimera allows for the definition of anchors on views. Anchors can be interrelated via links into a hyperweb. TexMo is similar in that models of mograms can be regarded as views where each model element can serve as an anchor for a relation. Chimera is not dynamic. It does not automatically evolve anchors while mograms are modified. Subsequent to modifications, Chimera users need to manually reestablish anchors and adapt the links to it. TexMo automatically evolves the relation model synchronously to modifications applied to mograms. Only after deleting code blocks containing keys, users need to manually update the dangling references. In Coral's constraints are just re-evaluated as soon as a mogram is modified. Thereby, relations do not have to be manually reestablished.

Jarzabek [51] describes specification of multi-language development environments using interrelated attribute grammars as language definitions. That is, resulting ASTs are syntactic per language representations in which cross-language relations are specified via horizontal attributes with attached semantic expressions. Semantic expressions can be considered as search-based relation model. The advantage of expressing a search-based relation model relying on attribute grammars is, that changes in interrelated fragments of heterogeneous mograms are automatically updated whenever semantic expressions are reevaluated. Unfortunately, the described IDE is VAX-based seems to be discontinued.

Meyers [62] discusses integrating tools in multi-view development systems. Language integration can be seen as a particular flavor of tool integration. Meyers describes basic tool integration on file system level, where each tool keeps a separate internal data representation. This corresponds to the per language representation in our taxonomy. Meyers' *canonical* representation for tool integration corresponds to our universal language representation. Our work extends Meyers work by identifying a per language group representation. Similarly, the prototype ToolNet [6, 30] integrates mograms in different languages by integrating tools. The authors of ToolNet propose a kind of message bus on which registered tools exchange actions applied to various mograms to facilitate for example static checking. Consequently, ToolNet uses a tool based per language representation.

This is similar to Coral's integration strategy, where EMF models are used for language representation and the EMFText-based parsers can be considered as tool adapters. Interestingly, the work hints at visualization, navigation, static

checking, and error fixing as the key features for cross-application relations. That supports our standpoint, that these are the four fundamental CLS mechanisms.

LiMonE [78] is an editor for literate programming integrating natural language and Unified Modeling Language (UML) models via Object Constraint Language (OCL) constraints. Similar to [45] it compiles the mograms in natural language and UML and the OCL constraints into a Prolog knowledge base and into Prolog rules. But since it relies on a custom language representation, it is harder to incorporate new languages into the tool.

A detailed description of the TexMo can be found in an earlier version of this paper [74]. Here we focused on the entire design space, thus we limited the description to the most important design decisions, facilitating a comparison with Coral.

5.3 Search-based relation model

In Sect. 3.4, we present the Coral DSL to define constraint libraries, which form a search-based relation model. Since our per language models are based on EMF, we could have used the Epsilon Comparison Language (ECL) [57], EMF-IncQuery [40], or Prolog [45] alternatively. Indeed the Coral DSL looks similar to the first two languages. However, since we wanted to capture the constraints in framework-specific libraries we decided to implement a separate DSL, tailored to our problem domain. Furthermore, we would like to experiment with different technologies for constraint evaluation. Currently, Coral DSL code is transformed into Groovy code. The generator and the evaluation code can be easily exchanged for further experiments. With the other mentioned solutions we would have been tied to a certain model query framework.

5.4 Inference

We observe three main trends in automatic inference of relations between mograms. First, there is *model matching* [16, 33, 87, 88] in the model-driven development community, where object-graphs, models and/or metamodels, are matched to each other and whenever a certain similarity measure for sub-graphs is fulfilled, relations, mostly trace links, are automatically created. Second, there is *schema matching* [75, 80] in the database community, that aims to automatically identify relations between various schemas (metamodels). Schema matching is similar to model-matching, although it often combines both semantic analysis of the schemas and their structural information. Third, there is *automatic traceability link generation* [4, 8, 24, 39, 60, 79] that tries to automatically identify trace links between natural language documents and source code.

With its staged phases of text intersection and abstraction to constraints on language level, our inference tool can be considered as a hybrid approach between automatic traceability link generation and schema/metamodel matching. Note, Coral's inference tool can be applied directly to visual mograms, since they are serialized into text files. In future work we will provide evaluation results for automatic inference of constraints between textual and visual mograms.

6 Conclusions and Future Work

We have presented an investigation of the MLDEs design space from three different perspectives.

First, we have identified four core cross-language support mechanisms, visualisation, navigation, static checking, and refactoring. We studied the existing literature and presented a taxonomy of tools and research proposals that address these mechanisms using different representations for languages, relation models, and different types of relations.

Second, we took the tool builder role, and described our experience with constructing two new and different MLDEs prototypes, TexMo and Coral, following two different design choices. Our experience with TexMo and Coral confirms the high adaptability of tools based on universal language representations. This representation however comes at a cost of limited richness of functionality.

Third, we have undertaken an empirical investigation of this space, showing that cross-language relations are ubiquitous even in relatively small systems, to the extent that one can hardly expect handling them correctly without tool support. This hypothesis has been further confirmed in experiments with users, who find using CLS mechanisms very helpful, and with language developers who report that very frequently they design languages related to other languages, and need to provide tooling to integrate them. In effect, this paper documents a strong incentive to construct industrial strength generic parametrizable MLDEs. The language development community is lacking a generic parametrizable MLDE.

Technical deliverables of this work include the two MLDE prototypes TexMo and Coral, a number of language representation models and relation models, along with a Coral DSL inference tool. All these tools are available online along with documentation and material used in the experiments.

From a technical perspective, both presented MLDEs, TexMo and Coral, do not only allow to interrelate mograms of different languages but also of mograms in a single language. For example, in Fig. 1 the Java class `LoginPage` extends the class `WebPage`. A Coral constraint interrelating the class name of the class `WebPage.java` and the name used in the `extends` statement can be declared. We do not focus on this fact in this paper, and in our current implementation, since we consider intra-language relations to be appropriately handled by existing tools. However, this ability can be used to enhance and customize static checks and visualizations beyond those provided by current IDEs without extending compilers and other tools.

We plan to extend our MLDEs with efficient language representations. For example, in Sect. 4.1 we measure the time it takes to check constraints, but we neglected the times for loading the mograms as models. The latter step is quite costly in comparison to the quick checks. Therefore, an effective, incremental loading strategy has to be researched and implemented in an upcoming version of Coral. Along this lines we plan to conduct a comparative study on different technical solutions for constraint encoding and constraint checking. As indicated in related work there are other model querying frameworks and we would like

to compare them to our Groovy-based constraint checker. We would like to find evidence for the most efficient technology.

Furthermore, we intend to extend Coral with language representations for office documents, such as word-processor files or spreadsheet files, and with language representations for visual languages, such as UML languages. This would allow the deployment of one MLDE in all development phases of a software system. We intend to provide an evaluation on the quality of Coral in combination with its constraint inference tool in a setting with interrelated visual and textual languages. Currently, we have preliminary insight from previous work [71], that the tool can be applied to mograms in visual languages directly, as they are persisted in textual form.

Acknowledgments We thank Uwe Aßmann and the Software Development Group at TU Dresden for hosting the first author for an extensive period during this work. Uwe Aßmann has suggested to survey the language developer community as part of this study. We also thank Jendrik Johannes, Sven Karol, Jan Reimann, Mirko Seifert, and Christian Wende, David Christiansen, Hannes Mehnert, Jan Polowinski, and Hendrik Pfeiffer for discussions and help with validating the experiments. We thank all those participating in our survey and in the previous experiment on CLS mechanisms. Last but not least, we thank the anonymous reviewers of the earlier version of this paper for their comments and suggestions.

References

1. Zend Technologies Ltd.: Taking the Pulse of the Developer Community. static.zend.com/topics/zend-developer-pulse-survey-report-0112-EN.pdf, seen: Feb. 2012
2. THE OPEN SOURCE DEVELOPER REPORT – 2010 Eclipse Community Survey. eclipse.org/org/press-release/20100604_survey2010.php (2011), seen: Mar. 2012
3. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. *IBM Systems Journal* 45(3), 515–526 (2006)
4. Alexander, I.: Towards automatic traceability in industrial practice. In: *In Proc. of the 1st Int. Workshop on Traceability*. pp. 26–31 (2002)
5. Alfaro, L.d., Henzinger, T.A.: Interface Theories for Component-Based Design. In: *EMSOFT* (2001)
6. Altheide, F., Dörr, H., Schürr, A.: Requirements to a framework for sustainable integration of system development tools. In: *Proc. of the 3rd European Systems Engineering Conference (EuSEC)*. pp. 53–57 (2002)
7. Anderson, K.M., Taylor, R.N., Whitehead, Jr., E.J.: Chimera: Hypermedia for Heterogeneous Software Development Environments. *ACM Trans. Inf. Syst.* 18 (July 2000)
8. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.* 28(10), 970–983 (Oct 2002), <http://dx.doi.org/10.1109/TSE.2002.1041053>
9. Aranega, V., Etien, A., Dekeyser, J.L.: Using an alternative trace for QVT. *Electronic Communications of the EASST* 42 (2011)

10. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. pp. 19–33. «UML» '01, Springer-Verlag, London, UK, UK (2001), <http://dl.acm.org/citation.cfm?id=647245.719475>
11. Atkinson, C., Kühne, T.: Model-driven development: A metamodeling foundation. *IEEE Softw.* 20(5), 36–41 (Sep 2003), <http://dx.doi.org/10.1109/MS.2003.1231149>
12. Aßmann, U., Bartho, A., Bürger, C., Cech, S., Demuth, B., Heidenreich, F., Johannes, J., Karol, S., Polowinski, J., Reimann, J., Schroeter, J., Seifert, M., Thiele, M., Wende, C., Wilke, C.: Dropsbox: the dresden open software toolbox. *Software & Systems Modeling* pp. 1–37 (2012), <http://dx.doi.org/10.1007/s10270-012-0284-6>
13. Badros, G.J.: JavaML: A Markup Language for Java Source Code. *Comput. Netw.* 33 (June 2000)
14. Barbier, F., Eveillard, S., Youbi, K., Guitton, O., Perrier, A., Cariou, E.: Model-Driven Reverse Engineering of COBOL-Based Applications, pp. 283–299. Morgan Kaufmann (2010), <http://www.sciencedirect.com/science/article/B6MH5-508779H-7/2/6b3199748873fdfa42e3a892ba1b4d19>
15. Bézivin, J.: On the unification power of models. *Software and System Modeling* 4(2), 171–188 (2005)
16. Branco, M.C., Troya, J., Czarnecki, K., Küster, J.M., Völzer, H.: Matching business process workflows across abstraction levels. In: France et al. [29], pp. 626–641
17. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In: Proc. of the IEEE/ACM International Conference on Automated Software Engineering (2010)
18. Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2004)
19. Colburn, T.: Philosophy and Computer Science. Explorations in Philosophy, M.E. Sharpe (2000), <http://books.google.dk/books?id=luF4EIMxqg4C>
20. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications (2000)
21. Dearle, F.: Groovy for Domain-Specific Languages. Packt Publishing, 1st edn. (2010)
22. Drivalos, N., Kolovos, D.S., Paige, R.F., Fernandes, K.J.: Software language engineering. chap. Engineering a DSL for Software Traceability, pp. 151–167. Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-00434-6_10
23. Egyed, A.: Automatically detecting and tracking inconsistencies in software design models. *IEEE Trans. Software Eng.* 37(2), 188–204 (2011)
24. Egyed, A., Grünbacher, P.: Automating requirements traceability: Beyond the record & replay paradigm. In: Proceedings of the 17th IEEE international conference on Automated software engineering. pp. 163–. ASE '02, IEEE Computer Society, Washington, DC, USA (2002), <http://dl.acm.org/citation.cfm?id=786769.787006>
25. Espinoza, A., Garbajosa, J.: The need for a unifying traceability scheme. In: ECMDA-TW 2005. SINTEF ICT Norway, Nuremberg, Germany (November, 2005 2005), <http://www.modelbased.net/drupal/node/19>
26. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2004)
27. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. pp.

- 307–309. SPLASH '10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1869542.1869625>
28. de Figueiredo Carneiro, G., Mendonça, M.G., Magnavita, R.C.: An experimental platform to characterize software comprehension activities supported by visualization. In: ICSE Companion (2009)
29. France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.): Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7590. Springer (2012)
30. Freude, R., Königs, A.: Tool Integration with Consistency Relations and their Visualisation. In: ESEC/ FSE Workshop on Tool Integration in System Development (2003)
31. Galletta, D.F., Henry, R.M., McCoy, S., Polak, P.: Web site delays: How tolerant are users? *J. AIS* 5(1), 0– (2004)
32. Grammel, B., Kastenholz, S.: A generic traceability framework for facet-based traceability data extraction in model-driven software development. In: Proceedings of the 6th ECMFA Traceability Workshop. pp. 7–14. ECMFA-TW '10, ACM, New York, NY, USA (2010)
33. Grammel, B., Kastenholz, S., Voigt, K.: Model matching for trace link generation in model-driven software development. In: France, R., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 7590, pp. 609–625. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-33666-9_39
34. Groenewegen, D.M., Hemel, Z., Visser, E.: Separation of Concerns and Linguistic Integration in WebDSL. *IEEE Software* 27(5) (2010)
35. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-Modelling: From Theory to Practice. In: Proc. of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I (2010)
36. Gómez, P., Sánchez, M., Florez, H., Villalobos, J.: Co-creation of models and metamodels for enterprise architecture projects. XM 2012 - Extreme Modeling Workshop (2012)
37. Halasz, F.G., Schwartz, M.D.: The Dexter Hypertext Reference Model. *Commun. ACM* 37(2) (1994)
38. Hammond, J.S., Schwaber, C., D'Silva, D.: IDE Usage Trends (02 2008), <http://www.forrester.com/Research/Document/Excerpt/0,7211,43181,00.html>
39. Hayes, J.H., Dekhtyar, A., Osborne, J.: Improving requirements tracing via information retrieval. In: Proceedings of the 11th IEEE International Conference on Requirements Engineering. pp. 138–. RE '03, IEEE Computer Society, Washington, DC, USA (2003), <http://dl.acm.org/citation.cfm?id=942807.943920>
40. Hegedüs, Á., Horváth, Á., Ráth, I., Varró, D.: Query-driven soft interconnection of emf models. In: France et al. [29], pp. 134–150
41. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and refinement of textual syntax for models. In: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications. pp. 114–129. ECMDA-FA '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02674-4_9
42. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: Proc. of the 2nd International Conference on Software Language Engineering (SLE 2009), Revised Selected Papers (2010)
43. Heidenreich, F., Johannes, J., Zschaler, S.: Aspect Orientation for Your Language of Choice. In: Workshop on Aspect-Oriented Modeling (AOM at MoDELS) (2007)

44. Henriksson, J., Johannes, J., Zschaler, S., Aßmann, U.: Reuseware - Adding Modularity to Your Language of Choice. *Journal of Object Technology* 6(9) (2007)
45. Hesselund, A.: SmartEMF: Guidance in Modeling Tools. In: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (2007)
46. Hesselund, A.: Domain-Specific Multimodeling. Ph.D. thesis, IT University of Copenhagen (2009)
47. Hesselund, A., Czarnecki, K., Wąsowski, A.: Guided development with multiple domain-specific languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) *MODELS. Lecture Notes in Computer Science*, vol. 4735, pp. 46–60. Springer (2007)
48. Hesselund, A., Sestoft, P.: Flow analysis of code customizations. In: *Proceedings of the 22nd European conference on Object-Oriented Programming*. pp. 285–308. ECOOP '08, Springer-Verlag, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-70592-5_13
49. Hesselund, A., Wąsowski, A.: Interfaces and Metainterfaces for Models and Meta-models. In: *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems* (2008)
50. Holst, W.: Meta: A Universal Meta-Language for Augmenting and Unifying Language Families, Featuring Meta(oopl) for Object-Oriented Programming Languages. In: *Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2005)
51. Jarzabek, S.: Specifying and generating multilanguage software development environments. *Softw. Eng. J.* 5(2), 125–137 (Apr 1990), <http://dx.doi.org/10.1049/sej.1990.0015>
52. Jouault, F.: Loosely Coupled Traceability for ATL. In: *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*. pp. 29–37 (2005)
53. Jouault, F., Vanhooft, B., Bruneliere, H., Doux, G., Berbers, Y., Bezivin, J.: Inter-DSL Coordination Support by Combining Megamodeling and Model Weaving. In: *Proceedings of the 2010 ACM Symposium on Applied Computing* (2010)
54. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (1990)
55. Kats, L.C.L., Visser, E.: The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) *OOPSLA*. pp. 444–463. ACM (2010), <http://dblp.uni-trier.de/db/conf/oopsla/oopsla2010.html#KatsV10>
56. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels (2008)
57. Kolovos, D.S.: Establishing correspondences between models with the epsilon comparison language. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) *ECMDA-FA. Lecture Notes in Computer Science*, vol. 5562, pp. 146–157. Springer (2009)
58. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Transformation Language. In: *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*. pp. 46–60. ICMT '08, Springer-Verlag, Berlin, Heidelberg (2008)
59. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On-demand merging of traceability links with models. (2006)
60. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: *Proceedings of the 25th International*

- Conference on Software Engineering. pp. 125–135. ICSE '03, IEEE Computer Society, Washington, DC, USA (2003), <http://dl.acm.org/citation.cfm?id=776816.776832>
61. McAffer, J., VanderLei, P., Archer, S.: OSGi and Equinox: Creating Highly Modular Java Systems. Addison-Wesley Professional, 1st edn. (2010)
 62. Meyers, S.: Difficulties in Integrating Multiview Development Systems. *IEEE Softw.* 8 (1991)
 63. Miller, R.B.: Response time in man-computer conversational transactions. In: Proceedings of the December 9-11, 1968, fall joint computer conference, part I. pp. 267–277. AFIPS '68 (Fall, part I), ACM, New York, NY, USA (1968), <http://doi.acm.org/10.1145/1476589.1476628>
 64. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: Proc. of the 31st International Conference on Software Engineering (2009)
 65. Northrop, L., Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-Large-Scale Systems - The Software Challenge of the Future. Tech. rep., Software Engineering Institute, Carnegie Mellon (June 2006), <http://www.sei.cmu.edu/uls/downloads.html>
 66. Nørmark, K., Østerbye, K.: Representing programs as hypertext. In: Lund Institute of Technology, Lund University. pp. 11–24 (1994)
 67. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, V1.1. <http://www.omg.org/spec/QVT/1.1/> (Jan 2011)
 68. Oldevik, J., Neple, T.: Traceability in model to text transformations. In: Proceedings of ECMDA Traceability Workshop ECMDA Traceability Workshop (ECMDA-TW) (2006)
 69. Østerbye, K., Nørmark, K.: An interaction engine for rich hypertexts. In: Ritchie, I., Guimarães, N. (eds.) ECHT. pp. 167–176. ACM (1994)
 70. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering. *Softw. Syst. Model.* 10 (October 2011)
 71. Pfeiffer, R.H., Wasowski, A.: Tengi interfaces for tracing between heterogeneous components. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE. Lecture Notes in Computer Science, vol. 7680, pp. 431–447. Springer (2011)
 72. Pfeiffer, R.H., Wasowski, A.: Taming the Confusion of Languages. In: Proceedings of the 7th European Conference on Modelling Foundations and Applications. pp. 312–328. ECMFA'11, Springer-Verlag, Berlin, Heidelberg (2011), <http://dl.acm.org/citation.cfm?id=2023522.2023552>
 73. Pfeiffer, R.H., Wasowski, A.: Cross-language support mechanisms significantly aid software development. In: France et al. [29], pp. 168–184
 74. Pfeiffer, R.H., Wasowski, A.: Texmo: a multi-language development environment. In: Proceedings of the 8th European conference on Modelling Foundations and Applications. pp. 178–193. ECMFA'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-31491-9_15
 75. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. *The VLDB Journal* 10(4), 334–350 (Dec 2001), <http://dx.doi.org/10.1007/s007780100057>
 76. Reimann, J., Seifert, M., Aßmann, U.: Role-Based Generic Model Refactoring. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part II (2010)
 77. Salay, R., Mylopoulos, J., Easterbrook, S.: Using Macromodels to Manage Collections of Related Models. In: Proc. of the 21st International Conference on Advanced Information Systems Engineering (2009)

78. Schulze, G., Chimiak-Opoka, J., Arlow, J.: An approach for synchronizing uml models and narrative text in literate modeling. In: France et al. [29], pp. 595–608
79. Sherba, S.A., Anderson, K.M., Faisal, M.: A framework for mapping traceability relationships. In: 2 nd International Workshop on Traceability in Emerging Forms of Software Engineering at 18th IEEE International Conference on Automated Software Engineering. pp. 32–39 (2003)
80. Shvaiko, P., Euzenat, J.: A survey of schema-based matching approaches 3730, 146–171 (2005)
81. Stallman, R.M.: Emacs the extensible, customizable self-documenting display editor. In: Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation. pp. 147–156. ACM, New York, NY, USA (1981), <http://doi.acm.org/10.1145/800209.806466>
82. Steinberger, M., Waldner, M., Streit, M., Lex, A., Schmalstieg, D.: Context-Preserving Visual Links. IEEE Transactions on Visualization and Computer Graphics (InfoVis '11) 17(12) (2011)
83. Strein, D., Kratz, H., Lowe, W.: Cross-Language Program Analysis and Refactoring. In: Proc. of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (2006)
84. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An Extensible Meta-Model for Program Analysis. IEEE Trans. Softw. Eng. 33 (September 2007)
85. Sufrin, B.: Formal specification of a display-oriented text editor. Science of Computer Programming 1(3), 157 – 202 (1982), <http://www.sciencedirect.com/science/article/pii/0167642382900144>
86. The Institute of Electrical and Eletronics Engineers: Ieee standard glossary of software engineering terminology. IEEE Standard (September 1990)
87. Voigt, K.: Semi-automatic matching of heterogeneous model-based specifications. In: Engels, G., Luckey, M., Pretschner, A., Reussner, R. (eds.) Software Engineering (Workshops). LNI, vol. 160, pp. 537–542. GI (2010)
88. Voigt, K., Ivanov, P., Rummler, A.: Matchbox: combined meta-model matching for semi-automatic mapping generation. In: Proceedings of the 2010 ACM Symposium on Applied Computing. pp. 2281–2288. SAC '10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1774088.1774563>
89. Wagner, S., Deissenboeck, F.: Abstractness, Specificity, and Complexity in Software Design. In: Proc. of the 2nd International Workshop on the Role of Abstraction in Software Engineering (2008)
90. Waldner, M., Puff, W., Lex, A., Streit, M., Schmalstieg, D.: Visual Links Across Applications. In: Proc. of Graphics Interface (2010)
91. Wilke, C., Bartho, A., Schroeter, J., Karol, S., Aßmann, U.: Elucidative development for model-based documentation. In: Furia, C., Nanz, S. (eds.) Objects, Models, Components, Patterns, Lecture Notes in Computer Science, vol. 7304, pp. 320–335. Springer Berlin / Heidelberg (2012)
92. Winkler, S., Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development. Softw. Syst. Model. 9(4), 529–565 (Sep 2010), <http://dx.doi.org/10.1007/s10270-009-0145-0>
93. Xing, Z., Stroulia, E.: Refactoring practice: How it is and how it should be supported - an Eclipse case study. In: Proc. of the 22nd IEEE International Conference on Software Maintenance (2006)
94. Yie, A., Wagelaar, D.: Advanced traceability for ATL. In: 1st International Workshop on Model Transformation with ATL. pp. 78–87 (2009)

Language-Independent Traceability with Lässig

Rolf-Helge Pfeiffer¹, Jan Reimann², and Andrzej Wąsowski¹

¹ IT University of Copenhagen, Denmark
`{ropf,wasowski}@itu.dk`

² Technische Universität Dresden, Germany
`jan.reimann@tu-dresden.de`

Abstract. Typical programming languages, including model transformation languages, do not support traceability. Because of that, applications requiring inter-object traceability implement traceability support repeatedly for different domains. In this paper we introduce a solution for generic traceability which enables the generation of trace models for all programming languages compiling to Virtual Machine (VM) bytecode by leveraging automatically generated observer aspects.

We implement our solution in a tool called *Lässig* adding traceability support to all programming languages compiling to the Java Virtual Machine (JVM). We evaluate and discuss general feasibility, correctness, and the performance overhead of our solution by applying it to three model-to-model transformations implemented in Xtend, Java, and Groovy. Our generic traceability solution is capable of automatically establishing complete sets of correct trace links for transformation programs in various languages and at a minimum cost. Lässig is available as an open-source project for integration into modeling frameworks.

1 Introduction

Model-driven Software Development (MDSD) relies on use of models to design, construct and maintain software systems. Many models in this process are related by various semantic relations. For example, in generative setups, model transformations automatically convert models into other models.

Strict quality management processes usually require that a project can identify and retrieve these relations. This ability is known as *traceability* and the stored relations are known as *trace links* which chronologically interrelate uniquely identifiable entities along a set of chained operations [21].

Co-evolution of related artifacts is a major challenge, especially for systems containing related models expressed in multiple languages [23]. If a model is modified, other related models need to be adapted accordingly. It is hard for developers to identify the affected artifacts. Trace links keep this information. Thus, automatic tracing of object relations via corresponding transformations, would allow to dramatically improve tool support for co-evolution of multi-language software systems.

Strictly speaking, traceability concerns not only the links between models, but also relations between other artifacts, so for example between models and

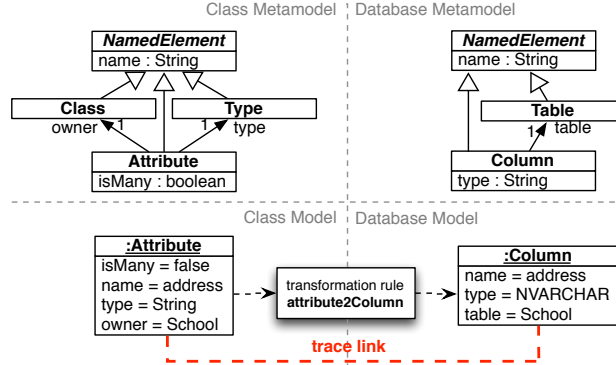


Fig. 1. Transformation of a class model attribute into a database model column

```

1 def Column create c: factory.createColumn() attribute2Column(Attribute a) {
2   if(!a.isMany) {
3     c.name = a.name
4     c.table = a.owner.class2Table()
5     c.type = a.type.name.toDbType()
6   }
7 }

```

Listing 1. A model transformation rule in Xtend transforming **Attributes** to **Columns** and causing a trace link between such instances

code, or between documentation and code. In this paper, we follow a simplifying assumption of Bézivin that *everything is a model* [3]—we discuss traceability between models, but our techniques are applicable to all the artifacts that can be represented by models, which includes code and documentation. Indeed, we believe that all kinds of development artifacts in contemporary software systems can be represented as models [13,22].

Consider an example transformation converting a class model into a database model; more precisely, the transformation of a class attribute to a database table column. Figure 1 shows the input and output languages of this transformation, while Listing 1 presents the transformation rule expressed in the popular Xtend language.³ Unfortunately, Xtend is one of the languages, which does not maintain any explicit link between the input instance of **Column** and the output instance of **Attribute**.

A similar problem appears for transformations implemented in Java. Listing 2 shows an excerpt of `org.eclipse.emf.ecore.impl.EClassImpl` class from the Eclipse Modeling Framework [28] (EMF), which locates `EObjects` based on fragments of unified resource identifiers. Here, no trace link is kept between the input instance of a `String` and the output instance of an `EObject`.

³ <http://xtend-lang.org/>

```

1 public EObject eObjectForURIFragmentSegment(String uriFragmentSegment) {
2     EObject result = eAllStructuralFeaturesData == null || eOperations != null && !eOperations.
        isEmpty() ? null : getEStructuralFeature(uriFragmentSegment);
3     return result != null ? result : super.eObjectForURIFragmentSegment(uriFragmentSegment);
4 }

```

Listing 2. A Java method causing a trace link between `String` and `EObject` instances

Trace links are usually not maintained automatically by transformation programs, since traceability is not a first class concern in most languages used for implementing transformations. Only few languages, such as the Atlas Transformation Language [6] (ATL) or the Epsilon Transformation Language [16] (ETL), automatically establish traces between the source objects and target objects of a transformation. So far, adding traceability to a transformation language has required deep insight into design and advanced language implementation skills. It could not be done orthogonally, in a language independent manner, and clearly not by language users (as opposed to language designers and implementers). Today, if traceability support is required either the system needs to be implemented in a programming language with built-in traceability support or tracing has to be added to relevant methods or transformation rules. The former is not suitable for legacy systems, as it would require reimplementation. The latter misses the opportunity to reuse application independent functionality, and pollutes business logics with it.

Our goal is to overcome the aforementioned problem by provision of generic traceability support for any (transformation) program implemented in a programming language compiling to bytecode of a virtual machine. The contributions of this paper are:

- Design of a generic aspect-based model-driven solution to support traceability for all programming languages compiling to bytecode of a virtual machine.
- Lässig, a prototypical open-source implementation⁴ of our solution for programming languages compiling to JVM bytecode.
- Identification of heuristics, which determine program structures creating trace links and discussion of extensions and alternative heuristics.
- Evaluation of our generic traceability solution by applying Lässig to three model transformations, each implemented in Xtend, Java, and Groovy.

The evaluation shows that the automatically established trace links are correct and complete. The obtained set of traces is similar to the one registered for ETL transformations, using dedicated traceability support. Finally, the additional runtime overhead for using the generic traceability approach is rather moderate. We hope that Lässig is of interest for tool builders and vendors allowing lightweight integration of traceability into modeling frameworks.

We proceed as follows. Section 2 presents the architecture of our generic traceability solution. We discuss the prototype implementation (Sect. 3), evaluate

⁴ <http://svn-st.inf.tu-dresden.de/websvn/wsvn/refactory/trunk/Lässig/>

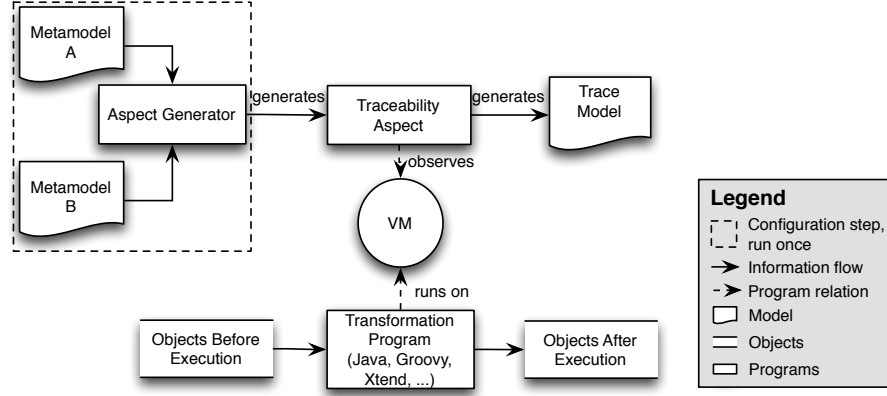


Fig. 2. Architecture for generic traceability

the idea (Sect. 4) and discuss the evaluation results (Sect. 5). Finally, we survey the related work (Sect. 6) and conclude with a sketch of future work (Sect. 7).

2 The Solution

2.1 Architecture

Previous work [11,32] argues to support traceability generically in existing transformation languages or frameworks by abstraction over particular transformation languages. With the help of such an abstraction trace links can be established generically for different transformation languages. We generalize this claim and argue that traceability can be generically added to arbitrary programming languages by relying on a common representation. Bytecode executed on VMs can serve as such a common denominator. Adding traceability to the common representation, uniformly integrates this orthogonal language feature to all programming languages compiling to the same VM.

Figure 2 illustrates the architecture of our solution for generic traceability. We use two metamodels to parametrize a code generator creating traceability code. The metamodels contain all classes for whose instances trace links should be established. In a model transformation scenario, as demonstrated in Sect. 4, the metamodels are readily available as model transformations are specified on top of them.

We use aspect weaving to instrument the transformation code with the traceability code. Recall the transformation rule in Listing 1. The rule creates a trace link between an object *a* of type **Attribute** (the rule’s argument) and an object *c* of type **Column** (return value) as properties of *a* are, with some modifications, assigned to properties of *c*. The result of the assigned property values is depicted in Fig. 1. The concern of traceability could be introduced

to the rule by inserting `tracemodel.addLink(a,c)` in line 6 (assuming a global `tracemodel`). To generate a complete trace model similar directives need to be added to every transformation rule.

Obviously, the concern of traceability is a cross-cutting concern [14,15,29] as it requires similar directives to be introduced in every transformation rule in any language in any domain that should support traceability. Such cross-cutting concerns are effectively handled by aspect-oriented programming (AOP) [17], encapsulating recurring code in aspects. The aspects can be woven anywhere the concern is required. This is the reason why our solution has an aspect-oriented architecture. A traceability aspect is generated once, for each pairwise combination of metamodels. The traceability code within the aspects is conceptually similar to the previous example. The aspects, the effect of combined metaclasses, and the aspect generation is detailed in Sect. 2.2.

Whenever programs transforming model elements of the types specified in the parametrization phase are executed on the VM, the traceability aspect is woven into the transformation's bytecode. There, trace links between transformed objects are automatically established and maintained in a trace model. We can say, that a traceability aspect *observes* the VM for transformations interrelating objects. Obviously, the created trace models maintain trace links created by programs of arbitrary domains implemented in arbitrary languages. Note, that the obtained trace model can be partial or incomplete, due to undecidability of program termination.

2.2 Heuristics for Traceability Aspects

In the realm of object-orientation, all development artifacts and their contents are objects, since these entities are *uniquely identifiable entities*. In model-driven software development (MDSD) all artifacts of the development life cycle can be considered *models* and their contents are *model elements* [4,18,24], all of which are objects again. In this paper, we use the terms *object* and *model element* synonymously.

Our solution provides traceability at object level. Consequently, all parametrizing *abstract* metaclasses in the metamodels are not considered as traceable. Therefore, no traceability code is generated for them.

Our solution relies on the following two heuristics to automatically establish trace links when observing transformations.

- (i) Related objects are an argument and a return value of a transformation (a method). Additionally, both objects are not `null` after method execution.
- (ii) Related objects are both arguments of a transformation (a method) and at least one of them is modified during method execution.

The rationale for (i) is, that a method parametrized with an object and returning a non-null object likely reads the argument to return the corresponding result. Thereby, both objects are in relation and should be linked. The rationale for (ii) is similar. A method parametrized with two arguments, where after


```

1 private pointcut findMethodA(Column t1, Attribute t2) : !within(Tracer) && execution(* *(..,
    Column, .., Attribute, ..)) && args(t1,t2,..);
2 private pointcut findMethodB(Attribute t1, Column t2) : !within(Tracer) && execution(* *(..,
    Attribute, .., Column, ..)) && args(t1,t2,..);
3 private pointcut findMethodC(Attribute t1) : !within(Tracer) && execution(Column *(..,
    Attribute, ..)) && args(t1,..);
4 private pointcut findMethodD(Column t1) : !within(Tracer) && execution(Attribute *(..,
    Column, ..)) && args(t1,..);

```

Listing 3. Generated traceability aspect for transformations between class models and relational schema

execution at least one argument is modified, likely reads one object to modify the other one. The simplicity of the two heuristics is their main power—it means that establishing trace links between objects can be done based on types of these objects and a simple check of input and output parameters of a method. It remains completely independent of the complexity of the transformation itself.

These heuristics are the basis for aspect generation. Consequently, they are implemented in the traceability aspect generator (Fig. 2). A generated aspect contains four pointcuts for each combination of metaclasses in the parametrization metamodels. Recall the example of transforming a class’ attributes to table columns (see Listing 1). Listing 3 illustrates the generated pointcut definitions for `Attribute` and `Column` types. The pointcuts `findMethodC` and `findMethodD` (lines 3 and 4), implement heuristic (i) where objects of types `Attribute` or `Column` are returned or are an argument respectively. Pointcuts `findMethodA` and `findMethodB` (lines 1 and 2), implement heuristic (ii) where objects of `Attribute` and `Column` are both arguments. The aspect contains also `advise` blocks, which are not shown here, due to their size.⁵ The `advise` blocks implement the checks of heuristics (i) and (ii) for non-null objects or for modified objects. Whenever the conditions of a heuristic hold in an executed transformation, a trace link is established.

Our experiments (Sect. 4) show that the above described two heuristics are quite powerful and generate correct trace models for our evaluation cases. In the design process we have considered two alternative heuristics, which we eventually discarded. We discuss them briefly below.

Transformation Rules with Multiple Arguments of the Same Type. As described, our solution establishes links for transformation rules with two arguments or with an argument and a return value which are combinations of two types (metaclasses). An extended heuristic would allow to establish trace links for transformation rules with either many arguments of the same type or with collections of types. Referring to the metaclasses used in Listing 3 the pointcuts implementing this heuristic would look, e.g., like in Listing 4.

But we argue not to apply this heuristics as we consider such transformation rules “bad style” of programming. Transformations of collections should call transformations of single instances. The latter transformations are matched

⁵ <http://svn-st.inf.tu-dresden.de/websvn/wsvn/refactory/trunk/Lässig/dk.itu.sdg.aspect.tracemodel.generator/output/Tracer.aj> shows the entire generated aspect.

```

1 private pointcut findMethod(Column t1, Attribute t2, Attribute t3, Attribute t4) : &&
   execution(* *(.., Column, .., Attribute, Attribute, Attribute, ..)) && args(t1,t2,t3,t4
   ,..);
2
3 private pointcut findMethod(Collection t1, Attribute t2) : && execution(* *(.., Collection<
   Column>, .., Attribute, ..)) && args(t1,t2,..);

```

Listing 4. Generated traceability aspect for transformations between class models and relational schema

by heuristics (i) or (ii). A larger study applying Lässig to industrial model transformations could give an incentive to implement this heuristic.

Transformation Rules Containing Transformation Code. As discussed previously, our heuristics establish trace links based on execution of transformation rules, simply inspecting the top activation frame on the call stack. A complex transformation encoded in a single rule results in a single trace link between the top most objects. Potentially, one could obtain more information, by inspecting the entire call stack, not just the top-most environment. An aspect observing temporally related accessor calls (get and set methods) of distinct objects within the control flow of a common transformation method could identify potential trace links between the accessed objects.

Unfortunately, AspectJ does not allow to properly identify objects being transformed during a VM call stack inspection.⁶ Furthermore, such a heuristic would limit the generality of the solution, as not all JVM languages implement attribute accesses via accessor methods.

3 Lässig: An Implementation

We implement our solution in a tool called *Lässig*, which provides traceability support for all compiled programming languages executed on the Java Virtual Machine. Lässig is implemented as a set of Eclipse bundles: one bundle for the aspect generator, one bundle containing the traceability aspects, and one bundle containing the trace model itself. Lässig requires that the metamodels parametrizing the traceability aspect generator are available as EMF models. Such metamodels are most often readily available for model transformations. When adding traceability to programs in general, the metamodels need to be created. They should contain a metaclass for each traceable JVM type.

Lässig relies on Equinox Weaving for aspect weaving. The traceability aspect resides in an OSGi bundle specifying which other bundles are observed, i.e., in which classes of other bundles the aspect is woven into. We use load-time weaving, which is triggered whenever the JVM loads a class for the first time.

When a transformation from an observed bundle is executed the woven traceability code is invoked and trace links are automatically recorded in an

⁶ <http://aspectj.2085585.n4.nabble.com/Pointcuts-for-Multi-paramter-Methods-and-for-Method-Control-Flow-td4650677.html>

in-memory trace model. It can be used as a knowledge-base for tools supporting developers in coding and co-evolution, or it can be serialized as an instance of the simple trace model⁷ [20] and directly inspected by developers.

4 Evaluation

In order to evaluate the quality of generically established trace links of JVM programs we investigate the following research questions:

- RQ1** What is precision and recall of automatically established trace links with respect to the definition of *trace link* presented in Sect. 1?
- RQ2** What is precision and recall of automatically established trace links in comparison to those established by a transformation language with a dedicated traceability support?
- RQ3** What is the performance overhead, in terms of time, of the generic traceability solution applied to model transformations in different JVM languages?

4.1 Experiment Setup

To evaluate our solution we rely on three model transformations as experiment subjects: (i) from tree models to graph models (tree2graph), (ii) from class models to relational schema (class2db), and (iii) from family models to person models (family2person). All three model transformations are well known canonical examples in the modeling community. We rely on independent specifications of these transformations from other projects. The specifications of tree2graph⁸ and class2db⁹ are taken from resources of the ETL community. The specification of family2person¹⁰ is taken from the ATL transformation zoo.

These transformations are often used in teaching transformation languages, so they cover all major concepts used in transformations: rules transforming model elements, their properties, containment relations, and references. Thus, we believe that they are relevant evaluation subjects, with reasonable coverage of constructs.

We implement each of the transformations in three languages: Xtend, Groovy, and Java. Xtend is often used for model transformation implementations. It compiles to Java source code. Groovy is a dynamic programming language for the JVM. Each of them compiles to JVM bytecode. Xtend, Groovy, and Java are among the most popular languages used in practice for implementing

⁷ <http://dev.eclipse.org/svnroot/modeling/org.eclipse.epsilon/trunk/examples/org.eclipse.epsilon.examples.metamodels/SimpleTrace.ecore>

⁸ <http://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.tree2graph>

⁹ <http://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.oo2db>

¹⁰ <http://www.eclipse.org/atl/atlTransformations/#Families2Persons>

model transformations. For this reason we believe they are interesting targets for evaluation of a language-independent traceability mechanisms.

The implementations of transformations in Xtend, Groovy, and Java follow a rule-based style. For each combination of metaclasses whose instances are transformed we implement a separate method (Groovy, Java) or rule (Xtend).

As input models we use a tree, families, and class model for the respective transformation. The models contain 6, 11, and 19 model elements respectively, see Table 1.

We run each transformation in every language in five separate test suites, in which each transformation is run six times. That is, each transformation is run 30 times in total. Since our solution relies on load-time weaving, a *test suite* here means that the transformation classes are reloaded for each test suite in a new Eclipse instance because executing a transformation the first time in a test suite will result in additional runtime because the traceability aspect is woven. After the first initial transformation it is re-run five times in each test suite.

The experiment is conducted on a 2.9GHz Intel i7 Mac Book with 8GB of RAM, of which 4GB are assigned to the Java 6 virtual machine. We use AspectJ (1.7.2), Xtend (2.3.1), and Groovy (2.0.0).

The model transformations, models and metamodels, as well as the automatically established trace models are available online together with Lässig.

4.2 Absolute Quality of Generic Traceability

RQ1. What is precision and recall of automatically established trace links with respect to the definition of trace link presented in Sect. 1?

For each transformation we serialize a trace model after completion. To answer this question, we manually compare the input models and output models together with the automatically established trace models. Then, we investigate if the established trace links are correct (precision) with respect to the definition of *trace*, and if we missed some traces (recall). Correctness criteria are (i) that the linked objects exist in the input and output models, and (ii) that the associated transformation rules establishing a trace link exist and actually transform an object or parts of it into another object of the appropriate type.

Results. The numbers of established trace links when applying Lässig to the transformations are presented in Table 1. For example, for the *tree2graph* transformation Lässig establishes 7, 13, and 14 trace links for the Java, Groovy, and Xtend transformations respectively.

Table 1. Number of trace links established for transformations in different languages

	Number of Trace Links			Model Sizes	
	Java	Groovy	Xtend	Input	Output
<i>tree2graph</i>	7	13	14	6	11
<i>family2person</i>	9	9	27	11	9
<i>class2db</i>	14	14	38	19	35

Recall, the definition of a *trace* link. A trace, links objects over a set of chained operations. Thus, a trace model for a language in which transformation rules are compiled to multiple methods in bytecode is incorrect if it does not contain multiple trace links for each method on bytecode level. In our experiment we found no incorrect links in this sense (100% precision and recall).

Discussion. The reason for the differing amount of established trace links for the three transformation languages is, that compilation of transformation rules to bytecode methods is language specific. For example, closures in Groovy methods (as in *tree2graph*) are compiled to separate unfolded methods in bytecode. Similarly, Xtend transformation rules creating model elements, are compiled to two consecutive methods. One for caching and one for the actual transformation.

We conclude that Lässig does not neglect trace links but establishes trace links correctly with respect to the programming language used.

4.3 Relative Quality of Generic Traceability

RQ2. What is precision and recall of automatically established trace links in comparison to those established by a transformation language with a dedicated traceability support?

So far, we have established experimentally that trace links are correct with respect to our trace definition (extracted from existing literature on traceability). Now, we investigate if the trace links are correct in comparison to those established by a language with first class traceability support. We implement and execute all three transformations in ETL. Since *tree2graph* and *class2db* were originally implemented in ETL we reuse these transformations. We manually convert the *family2person* transformation from ATL to ETL. Subsequently, we manually compare the ETL trace links with those established by Lässig. The trace link sets obtained with ETL serve as the baseline when investigating the research question.

Besides the correctness criteria explained in Sect. 4.2, the following criteria must be satisfied: The set of established trace links must not be smaller than the set of trace links generated from ETL, i.e., there must be an injection from the set of trace links generated by ETL to the set of trace links generated by Lässig.

Table 2. Number of conceptual trace links established for transformations in different languages compared to ETL trace links

	No. of Conceptual Trace Links				Model Sizes	
	ETL	Java	Groovy	Xtend	Input	Output
<i>tree2graph</i>	6	7	7	7	6	11
<i>family2person</i>	9	9	9	9	11	9
<i>class2db</i>	14	14	14	14	19	35

Results. The results of this experiment are presented in Table 2. The first column contains the number of trace links established by ETL. For example, `tree2graph` in ETL results in 6 trace links. On the other hand Lässig establishes 7 trace links for Java, Groovy and Xtend respectively. Some languages produce more trace links than others for the same transformation due to the way they are compiled. To allow for a comparison we collapse multiple trace links from consecutively executed caching and transformation methods into *conceptual* trace links. Conceptual trace links relate to objects (disregarding JVM operations linking them) — so several links between the same objects are collapsed into a single one, if they only differ by consecutive operations on bytecode level causing the link, but all belong to the same transformation rule.

All links established by ETL are matched by links generated by Lässig (100% recall). In some cases Lässig establishes more trace links than a corresponding ETL transformation. For example the `tree2graph` transformation results in a precision of approximately 86%. For this transformation in Java, Groovy, and Xtend one false positive with respect to ETL is established respectively. For the other two transformations Lässig establishes a corresponding trace link for any ETL trace link. Thus, both precision and recall are 100% for these transformations.

Discussion. The disparity between the numbers of recovered traces by Lässig and ETL is caused by ETL implicitly transforming root model elements without an explicit transformation rule. For example, `tree2graph` in ETL consists of one transformation rule converting model elements of type `Tree` to model elements of type `Node`. The graph model’s root element of type `Graph` is generated automatically without an explicit transformation rule. That is, ETL’s trace model does not contain a trace link between two model elements of respective types `Tree` to `Graph`. On the other hand, all the Java, Groovy, and Xtend transformations consist of two transformation rules. One from model elements `Tree` to `Node` and one for model elements `Tree` to `Graph`.

Since our solution integrates traceability on bytecode level, Lässig’s trace models for Xtend transformations are always larger than trace models from ETL transformations. For Java and Groovy they might be larger, depending on the chosen transformation rules and the chosen programming style. However, the larger trace models are still correct because they contain a corresponding trace link for each trace link in an ETL trace model.

4.4 Performance Overhead of Generic Traceability

RQ3. What is the performance overhead, in terms of time, of the generic traceability solution applied to model transformations in different JVM languages?

Introducing a new concern into a software system is always associated with a cost. Traceability cannot be provided for free. To answer this question we run a controlled experiment with two factors on the same experimental subjects as before: with Lässig enabled (the first factor) and with Lässig disabled (the second factor). For each factor, we run each transformation in Xtend, Groovy, and Java

in five separate test suites, in which each transformation is run six times. Per test suite, we distinguish between transformations run for the first time (weaving of traceability aspect to transformation code) and subsequent transformation executions (traceability aspect is already woven).

Results. Table 3 provides an overview of results of time measures. For each of the three model transformations we provide the execution time in milliseconds. The speed ratio in the rightmost columns shows how much longer a transformation runs with traceability enabled compared to the same transformation without traceability. Effectively the cost of generic traceability (with respect to no traceability at all) ranges from 4% to 400%.

The only outlier in this experiment are the transformations implemented in Groovy. They are generally slower compared than those in Java and Xtend. Somewhat surprisingly, the Groovy class2db transformations on the very first class load with enabled traceability are 2% faster than without traceability.

Discussion. It is obvious that all transformations take considerably longer on class load than in subsequent runs. Also obvious is that different languages are more or less efficient in their JVM implementation. However, the large slowdowns can be observed in the time efficient implementations of Java and Xtend, where the average runtimes for each transformation with traceability enabled is always below two milliseconds. So, still with Lässig’s traceability enabled, the transformations run fast; way faster than for example the corresponding Groovy transformations. Performance of transformations is usually more important in high volume processing, and here it is beneficial that after the initial class loading, the performance usually improves.

For the measurement of Lässig’s timing properties we may not have chosen a sufficiently large number of iterations. But we think that our results, after

Table 3. Times for running the model transformations with and without traceability

	Running Times [ms]			Speed Ratio		
	Java	Groovy	Xtend	Java	Groovy	Xtend
tree2graph						
with tracing on 1st class load	32.80	869.20	12.00	1.62	1.18	3.53
after 1st class load	0.16	1.36	0.56	2.00	1.31	2.33
without tracing on 1st class load	20.20	736.60	3.40			
after 1st class load	0.08	1.04	0.24			
family2person						
with tracing on 1st class load	62.60	602.20	59.00	4.89	1.51	5.18
after 1st class load	0.16	1.20	0.32	4.00	1.11	4.00
without tracing on 1st class load	12.80	399.60	11.40			
after 1st class load	0.04	1.08	0.08			
class2db						
with tracing on 1st class load	48.80	913.60	28.20	2.05	0.98	3.92
after 1st class load	0.92	3.84	1.64	1.35	1.04	2.93
without tracing on 1st class load	23.80	931.80	7.20			
after 1st class load	0.68	3.68	0.56			

ignoring the outlier, give an indication of how much resources Lässig’s traceability mechanism consumes on top of the plain transformations.

5 Threats to Validity

Threats to Internal Validity. First, the three subject model transformations might not be representative. They are all small, ranging from one rule in `tree2graph` to eight rules in `class2db`. However, other transformations even if they consist of more rules, would not encode different transformation patterns. More importantly, Lässig’s aspects consist of two Cartesian products of the sets of metaclasses from both metamodels A and B and creates four pointcuts for each metaclass tuple—one Cartesian product for $A \times B$ and one for $B \times A$. Thus, it is not relevant how complex the transformations are as Lässig only depends on matching metaclass tuples in the pointcuts. Thus, all executed transformation rules are traced independently of their complexity or their amount. Note though that the current solution is not applicable to model-to-text transformations, and we do not claim any success there.

Second, the size and complexity of the chosen models and metamodels may be too small. But again, even though the models may be small (six to 35 model elements), they contain all typical model structures, such as containment relations, references between model classes, etc. Again, the internal complexity of models and metamodels has no influence on the solution as the generated traceability code only relies on the Cartesian products of the metaclasses involved. Other structures in the transformed models are irrelevant for identification of executed transformation rules.

Third, when implementing the model transformations in Java, Groovy and Xtend we might be biased to implement method or rule signatures which are certainly matched by the pointcuts in the generated observing aspect. We tried to minimize this risk by converting the ETL and ATL transformations consistently to the other languages, just adjusting the syntax.

Lastly, the reliability for the performance overhead measurements could have been improved, first by larger input models as they result in longer runtimes per transformation decreasing the effects startup time delays caused by Just-In-Time compilations and the garbage collector. These effects could be further decreased by executing more iterations of each transformation. Running an extensive study for performance overhead of generic traceability will be addressed in future work.

Threats to External Validity. The choice of the experiment computer and the choice of concrete language versions, may produce particularly fast results when establishing the trace models. For reproducibility we will gladly share our experiment Eclipse setup if requested.

6 Related Work

As already mentioned, some model transformation languages provide built-in traceability support. For example, ETL [16] automatically generates a trace model

via a post condition guarding a model transformation. A similar mechanism [33] establishes trace models in ATL. Also QVT [19] has built-in traceability support. Similarly to Lässig the three languages establish trace links between objects serving as arguments or as arguments and return values of transformation rules [1]. The main difference between Lässig and the previous languages is, that they implement traceability support for a particular language only. In contrast, Lässig is language independent. It applies the same traceability support to any JVM language as traceability is realized on bytecode level.

Currently, Lässig is applicable to languages compiling to Java bytecode. Interpreted languages cannot be supported generically as interpreters often obfuscate the relation between the program objects and JVM objects at the bytecode. Implementing interpreted languages via language workbenches, such as EMFText [2] or Xtext [8] and mapping them to Java, e.g., with Xbase [7], is likely the least expensive manner to let Lässig provide traceability support to such languages.

Grammel et al. [12] categorize generation of trace models into two major groups. First, by utilizing the transformation program or second, independently of the transformation program. Clearly, Lässig utilizes the transformation program by observing its execution and establishing trace links as soon as objects of interest are modified. ETL's, ATL's, and QVT's traceability mechanisms fall into the same category. The second category is well researched as *model matching* [5,12,30,31] in the modeling community or as *schema matching* [25,27] in the database community. The former matches models and metamodels, in general object graphs, to each other and whenever a certain similarity measure between sub-graphs is fulfilled trace links are automatically created. The latter is similar to model-matching, although it often incorporates semantic analysis of the schemas in addition to their structural information.

Also other works [11,32] propose a solution for generic traceability support. Both solutions rely on a generic traceability interface abstracting from concrete transformation languages. In both solutions, this interface needs to be implemented repeatedly for any language which should support traceability, which is not required by Lässig.

Jouault [14] applies a model transformation to merge traceability rules into existing ATL transformation rules. This can be seen as an aspect-oriented programming-like technique for ATL, where ATL's metamodel is the join-point model for static weaving. This is similar to Lässig, which also requires a joinpoint model for aspect weaving. But thanks to relying purely on the JVM bytecode to provide the joinpoint model, Lässig is significantly more generic than Jouault's solution. Furthermore, Lässig automatically generates the traceability code out of metamodels. That is, Lässig provides traceability with no programming effort.

Fabro and Valduriez [9] utilize metamodels to generate model transformations semi-automatically. The goal is to relieve developers of manual implementation of recurring code patterns. Lässig can be considered a domain-specific refinement of the described solution, where the restriction to generation of traceability code allows for complete automation.

7 Future Work & Conclusion

We have introduced a solution for generic traceability for languages compiling to a VM. We provide Lässig, a prototype implementing our solution for all programming languages compiling to Java bytecode. We have demonstrated that Lässig is a practical and feasible solution. It automatically establishes correct and complete trace models.

Lässig has one limitation, the granularity of the trace model, i.e., the number of automatically established trace links depends on the quality of the observed transformation code. A transformation rule implementing a complex transformation of many objects of different types results in a trace model containing only a single trace link. We do not think that this limitation is very serious. First, implementing such a “bad style” transformation in, for example, ETL results in a similarly sparse trace model and second, such a sparse model still contains correct trace links maintaining more information than available without Lässig.

We demonstrate the capability of Lässig. We show that Lässig provides traceability at very low additional cost. It is neither necessary to manually implement traceability support for different domains in different languages nor is it necessary to learn aspect-oriented programming. Instead developers just parametrize a code generator with metamodels and thereby generate traceability aspects. We have experienced that Lässig generates 100% correct trace models. Currently, the only requirement is to follow good style of writing transformations, i.e., one method or transformation rule per combination of transformed metaclasses.

In the paper, we have used one step transformations to evaluate the tools. But Lässig can handle chains of transformations as well. If all the development artifacts are projected as models, it is possible to establish sequences of trace links that span larger parts of development process (end-to-end traceability), as long as all the steps are executed in a JVM.

In future we plan to evaluate Lässig in combination with code generators, i.e., model-to-text transformations. We assume that Lässig can be applied effectively since many generators use method signatures matching `pointcut findGenMethod(Type t1): execution(String *(.., Type, ..)) && args(t1,..) pointcuts`. Such a heuristics is likely to work with Xpand generator templates in Xtend. However verifying this requires extending Lässig to keep track of locations in the generated text file.

Beyond the general traceability problem, Lässig can be used in a broader range of applications. Especially, in the area of model refactoring Lässig can be applied to provide trace links between refactored models and dependent models. For example, Refactory [26] is a generic model refactoring framework. By integrating Lässig and its trace models, Refactory could automatically apply co-refactorings to dependent models.

Acknowledgements. We cordially thank Julia Schröter and Claas Wilke for their helpful comments on an earlier version of this paper. This research has been co-funded by the European Social Fund and the Federal State of Saxony within the project #0809518061.

References

1. Aranega, V., Etien, A., Dekeyser, J.L.: Using an alternative trace for QVT. *Electronic Communications of the EASST* 42 (2011)
2. Aßmann, U., Bartho, A., Bürger, C., Cech, S., Demuth, B., Heidenreich, F., Johannes, J., Karol, S., Polowski, J., Reimann, J., Schroeter, J., Seifert, M., Thiele, M., Wende, C., Wilke, C.: DropsBox: The Dresden Open Software Toolbox – Domain-Specific Modelling Tools beyond Metamodels and Transformations. *Software & Systems Modeling* pp. 1–37 (2012)
3. Bézivin, J.: On the unification power of models. *Software and System Modeling* 4(2), 171–188 (2005)
4. Bézivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2004)
5. Branco, M.C., Troya, J., Czarnecki, K., Küster, J.M., Völzer, H.: Matching Business Process Workflows across Abstraction Levels. In: France et al. [10], pp. 626–641
6. Eclipse Foundation: ATLAS Transformation Language. <http://www.eclipse.org/m2m/atl> (Apr 2012)
7. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for Java. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. pp. 112–121. GPCE '12, ACM, New York, NY, USA (2012)
8. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. pp. 307–309. SPLASH '10, ACM, New York, NY, USA (2010)
9. Fabro, M.D.D., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. *Software and System Modeling* 8(3), 305–324 (2009)
10. France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.): *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings, Lecture Notes in Computer Science*, vol. 7590. Springer (2012)
11. Grammel, B., Kastenholz, S.: A generic traceability framework for facet-based traceability data extraction in model-driven software development. In: *Proceedings of the 6th ECMFA Traceability Workshop*. pp. 7–14. ECMFA-TW '10, ACM, New York, NY, USA (2010)
12. Grammel, B., Kastenholz, S., Voigt, K.: Model Matching for Trace Link Generation in Model-Driven Software Development. In: France, R., Kazmeier, J., Breu, R., Atkinson, C. (eds.) *Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science*, vol. 7590, pp. 609–625. Springer Berlin Heidelberg (2012)
13. Hesselund, A., Wąsowski, A.: Interfaces and Metainterfaces for Models and Metamodels. In: *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems* (2008)
14. Jouault, F.: Loosely Coupled Traceability for ATL. In: *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*. pp. 29–37 (2005)

15. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J. (ed.) ECOOP 2001 — Object-Oriented Programming, Lecture Notes in Computer Science, vol. 2072, pp. 327–354. Springer Berlin Heidelberg (2001)
16. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Transformation Language. In: Proceedings of the 1st international conference on Theory and Practice of Model Transformations. pp. 46–60. ICMT '08, Springer-Verlag, Berlin, Heidelberg (2008)
17. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning (2003)
18. Lauenroth, K., Pohl, K.: Software product line engineering - foundations, principles, and techniques, chap. 4, pp. 72–86. Springer (2005)
19. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, V1.1.1. <http://www.omg.org/spec/QVT/1.1/> (Jan 2011)
20. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering. *Software & Systems Modeling* 10 (October 2011)
21. Paige, R.F., Olsen, G., Kolovos, D., Zschaler, S., Power, C.: Building Model-Driven Engineering Traceability Classifications. In: 4th ECMDA Traceability Workshop (2008)
22. Pfeiffer, R.H., Wąsowski, A.: Taming the Confusion of Languages. In: Proceedings of the 7th European Conference on Modelling Foundations and Applications (2011)
23. Pfeiffer, R.H., Wąsowski, A.: Cross-Language Support Mechanisms Significantly Aid Software Development. In: France et al. [10], pp. 168–184
24. Pfeiffer, R.H., Wąsowski, A.: TexMo: a multi-language development environment. In: Proceedings of the 8th European conference on Modelling Foundations and Applications. pp. 178–193. ECMFA'12, Springer-Verlag, Berlin, Heidelberg (2012)
25. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. *The VLDB Journal* 10(4), 334–350 (Dec 2001)
26. Reimann, J., Seifert, M., Aßmann, U.: On the reuse and recommendation of model refactoring specifications. *Software & Systems Modeling* pp. 1–18 (2012)
27. Shvaiko, P., Euzenat, J.: A Survey of Schema-Based Matching Approaches 3730, 146–171 (2005)
28. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd Edition. Pearson Education (Jan 2009)
29. Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.M.: N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 21st international conference on Software engineering. pp. 107–119. ICSE '99, ACM, New York, NY, USA (1999)
30. Voigt, K.: Semi-automatic Matching of Heterogeneous Model-based Specifications. In: Engels, G., Luckey, M., Pretschner, A., Reussner, R. (eds.) *Software Engineering (Workshops)*. LNI, vol. 160, pp. 537–542. GI (2010)
31. Voigt, K., Ivanov, P., Rummler, A.: MatchBox: combined meta-model matching for semi-automatic mapping generation. In: Proceedings of the 2010 ACM Symposium on Applied Computing. pp. 2281–2288. SAC '10, ACM, New York, NY, USA (2010)
32. Walderhaug, S., Johansen, U., Stav, E., Aagedal, J.: Towards a Generic Solution for Traceability in MDD. In: ECMDA Traceability Workshop (ECMDA-TW). pp. 41–50 (2006)
33. Yie, A., Wagelaar, D.: Advanced traceability for ATL. In: 1st International Workshop on Model Transformation with ATL. pp. 78–87 (2009)

Appendices

A

Variability Mechanisms in Software Ecosystems: Closed versus Open Platforms – Under Submission

Variability Mechanisms in Software Ecosystems: Closed versus Open Platforms

Thorsten Berger^{1,3}, R.-Helge Pfeiffer¹, Reinhard Tartler², Steffen Dienst³,
Krzysztof Czarnecki⁴, Andrzej Wąsowski¹, Steven She⁴

¹IT University of Copenhagen, ²University of Erlangen-Nuremberg, ³University of Leipzig, ⁴University of Waterloo

ABSTRACT

Leveraging open platforms to establish vibrant ecosystems of software has received increasing attention. Probably the most successful ecosystems rose in the mobile phone domain, where the shift from closed and centrally managed systems to open and extensible platforms such as Android and iOS led to tremendous growths. Large communities have helped to virtually explode the variety of mobile applications, allowing consumers to customize their mobile phones to great degrees. This variety is achieved using *variability mechanisms*. While closed platforms manage variability centrally and do not support consumers to use arbitrary third-party contributions, open platforms foster distributed free markets of assets and provide consumer-friendly tools to use them. But what are the underlying mechanisms that sustain success and growth of these two classes of ecosystems? Intrigued by this question, we attempt to study variability mechanisms in closed and open platforms. We qualitatively and quantitatively analyze five successful ecosystems. Our key observations are that variability models work best in centralized closed ecosystems, that dependency structures are surprisingly dense in all ecosystems, and that the fast growth of open ecosystems relies on capability-based dependencies, which foster distribution, but require stable centralized vocabularies.

1. INTRODUCTION

Software ecosystems are increasingly popular for their economic, strategic, and technical advantages. From a user's perspective, ecosystems are a successful approach to mass customization: users select the desired functionality of their instance—a phone, an IDE, or an operating system—using proper tools. From an economic perspective, ecosystems enable the sharing of a commodity burden—when many companies contribute to a project—or foster new business models and markets.

Large software ecosystems approach *variability*—the diversity of systems they offer—in very different ways. Consider the Linux Kernel and the Android application platform for

mobile devices. Linux is a highly configurable system using mechanisms known from software product lines [39, 15], such as a centralized variability model, a configurator, the C preprocessor, and a complex build system. Nevertheless, it is more than just a traditional software product line—it is a software ecosystem [16]. Over 6000 individuals from over 600 companies [19] have helped to more than double the Linux Kernel code base from 3.5M to 7.9M LOC in the last six years. Android is a service-oriented architecture that also manages huge variability, but in a more compositional and open way. Users derive a concrete system (a mobile device) by selecting apps from online repositories (app stores) using an installer tool. It has no centralized variability model, but uses distributed manifest files. Android is also an ecosystem, spanning an industrial consortium developing the main platform, device providers, and a vast and vibrant market of third-party apps.

The Linux Kernel and Android represent two major classes of large, highly successful ecosystems that follow the best suited strategies for their goals and domain. Linux has a predominantly *closed platform*. It carefully controls the admission of new features into its official release and has no official facility that allows users to install third-party features. Android has an *open platform* by supporting a free market of apps. In contrast to Linux Kernel's respectable, yet controlled growth, the Android ecosystem has virtually exploded—similar to other mobile application platforms, such as iOS. Created five years ago, the Android ecosystem boasts over 650,000 apps today.

Research has addressed software ecosystems, but focused on economic, strategic, and organizational aspects [17, 32, 42], largely sidestepping technology. While ecosystems are clearly driven by business and strategic forces, it largely remains speculation which and how variability mechanisms sustain their success and growth. What are their characteristics and how do ecosystems with closed and open platforms differ? How is a mechanism related to an ecosystem's organization or to dependency structures? Are closed mechanisms even applicable in open platforms?

We study variability mechanisms in closed and open platforms by qualitatively and quantitatively analyzing some of the most successful and fastest-growing ecosystems in existence today: the eCos operating system (OS), the Linux Kernel, the Debian Linux distribution, the Eclipse IDE, and the Android platform. They all successfully facilitate and manage massive variability, approaching it from different organizational and business perspectives and using different mechanisms. We hypothesize that if we understand the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPLC '13 Tokyo, Japan

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

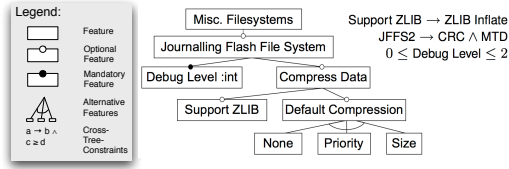


Figure 1: Feature model example [15]. The feature **Debug Level** is mandatory for the **Journalling Flash File System**, whereas **Compress Data** is optional.

causalities of choosing a technical solution, we will be able to predict how it sustains success and growth of an ecosystem, to ultimately guide development and management.

The *research objectives* of our study are to **(O.1)** identify and analyze the variability mechanisms in each class of ecosystems; and **(O.2)** discover relationships and causalities among the mechanisms and organizational structures found.

Our *contributions* comprise: **(C.1)** a conceptual framework defining key variability mechanisms and organizational structures within and across the ecosystems; **(C.2)** an instantiation of the framework with empirical data for each ecosystem; **(C.3)** a set of phenomena and hypotheses emerging from data; and **(C.4)** extracted datasets about all five ecosystems for reproducibility and further research. We report the key high-level findings of our study in this paper and provide an **online Appendix**¹ with additional statistics, details on our datasets, and the implementation of the static analysis of Android bytecode.

On a final note, our study represents the exploratory phase in the long-term process of theory building. We discover phenomena and generate testable hypotheses based on empirical evidence. Although closed and open ecosystems appear to have substantial differences, our exploratory study shows that it is possible to create a common theory behind both classes. In fact, developing models to describe ecosystems is a major research issue [26]. Our findings generate requirements for tool builders. Our datasets can serve as realistic benchmarks. For example, all of the studied ecosystems have a high density of dependencies, which tools must be able to cope with. Finally, we raise future research questions.

2. RESEARCH ISSUES

Software ecosystems are an emergent field of research and have been addressed from various perspectives. Unfortunately, research has not agreed on a definition of ecosystems from the perspective of technology yet, although they are often seen as technical constructs [12]—arguably with fluid boundaries to related paradigms, such as distributed systems or componentware. In this work, we take the view of ecosystems being extensions of software product lines of substantial size [16, 26, 25, 38].

Definitions. We define an ecosystem as a large system composed of interrelated assets developed by communities of developers upon a common technological platform. Consumers derive instances by making decisions in an automated, tool-supported process. A platform is *open* when there is explicit technical support for consumers to use third-party assets in an instance. It is *closed* when outside contributions need to be integrated into the platform with a controlled process.

¹<http://itu.dk/~thbe/ecosystems/appendix.pdf>

```

1 Package: gawk
2 Version: 1:3.1.7.dfsg-5
3 Maintainer: Arthur Loiret <aloiret@debian.org>
4 Depends: libc6 (>= 2.3)
5 Provides: awk
6 Section: interpreters
7 Priority: optional
8 Description: a pattern scanning and processing language

```

Figure 2: Excerpt of a Debian manifest containing metadata, such as package name (l. 1), version (l. 2), dependencies (l. 4–5), and categorization (l. 6–7)

Our study centers around the following research issues.

Conceptual Framework. We conjecture that closed and open platforms have implications to organizational structures, variability mechanisms, and dependencies. To compare the two heterogeneous classes, it is instrumental to define a conceptual framework that unifies related ecosystem-specific aspects with a common terminology.

Organization & Scale. Expecting different organizational structures, we aim to understand the organization of development and variability management; and the extent to which ecosystems are controlled. This analysis allows us to define the organizational context in which mechanisms are applicable. Likewise, to draw conclusions about scalability, we need to estimate scales and growth rates, which differ significantly given the diverse economies that drive ecosystems.

Variability Mechanisms. The closed platforms eCos and the Linux Kernel manage variability using mechanisms known from software product line engineering (SPLE) [39, 15]. SPLE allows to efficiently create portfolios of systems in an application domain by leveraging the commonalities and carefully managing the variabilities among them [18]. Variability models, such as feature or decision models [29, 37], are popular means to handle variability. Variability management is the corresponding discipline of taming variability-induced complexity in product lines. It comprises activities such as modeling, scoping (controlling and restricting contributions) and maintaining variability information (parameterization, dependencies, versioning). However, these activities are rather heavyweight and require advanced technical skills. We conjecture they hinder contributions, and we expect to find lean techniques in open platforms, not seen in SPLE.

eCos and Linux have variability models, which abstractly represent thousands of variabilities (drivers, processor types, scheduling algorithms, diagnostics) and the dependencies among them. Fig. 1 shows a sample feature model [29] of a filesystem. In contrast, Debian, Eclipse, and Android rely on distributed manifest files to express variability information. Fig. 2 shows an excerpt of a Debian manifest. Understanding the applicability of variability models [15, 39], and their relationship to manifests [20, 24, 36], is an important research issue in SPLE.

Asset packaging is a prerequisite for open platforms to support coarse-grained variability. We expect differences in packaging, encapsulation, and parameterization support; also in facilities for interactions. We conjecture that processes of making decisions differ. Not all platforms support derivation of a whole instance due to complexity reasons (Android handsets always come with pre-installed/pre-configured apps). Reconfiguration of an initial instance, on the other hand, requires special binding mechanisms.

Dependencies. Interactions between assets introduce dependencies that are declared in variability models or mani-

fects using constraint languages. Dependencies complicate development and maintenance, but also challenge derivation and reconfiguration tools. To understand how ecosystems cope with complexity, it is crucial to understand the dependency structures that tools and consumers manage. We expect the different representations and granularities of variability to influence dependency structures. Interestingly, Android is the only platform that does not declare dependencies and handles interactions fully dynamically. Analyzing its mechanisms helps to understand how one of the largest and fastest-growing ecosystems tackles complexity.

3. METHODOLOGY

We perform five exploratory case studies [22, 27]. They are aimed at discovering real-world phenomena and generating hypotheses from empirical evidence, which is the exploratory phase of theory building [22]. Case studies have been successfully used before in a similar context, to study open source software development (such as [35]). Generating hypotheses by analysis of case studies is a highly qualitative and interpretive process. These hypotheses need to be refuted or confirmed using other methods, such as experiments and simulations, which is subject for future work.

The major part of our analysis is qualitative. It aims at identifying mechanisms and organizational structures in the studied ecosystems and relationships among them. During the analysis, we iteratively built a conceptual framework of these mechanisms and structures, allowing us to compare their use across the ecosystems. The framework is summarized in Fig. 3 and in the concept hierarchy shown in the left-most column in the subsequent tables. We seeded the framework with mechanisms known from SPLE and then expanded to those specific to open ecosystems. Many are inspired from literature, such as [15] (variability models, dependencies), [21] (binding time/mode, openness), and [40] (interaction, encapsulation); others were added as discovered.

Quantitative analyses allow us to ask questions about occurrence and frequency of identified mechanisms. It is instrumental to identify potential correlations between qualitative concepts, such as openness of the platform, and quantitative ones, such as growth rate of an ecosystem or dependency structures.

3.1 Subject Selection Criteria

Although representativeness of case study subjects is generally not required for theory building [23], our selection strives for broad applicability of the resulting conceptual framework. We chose five successful ecosystems spanning diverse domains, ranging from feature-based systems with variability models and static compile-time binding, through component-oriented architectures specifying variability in separate manifest files associated with packages, to highly dynamic service-oriented systems with runtime resolution of dependencies between assets.

eCos is a free real-time OS for deeply-embedded applications—a domain that requires high portability, low memory usage, and small binary images. With a market share of 5–6%, it powers, among others, multimedia, networking and automotive devices [5]. Consumers of eCos are highly specialized developers of embedded systems. eCos maintains advanced tools, such as a configurator with a reasoning engine. **Linux** is a free general OS kernel targeting a much broader range of hardware than eCos. Its consumers include

Linux distributors, who customize and release specialized kernels, and technically skilled end users, who sometimes also configure, compile, and install a custom kernel. Linux also provides a configurator, but much less advanced [15] than in eCos. **Debian** is a complete OS with a large selection of applications. It is available for many hardware architectures, ranging from embedded systems to high performance computers. Its consumers are both non-technical end users and system administrators with high technical expertise. Debian provides suitable installers and configurators for beginners and experts. The **Eclipse IDE** is a foundation for highly-customizable development tools (the Rich Client Platform for building arbitrary GUI software is out of scope of this work). Although users of the Eclipse IDE are technically-skilled developers, extending the system is supported by a convenient installer. **Android** is a free OS for mobile devices, including smartphones, tablets and netbooks, that can be extended with third party applications (apps). The target consumers of Android are non-technical end users, deriving their system by installing apps with a user-friendly installer.

Even though, Eclipse is a package in Debian, and Linux is the underlying kernel of Android and Debian, we clearly distinguish these ecosystems, analyze and compare them on their own.

Linux' and eCos' platform are predominantly closed. In Linux, additions must be applied to the source tree as git branches or patch sets. "Out-of-tree" development is actively discouraged [11] and deriving such an instance not tool-supported. Exceptions are loadable kernel modules from commercial vendors. In eCos, although openness was a goal of its packaging mechanism, contributing requires programming effort. In contrast, Debian, Eclipse, and Android are open by design and offer tools to easily install extensions distributed on free markets.

3.2 Data Sources and Analysis Infrastructure

Sources are cited as we use them in the text. In the qualitative analysis, we relied on official documents such as the Debian Policy [2] and the Eclipse Development Process description [10]. We also examined tools and languages used in the subjects. For the quantitative measures, we used statically extracted data. Since analyzing whole ecosystems is infeasible given their open and uncontrolled nature, we mined substantial subsets by considering the most vibrant parts—the major distribution sources. For eCos, we considered all i386-specific and hardware-independent packages from the repository (v. 3.0). For Linux, we analyzed the x86 architecture from the 2.6.32 codebase. Debian's subset are all binary i386 packages from the 6.0 distribution. For Eclipse, we analyzed the Helios 3.6 modeling distribution together with bundles from the associated repository. For Android, we gathered nearly all available free apps from the app store over a period of 14 months.

We developed analysis tools for each ecosystem. For eCos and Linux, we partly reused our previously developed infrastructure from [15]; for Eclipse, we exploited the platform API to query information; for Debian, we analyzed the package indices used by the native installers. Analyzing Android was most challenging: we implemented static analysis techniques to identify dependencies in Android bytecode. More details, including exact dates and versions used for estimations, and all datasets are available in our online Appendix.

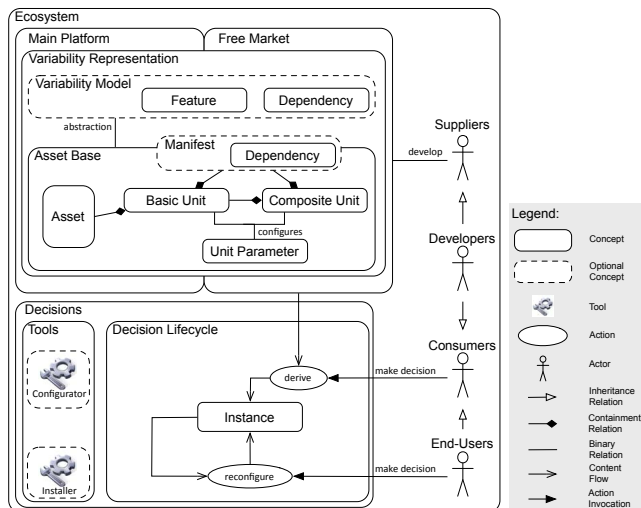


Figure 3: The Conceptual Framework

4. CONCEPTUAL FRAMEWORK

Figure 3 presents the core part of our conceptual framework. We describe it in this section, and then use it to characterize and compare the ecosystems in a uniform manner. The general framework concepts are typeset in **sans-serif** and their ecosystem-specific instantiations in *curative*.

An **ecosystem** is a universe of shared assets centered around a technical **platform**. In this universe, various roles, mainly **suppliers** and **consumers**, interact in order to develop, manage, and consume assets. More roles exist, but modeling them is out of our scope. A **platform** denotes the technical aspects of an **ecosystem**: a variability-enabled architecture, a set of shared core assets, tools, frameworks, and patterns, together with organizational and process-related concerns. Every vital **ecosystem** has a controlled central part, the **main platform**, which is managed by the **platform supplier**. **Free market** is the less-controlled, complementary part of the **ecosystem** that provides third-party assets extending the main platform. Alternative **platforms** may exist as **derivatives** of the main **platform** for specific needs. For example, Ubuntu is a Debian derivative for desktop and laptop users. Since **derivatives** do not belong to the **free market**, we ignore them in this study.

Assets are any artifacts, such as source code, binaries, media files, or documentation. Each of the studied platforms packages **assets** into **basic units**, such as *Debian packages* or *Eclipse bundles*. **Composite units**, such as *Debian meta packages*, aggregate sets of **basic units**.

Variability in the platforms has two forms: basic units can be optional, or vary inside, or both. Unit parameters, such as *properties* in Eclipse, describe variability within basic units.

An instance (e.g. a customized Linux Kernel or Android system) is a concrete system derived from the main platform and the free market by making decisions—more precisely, by selecting and configuring assets, thus, resolving variability. Usually, an instance can be reconfigured later.

Variability information (**dependencies** and **unit parameters**) is specified either within a **variability model** or in distributed manifests. Variability models are system-wide abstractions over the concrete **assets** and declare **features** and **dependencies** using a dedicated language [15]. **Features** are abstract entities that are mapped to **units** and **unit parameters**. Instead of making decisions directly on the **assets**, **derivation** is based on deciding

features. Manifests directly reflect variability information of the assets, without the ability to introduce abstractions, for example, to optimize dependency structures. Such abstraction is only partially available by introducing empty assets whose manifests aggregate dependencies, like *virtual packages* in Debian.

Each ecosystem supports derivation and reconfiguration by automated tools: configurators for the variability model-based platforms (eCos, Linux) and installers for manifest-based platforms (Debian, Eclipse, Android). Such automated tools assist consumers with intelligent choice propagation, conflict resolution, and optimization based on the dependencies. The latter are declared either among **features** within the **variability model**, or among **basic or composite units** within the manifest.

5. ORGANIZATION & SCALE

In our study, we identified the following organizational structures and ecosystem scales achieved over time.

5.1 Organization

The development and variability management are organized as follows in each ecosystem; see Table 1.

eCos' main platform is its free edition, maintained and developed by the main supplier eCosCentric and external contributors [4]. Both development and variability management are *centralized* in the main platform. We have not found reliable information about the process used for contributions. However, the main platform is controlled by a group of currently ten maintainers, which indicates that contributions have to pass their reviews. Only a marginal free market emerged on the fringe of the main platform, although eCos' packaging mechanism and its modular variability language were designed to encourage contributions. No uniform distribution channel exists for the free market.

Linux’ main platform is the mainline kernel. The variability management is *centralized*, with only a few maintainers controlling the variability model [19]. In contrast, the development is highly *distributed*, comprising thousands of developers and maintainers. However, contributions have to pass thorough reviews through the maintainer hierarchy. Although no uniform distribution channel (beyond mailing lists) outside the main platform exists, an unorganized free market with third-party modules (mostly drivers) emerged.

Debian's main platform is the central repository contain-

Table 2: Estimated Scales and Growth Rates

	eCos	Linux	Debian	Eclipse	Android
Main platform scale					
Basic Units	3948 ¹	25,861 ¹	28,232 ²	5,787 ³	83 ⁴
Features	2,859	10,415	N/A	N/A	N/A
LOC	0.9M	7.9M	762M	21.2M	1M
Free market scale					
Basic Units	>1,530 ¹	—	>15,179 ²	>1,897 ³	>651K ⁴
Features	>315	—	N/A	N/A	N/A
LOC	>279K	—	>410M	>6.9M	>1G
Growth rates					
Inception year	1999 ^(v1.1)	1991 ^(v0.01)	1996 ^(v1.1)	2001 ^(v1.0)	2008
Inception LOC	76k	10k	13M	141k	1.128M ⁵
Current LOC	1.2M	7.9M	1.2G	28.1M	1G
Growth per year	32%	39%	35%	80%	353%

¹ Files ² Packages ³ Bundles ⁴ Apps ⁵ Android OS and apps

Table 1: Ecosystem Domains and Organization.

		eCos	Linux Kernel	Debian	Eclipse	Android
Domain	Software domain	embedded OS	general-purpose OS kernel	OS & application software	software development tools	OS & applications for mobile devices
	Consumer skills	highly-technical	highly-technical	non- and technical	technical	non-technical
Organization	Main Platform	free eCos edition	mainline kernel	Debian Archive ('main' section)	yearly official platform release	Android OS and Google Apps
	Development Variability mgmt.	centralized	distributed	distributed	distributed	distributed
	Free market	centralized	centralized	distributed	distributed	centralized
	distribution channel	packages	kernel modules (drivers), patches	mostly commercial packages	bundles on update sites/market places	apps on market places
		none	none	marginal third-party repos.	Eclipse Marketplace	Google Play store

ing the official distribution. Both development and variability management are *distributed*, comprising over thousand package maintainers, who maintain packages (particularly their manifests, see Fig. 2) that are sourced from free and open source software [30]. The main platform tries to be as inclusive as possible, with little restrictions to contributors, while reviews still assure quality [2]. A free market, with mostly commercial and non-free packages in scattered third-party repositories, complements the main platform.

Eclipse's main platform is represented by the yearly releases of the IDE. It consists of independently managed projects following the Eclipse Development Process [10] and is controlled by its supplier, the Eclipse Foundation. Contributions of new projects undergo thorough reviews. Both the development and variability management is *distributed* in the main platform. Eclipse has a complementary free market, mainly represented by the Eclipse Marketplace [3] and further repositories, such as Yoxos [9] and smaller update sites for Eclipse's installer.

Android's main platform comprises the OS and pre-installed apps. While the development is *distributed*, the variability management of the main platform is *centralized* and fully controlled by Android's supplier, the Google-led Open Handset Alliance. Individual sub-projects exist, each having a project lead (typically a Google employee [1]. Contributions to the main platform are possible, but with thorough reviews. A free market is an essential goal of Android. The main distribution channel (Google Play store) is wide open to third-party contributions of arbitrary applications.

5.2 Scale & Growth

We conservatively estimated main platform and free market sizes, as shown in Table 2 and detailed in the Appendix.

eCos has the smallest main platform, comprising only 502 packages and a marginal free market. **Linux** is much larger, given its support of a much wider variety of hardware. We could not estimate the possibly large, but unorganized free market. **Debian** has the most inclusive and largest main platform in our study, given that it is relatively easy to contribute new packages. As a result, the free market [8] is comparatively small, half the size of the main platform. **Eclipse's** main platform and free market are both of medium size, compared to the others. The main platform (Helios 3.6) is three times larger than the two free market repositories [3] and [9]. However, the whole free market might be significantly larger, as the ecosystem is heavily scattered. **Android** is an ecosystem with a free market that is over 1,000 times larger

than the main platform [6]. The main platform, which is relatively closed and strongly filters outside contributions, is very small with only 83 apps.

We estimated yearly growth rates of our subjects—Table 2—by fitting an exponential growth function to the size difference between initial release and current state. Not surprisingly, these confirm that platforms with intended free markets (Eclipse, Android) grew considerably faster than those that focus on the main platform (eCos, Linux, Debian).

6. MECHANISMS

In our study, we identified and characterized variability mechanisms both from a technical (how instances vary) and a consumer perspective (how and when consumers make decisions). Table 3 summarizes our observations.

6.1 Variability Representation

Variability Model and Language. Linux' and eCos' feature-model-like [29, 15] variability models are declared in the Kconfig and CDL language [15]. CDL was designed to encourage contributions and allows a modularized specification of models, distributed over individual eCos packages.

Manifest and Schema. Debian, Eclipse and Android declare their variability information in text- or xml-based manifests inside packaged basic units and maintained together with it. For brevity, we do not report further details on the modeling languages and manifest schemas, but refer to previous work [15], Appendix, and implementation of our analysis infrastructure.

Units, Unit Parameters, and Features. In eCos, basic units are source files with internal variability controlled by preprocessor symbols (unit parameters) and realized via `#ifdef` statements. Composite units are packages, which are aggregations of source files, test cases, or other resources, together with a variability model of the package. eCos' configurator aggregates partial models into a single whole, depending on the set of loaded packages. A feature-to-code mapping (declared in the model) connects features with implementation assets; it is used to derive a concrete instance. **Linux** has two types of basic units: (1) source files with preprocessor symbols (unit parameters) as in eCos, and (2) loadable kernel modules that extend Linux at runtime. No concept for composite units exists. The feature-to-code mapping resides in the build system [14]. **Debian's** basic units are packages—file archives with helper scripts and a manifest. Composite units are realized by meta packages, whose purpose is to aggregate other packages via dependencies.

Table 3: Variability Mechanisms

	eCos	Linux Kernel	Debian	Eclipse	Android
Variability Representation	Variability model	feature-model-like	feature-model-like	N/A	N/A
	Features	packages, components, options, interfaces	configs, choices, menuconfigs, menus	N/A	N/A
	Language	CDL	Kconfig	N/A	N/A
	Manifest (Schema)	N/A	N/A	y (textual DSL)	y (XML-based DSL)
	Asset Base			y (OSGI manifest)	
	Basic units	files	files, kernel modules	bundles	apps
Decisions	Composite units	packages	meta packages	features	N/A
	Unit parameters	preproc. symbols	debconf options	properties/ preferences	N/A
	Grouping and categorization	variability model	variability model	market place categories	app store categories
	Decision lifecycle	derivation	derivation, reconfig.	reconfiguration	reconfiguration
	Decision binding	static	static & dynamic	dynamic	dynamic
	Interface mechanisms	C header files	C header files	package-specific	explicit public components, predef. data formats
Encapsulation	Interface specification	documented interfaces for components, e.g., drivers	documented interfaces for components, e.g., drivers	package-specific, documented policies for some domains	explicit public interfaces defined by OSGI manifest
					explicit public components, predef. data formats
Interactions	Managed by runtime system	N/A	N/A	N/A	Equinox OSGI
	Interaction mechanisms	static linking	static & dynamic linking	dpkg-triggers, documented policies	class reference, services, extension points
	Interaction binding	early static	early static & dynamic	not specified	late static & dynamic

The tool `debconf` realizes unit parameters and is used by scripts to configure the packaged software. It prompts users to make configuration choices during package installation. **Eclipse’s** basic units are OSGI bundles—dynamically loadable modules tying together artifacts such as Java classes, images, configuration files, and metadata. Bundles run in a virtual machine. Unit parameters are provided by several mechanisms, including the preference store and configuration admin service. Composite units, called “features”, aggregate multiple bundles with branding and update information. **Android** is composed of apps—individual application programs representing basic units. Most apps run in a virtual machine (Dalvik). Android has no concept of composite units, and no dedicated mechanism for unit parameters. Apps read global settings from a special class or a data storage.

Grouping and Categorization. To organize units and features, eCos and Linux use the hierarchy of their variability models [15], whereas the open platforms rely on diverse, often informal and distributed categorization systems. These are integrated in the Eclipse Marketplace and Google Play. Debian offers community-driven categorizations: Debtags [43].

6.2 Decisions

The most distinguishing characteristics of decisions we identified are their lifecycle, binding, and tool support.

Decision Lifecycle. A *decision lifecycle* characterizes when and how end users decide the presence or absence of units—whether they derive an instance from scratch, or only reconfigure one. In Linux and eCos, users derive an instance using configurators. In the other ecosystems, end users normally reconfigure an initial instance provided by the supplier. Eclipse comes in one of eleven pre-instantiated editions. An Android instance is delivered with the mobile device. A Debian end-user usually installs a minimal system before it can be reconfigured by installing and removing packages.

Decision Binding. Decisions can have different *binding mode* and *binding time*. Binding mode characterizes whether a decision can be changed. For eCos and Linux, it is static, since these systems require to re-derive the instance for changes. However, Linux also allows late dynamic decision binding by means of loadable kernel modules. Debian, Eclipse and Android are dynamic as they allow basic units or composite units to be installed and removed at run-time.

6.3 Encapsulation & Interactions

Encapsulation. Our closed platforms offer no encapsulation concepts beyond C header files; only implementation guidelines for interfaces of loadable kernel modules exist in Linux. In Debian, interfaces are solely package-specific; however, Debian has policies for some domains, such as Java libraries or Emacs extensions. Eclipse encapsulates all classes and resources in the bundle; public functionality—Java packages, OSGi service interfaces, extension points—must be declared in the manifest. Android apps can provide public components that are described and advertised to other apps with intent filters (see Sect. 7.1).

Interaction mechanisms and binding. Interactions among basic units requires identifying and binding the concrete target. eCos and Linux use static interaction binding; technically, all selected basic units are linked into a single binary image. Linux also supports late dynamic interaction binding through kernel modules. In Debian, interaction binding is mostly package-specific, however, several policy documents prescribe guidelines for interaction in some domains. As a major difference, the open platforms Eclipse and Android both provide a runtime system with full control over interactions. Eclipse offers three facilities: direct class referencing, extension points and services. Except for services, interaction targets are bound late but statically—due to Java classloader restrictions. Android provides a purely dynamic

facility for interaction with its intent mechanism. The interaction target—specified by parameters of an intent—is continuously reevaluated at runtime and could easily change when apps are exchanged or reinstalled.

7. DEPENDENCIES

In our study, we identified the following mechanisms to express dependencies and resulting dependency structures. Table 4 summarizes the core characteristics.

7.1 Specification, Semantics & Expressiveness

eCos and Linux declare dependencies among features in their variability models. Due to their high level of abstraction, variability models allow flexible specification of intricate dependency structures. This flexibility comes at the cost of maintaining additional artifacts—variability model [31] and feature-to-code mapping [14], which need to be coordinated. Debian’s and Eclipse’s specification of dependencies among basic units in manifests is more direct, but less flexible. Android approaches the problem entirely dynamically. No static specifications of dependencies among apps are used. Apps can only declare to be open for interaction by setting a flag, or defining an intent filter, stating that the app can handle specific service requests (Appendix Sect. 5). Android’s installer does not enforce dependencies statically; apps handle unsatisfied dependencies at runtime.

We identified a special class of dependencies in each ecosystem: dependencies on capabilities, as opposed to direct dependencies. Capabilities are abstractions over functionality provided by one or more units or features. For example, the capability to open URLs is provided by multiple web browsers. In Fig. 4, we detail the roles assumed by units and capabilities in dependencies: *providing* and *depending* on other units and capabilities; see Appendix for additional explanation. Some platforms provide explicit capability constructs, such as CDL interfaces in eCos and virtual packages in Debian (1.5 in Fig. 2). Eclipse uses names of Java packages as capabilities. Android provides the richest specification via *intent filters*. These form a simple DSL or an ontology, which can be used by contributors to increase reuse. Interestingly, the community launched repositories with additional vocabulary [7]. Finally, Kconfig has no explicit capability construct, but some features in the Linux model play this role.

We also classified the dependencies by their semantics (modality). Hard dependencies must always be satisfied. Soft dependencies represent suggestions or defaults. Table 4 shows the keywords in the variability languages/schemas declaring a certain type of dependency.

The constraint languages for declaring dependencies differ in expressiveness. eCos’ CDL supports most operators of a modern programming language [15]. Kconfig supports any Boolean dependencies and equality on strings and numbers. Notably, it uses three-state logic for dealing with loadable ker-

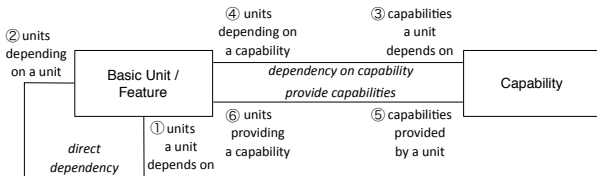


Figure 4: Dependency Metamodel

Table 5: Dependency Statistics

	eCos	Linux	Debian	Eclipse	Android
Ecosystem subset					
Basic units	1023 ¹	10,326 ¹ 2,814 ²	28,232 ³	2,105 ⁴	281,079 ⁵
Features	1,244	6,308	N/A	N/A	N/A
LOC	302K	4.3M	782M	7.8M	433M
LOC per basic unit [†]	295	416	27,699	3,705	1,539
Basic units/features					
W/ dependencies	99%	100%	96%	89%	69%
direct①	99%	100%	95%	81%	14%
to capability③	8%	N/A	24%	27%	68%
W/ depending units②	42%	31%	62%	57%	N/A
Providing capability⑤	10%	N/A	13%	80%	100%
Dependencies ①③					
# per basic unit/feature [‡]	1	2	4	6	1
Capabilities					
W/ depending units④	44%	N/A	54%	11%	N/A

¹ Files ² Loadable modules ³ Packages ⁴ Bundles ⁵ Apps
[†] Average [‡] Median ○ Numbers refer to our meta model (Fig. 4).

nel modules [15]. Debian supports any Boolean dependencies among packages and comparisons on version ranges. Exclusions are specified via conflicts and breaks, and defaults via recommends. Debian provides even more modalities, mainly to drive package update, replacement, and removal processes. Eclipse supports implications, conjunctions, and version comparisons, but lacks negations and disjunctions. It is not easily possible to exclude bundles or declare alternatives.

7.2 Dependency Structures

To study dependency structures, we computed cardinalities for all association ends in our dependency meta-model (see Fig. 4). Detailed diagrams are available in Appendix (Sect. 6). **Connectivity.** The connectivity of the dependency graph indicates the proportion of units and features for which dependency information has to be maintained. The number of units or features having direct (① in Fig. 4) and capability-based (③) dependencies is surprisingly high, regardless of platform openness. The highest is observed in Linux, where almost all features reference others, and in eCos, where it reaches 99%. These numbers are high partly because every non-root feature implies its parent in the model hierarchy. Still, many features (30% in eCos, 85% in Linux) declare cross-hierarchy dependencies. These are known to critically influence hardness of reasoning both for configuration tools [33] and for users, by introducing intricate implications of choices. Notably, the high percentage of cross-hierarchy dependencies in Linux challenges assumptions about the complexity of models made before [33]. Finally, in the open systems, most basic units also participate in many dependencies: Debian has the highest amount with 96%, followed by Eclipse with 89%, and Android with 69%.

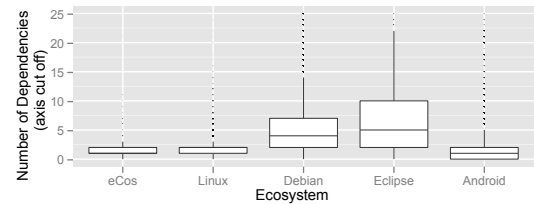


Figure 5: Dependencies per feature or basic unit

Table 4: Dependency Mechanisms

	eCos	Linux Kernel	Debian	Eclipse	Android	
Dependencies	Direct dependency					
	Target	features	features	basic units	basic units	
	Types (hard/ <i>soft</i>)	hierarchy, requires, active_if, default, calculated	selects, prompt condition, default	depends, pre-depends, recommends, breaks, conflicts, suggests, enhances	Require-Bundle	explicit intent
	Capability-based dependency					
	Target	CDL interfaces	N/A	virtual packages	Java packages	intent filters
	Types	same as direct dep.	N/A	same as direct dep.	Import-Package	implicit intent
	Common vocabulary	N/A	N/A	N/A	via API	via API
	Provide capabilities	implements	N/A	provides	Export-Package	via intent filter
Expressiveness	any Boolean; arithmetic & string operations	any Boolean; number/string equality	any Boolean; version comparison	conjunction & implication; version comparison	N/A	

Density. The density of the dependency graph indicates how much dependency information needs to be maintained per unit or feature. To assess it, we considered the number of dependencies per unit or feature, see Fig. 5.

Except Android, the open platforms have more dependencies per unit than the others per feature. Interestingly, there are many outliers, such as an app with 96 dependencies in Android, a package with 323 dependencies in Debian, and a bundle with 419 dependencies in Eclipse. Some Debian outliers have many soft dependencies (modalities like *suggests* and *recommends*), although most dependencies are hard in Debian (Appendix Sect. 6.2.2). While many Eclipse outliers are caused by many Java package imports (capability-based dependencies), most dependencies are direct ones on bundles (Appendix Sect. 6.2.1).

We also investigated the reverse dependencies (② and ④ in Fig. 4). If units have many, they are particularly hard to evolve, since dependencies on them are not specified directly together with the unit. Evolution of such units can break dependencies easily. We obtained numbers for all systems except Android, due to limitations of our static analysis (Appendix Sect. 5.3.2). We find that the open ecosystems have higher proportions of units being referenced (Debian: 62%, Eclipse: 57%) than the others for features (eCos: 42%, Linux: 31%). We further notice that, particularly in Debian, 44% of packages depend on `libc6`, whereas in the other subjects, we not observed such an outstanding central unit or feature.

Capabilities Interestingly, the percentage of units or features with direct dependencies drops significantly from eCos with 99% to Android with only 14%. The opposite is observed for capability-based dependencies, which rise from 8% in eCos to 68% in Android. Dependencies on capabilities increase variability (more than one web browser can fulfill the open URL capability), decrease coupling (an app no longer depends on a specific browser), improve flexibility and communication among developers, since capabilities indicate that specific functionality is available.

8. PHENOMENA & HYPOTHESES

The exploratory methodology of our study and the small set of subjects prevent us from drawing statistically significant conclusions. Thus, we synthesize our findings as phenomena (facts that hold about our subjects), hypotheses (proposed explanations), and interesting research directions indicated by data.

Mechanisms. Mechanisms in the closed platforms are characterized by variability models, expressive dependency facilities, early static decision binding, and enabling fine-grained basic units. In the open platforms, we expectedly found easy-to-use mechanisms that promote contributions: uniform distribution channels within a free market, asset packaging, manifests, runtime resolution of dependencies, highly dynamic runtimes, and interface mechanisms. We found that a clear difference between manifests and variability models is that manifests are always fully distributed, created as individual units with bilateral relations to other manifests, and used and evolved as individual units. In contrast, variability models, even if split over multiple files, are created around a central hierarchy, and used and evolved as a whole. Variability models with their rich languages and the arbitrary asset mapping enable fine-grained mechanisms and almost arbitrary cross-cutting contributions.

Organization. The existence of variability models correlates with centralized variability management in our subjects (Tables 1, 3). Although many developers can contribute code and changes to the models, a core team must watch the impact of changes. Notably, we made the same observation in our recent survey of industrial variability modeling [13].

HYPOTHESIS 1. *A centralized variability model is fragile and has to be managed centrally by a small team.*

This hypothesis explains the absence of variability models in open platforms. But whether a distributed variability model could facilitate distributed variability management remains an interesting research question. There is so far no empirical evidence, since eCos has a distributed (via eCos packages) variability model described in a rich language, but failed to create a vibrant ecosystem of assets with distributed variability management.

Furthermore, the organizations of development and variability management are independent. Linux’ development process is highly distributed, while ensuring centralized variability management. This observation challenges claims [41, 42, 36] that only distributed variability management is suited for distributed (or composition-oriented) development.

We observe different processes for contributions to the ecosystems. The closed platforms strongly filter contributions using heavyweight processes including manual reviews, while the open platforms offer lean processes through their distribution channels. In turn, open platforms only allow coarse-grained contributions, with Eclipse and Android ad-

ditionally relying on highly dynamic runtimes (virtual machines) and interface mechanisms. The following hypothesis strives to explain this relationship:

HYPOTHESIS 2. *Closed platforms must compensate missing guarantees of encapsulation and interface mechanisms with heavyweight processes and strict policies to assure quality.*

Dependencies. One of our most interesting findings were capability-based dependencies. We are not aware of SPLE literature describing such dependencies nor any academic language supporting them. Their widespread use indicates two important requirements for open platforms: (1) language support and (2) management of centralized stable vocabularies. The platforms with higher proportions of capability-based dependencies grow significantly faster. Although there are many reasons for growth, such as business context, sheer manpower of a vibrant community, or huge market demand, we hypothesize that:

HYPOTHESIS 3. *A high amount of capability-based dependencies positively influences growth.*

For significant impact, capabilities should not just be labels (Debian, Eclipse), but described in a rich DSL, similar to intent filters (see Appendix) in Android. Further, the fast-growing ecosystems ($\geq 80\%$ a year) rely on dynamic decision binding and service-oriented composition mechanisms, like runtime-service lookup, download and installation. We conjecture that these are essential for fast growth.

Variability models appear to impact dependency structures. In closed platforms, the median of *declared* dependencies per feature is lower than per basic unit in the open platforms. This phenomenon is seen across the subject spectrum without Android, which does not declare dependencies. Variability models let developers optimize and collapse implementation-level dependencies, while the coordination cost for these activities in a distributed setting may be too high. Still, there can be other reasons for the lower number of dependencies in the systems with variability models, so this controversial observation requires further research.

Variability Management Research. Recall that variability management aims at taming variability. The accumulation of new activities of very different nature in open platforms calls for a new discipline in variability research: Variability Encouragement. Verifying its underlying activities, such as maintaining capability vocabularies and processes with little restrictions on contributions, and relating them to known software engineering practices constitutes follow-up research.

9. THREATS TO VALIDITY

External Validity. We have purposely selected heterogeneous ecosystems to increase the generality of our conclusions. Although being among the largest and most successful, they might not be representative for each class. We mitigate this threat by using an exploratory research method: instead of testing hypotheses, we record observed phenomena and generate hypotheses. We limit data sources to reliable documents, freely available source code, and tools. Confronting our results with other data, such as interviews, would be valuable future work.

Internal Validity. In the quantitative analysis, some numbers are estimated using interpolations and safe assumptions (lower bounds) and may be inaccurate. We address this threat by providing data sets, details on data sources,

additional diagrams, and our analysis tools in an Appendix. Dependencies seem difficult to compare between closed and open platforms. Furthermore, Android does not declare dependencies; it is not clear whether our extracted dependencies are comparable to declared ones. In fact, it is subject of ongoing research whether these are generally comparable or not. Therefore, we avoid comparing dependency numbers for Android to other systems. Further, the Debian and Eclipse analysis disregards dependencies on particular unit versions, which may impact accuracy. We believe this simplification is acceptable, as such dependencies are mainly used to assist system upgrades, not addressed by us. Still, all these numbers indicate scalability requirements for tools. In that sense (algorithmic hardness), they are useful standalone and, to a large extent, comparable.

10. RELATED WORK

Barbosa et al. [12] review publications on software ecosystems using a systematic mapping study. They confirm that ecosystem are technical constructs, related to open source software and SPLE. Bosch [16] contributes a taxonomy of software ecosystems, which is applicable to all our subjects. He emphasizes economical incentives that ecosystems offer, such as value and attractiveness for users, collaboration, and the practical scalability of ecosystems. McGregor [32] discusses transactions between organizations participating in an ecosystem, their strategic and economic advantages, and possible risks, for example resulting from an unplanned scope. Gulp et al. [42] analyze development processes used by Eclipse and Debian and show that their development can be performed successfully using practices not seen in SPLE. Messerschmitt et al. [34] characterize software ecosystems according to their context. They identify stakeholders and their interests and views on software systems. All these works aim at understanding the various forms of ecosystems that arose. We extend this research by empirically analyzing two classes of ecosystems (closed and open platforms) that allow *automatic* derivation of instances. Furthermore, we look closer into technical aspects, and relating them to organizational structures.

Jansen et al. [26] present a research agenda for software ecosystems, proposing to study ecosystems such as MySQL/PHP, Microsoft Windows, and iPhone apps. We deliver on this agenda by investigating similar systems. They announce the *characterization and modeling* of software ecosystems as a main challenge. Kabbedijk et al. [28] study *defining characteristics* of open-source ecosystems, using Ruby, which also uses manifests, as a reference. They focus on the role of developers and basic units (gems). Our framework uses different study subjects and includes more concepts. Seidl et al. [38] model the evolution of assets in a software ecosystem. They develop a metamodel focusing on versioning and versioned dependencies and discuss possible analysis on top of metamodel instances. Their focus is different from ours, we focus on mechanisms and ecosystem organization. Cosmo et al. [20] and Galindo et al. [24] show that subsets of variability models can easily be converted into a Debian package structure with manifest files and back. Schmid [36] compares Debian package manifests and Eclipse bundle manifests with FODA feature models [29]. We see our study as an extension to this research. However, our work is both qualitative and quantitative, analyzing large ecosystem subsets. We also develop hypotheses that explain interesting observations.

11. CONCLUSIONS

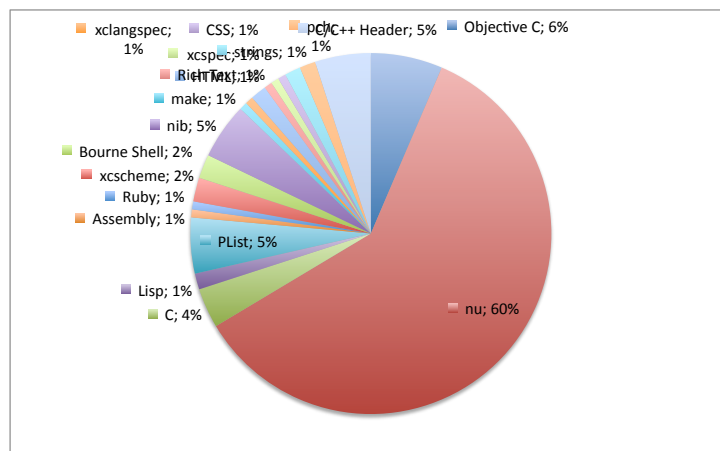
With our exploratory study of five successful ecosystems with closed and open platforms, we took one, but self-contained step towards building a theory that explains variability mechanisms in ecosystems. We contribute a conceptual framework about variability in ecosystems, and phenomena and hypotheses that have practical implications for project management, architecture, and tool support. Among others, we observe that closed platforms allow almost arbitrary changes, but need heavyweight processes to assure quality; that variability models are too fragile for distributed variability management; that open platforms with vibrant free markets imply strictly controlled main platforms and capability-based dependencies. These rely on a centralized and a stable vocabulary. Finally, the mechanisms found in open platforms call for research on *variability encouragement*.

12. REFERENCES

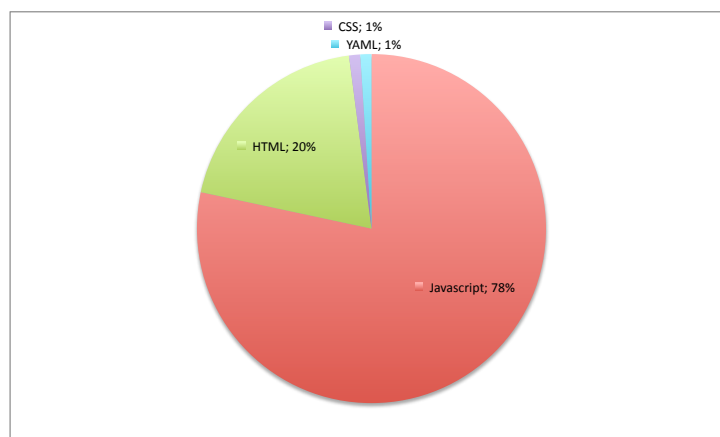
- [1] Android Open Source Project – People and Roles. <http://source.android.com/source/roles.html>.
- [2] Debian policy. <http://debian.org/doc/debian-policy>.
- [3] Eclipse marketplace. <http://marketplace.eclipse.org>.
- [4] eCos. <http://ecos.sourceforge.org/>.
- [5] eCos and RedBoot based products showcase. <http://ecoscentric.com/ecos/examples.shtml>.
- [6] Number of available Android applications. <http://appbrain.com/stats/number-of-android-apps>.
- [7] OpenIntents. <http://openintents.org>.
- [8] Unofficial Debian Repositories. <http://apt-get.org>.
- [9] Yoxos on Demand. <http://ondemand.yoxos.com>.
- [10] Eclipse Development Process. http://eclipse.org/projects/dev_process/development_process_2010.pdf, 2011.
- [11] Some development model notes. <http://lwn.net/Articles/108484>, May 2011.
- [12] O. Barbosa and C. Alves. A systematic mapping study on software ecosystems. In *IWSECO*, 2011.
- [13] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *VaMoS*, 2013.
- [14] T. Berger, S. She, K. Czarnecki, and A. Wąsowski. Feature-to-Code mapping in two large product lines. In *SPLC*, 2010.
- [15] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *ASE*, 2010.
- [16] J. Bosch. From software product lines to software ecosystems. In *SPLC*, 2009.
- [17] C. Burkard, T. Widjaja, and P. Buxmann. Software ecosystems. *Wirtschaftsinformatik*, 54, 2012.
- [18] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
- [19] J. Corbet, G. Kroah-Hartman, and A. McPherson. Linux kernel development. https://www.linuxfoundation.org/sites/main/files/lf_linux_kernel_development_2010.pdf, 2010.
- [20] R. D. Cosmo and S. Zacchiroli. Feature diagrams as package dependencies. In *SPLC*, 2010.
- [21] K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.
- [22] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In *Guide to Advanced Empirical Software Engineering*. Springer, 2008.
- [23] K. M. Eisenhardt and M. E. Graebner. Theory building from cases: Opportunities and challenges. *Academy of management journal*, 50(1):25–32, 2007.
- [24] J. A. Galindo, D. Benavides, and S. Segura. Debian packages repositories as Software Product Line models. In *ACoTA*, 2010.
- [25] IT Radar. Software ecosystems - interview with slinger jansen. http://www.it-radar.org/serendipity/uploads/transkripte/SECO-Transcript_1.pdf, 2012.
- [26] S. Jansen, A. Finkelstein, and S. Brinkkemper. A sense of community: A research agenda for software ecosystems. 2009.
- [27] M. Jørgensen and D. Sjøberg. Generalization and theory-building in software engineering research. In *EASE'04, at ICSE'04*, 2004.
- [28] J. Kabbedijk and S. Jansen. Steering Insight: An Exploration of the Ruby Software Ecosystem. In *ICSOB, LNBIP*, 2011.
- [29] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU, 1990.
- [30] M. Krafft. *The Debian System*. Open Source Press, 2005.
- [31] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the Linux kernel variability model. In *SPLC*, 2010.
- [32] J. D. McGregor. Ecosystems, continued. *Journal of Object Technology*, 8(7), 2009.
- [33] M. Mendonca, A. Wąsowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *SPLC*, 2009.
- [34] D. G. Messerschmitt and C. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, 2003.
- [35] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the apache server. In *ICSE*, 2000.
- [36] K. Schmid. Variability modeling for distributed development - a comparison with established practice. In *SPLC*. 2010.
- [37] K. Schmid, R. Rabiser, and P. Grünbacher. A comparison of decision modeling approaches in product lines. In *VaMoS*, 2011.
- [38] C. Seidl and U. Assmann. Towards modeling and analyzing variability in evolving software ecosystems. In *VaMoS*, 2013.
- [39] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the Linux Kernel a Software Product Line? In *SPLC-OSSPL*, 2007.
- [40] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.
- [41] J. Van Gurp and C. Prehofer. From SPLs to Open, Compositional Platforms. In *Dagstuhl Seminar 08142*, 2008.
- [42] J. van Gurp, C. Prehofer, and J. Bosch. Comparing practices for reuse in integration-oriented software product lines and large open source software projects. *SPE*, 2010.
- [43] E. Zini. A cute introduction to debtags. In *5th annual Debian Conference*, volume 10, page 17, 2005.

B

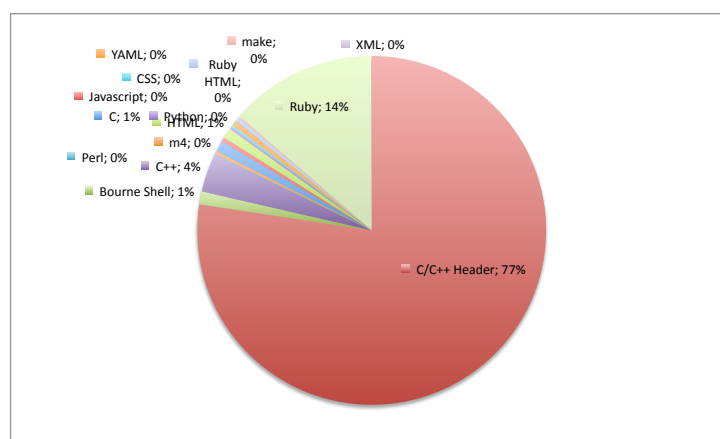
Multi-language Software Systems on GitHub



(a) Nu <https://github.com/timburks/nu>

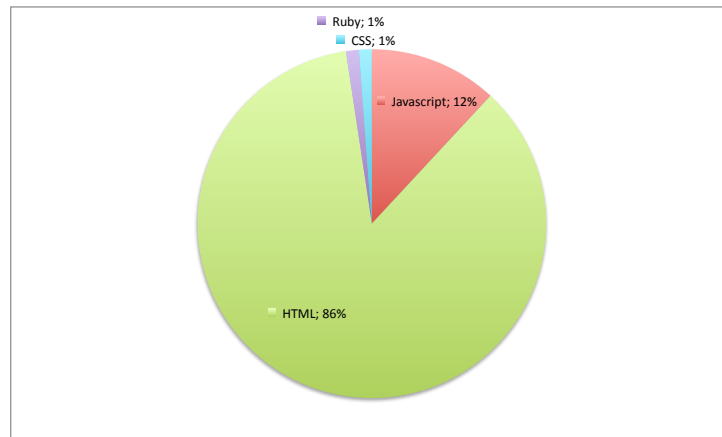


(b) Prototype <https://github.com/sstephenson/prototype>

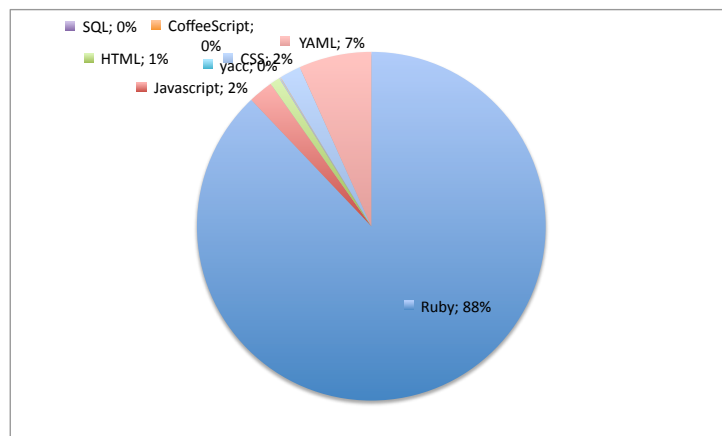


(c) Passanger <https://github.com/FooBarWidget/passenger>

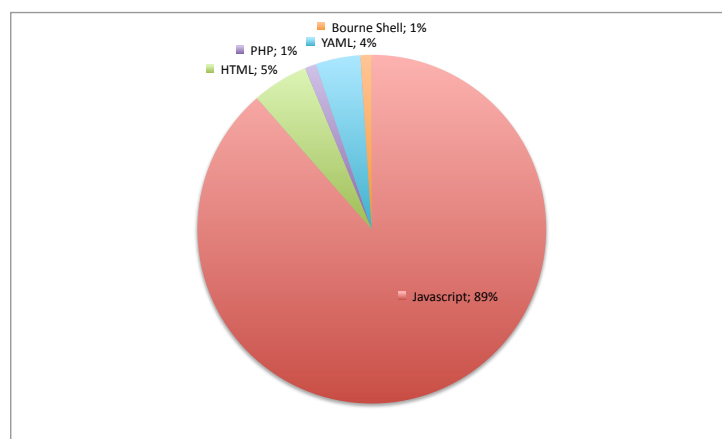
Figure B.1: The twelve most interesting projects on GitHub and the constituting programming languages



(a) *Scriptaculous* <https://github.com/madrobby/scriptaculous>

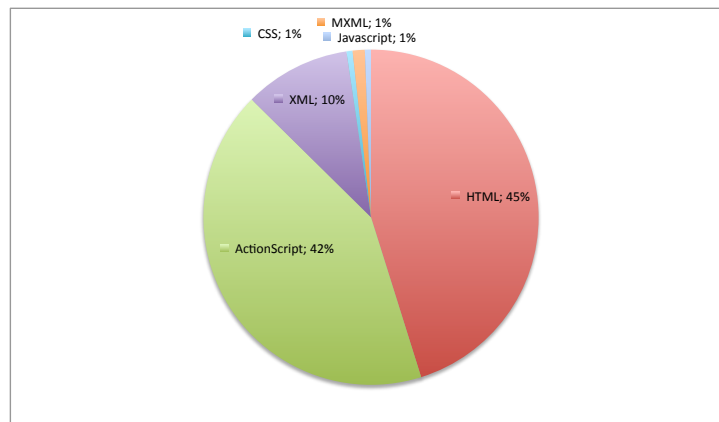


(b) *Rails* <https://github.com/rails/rails>

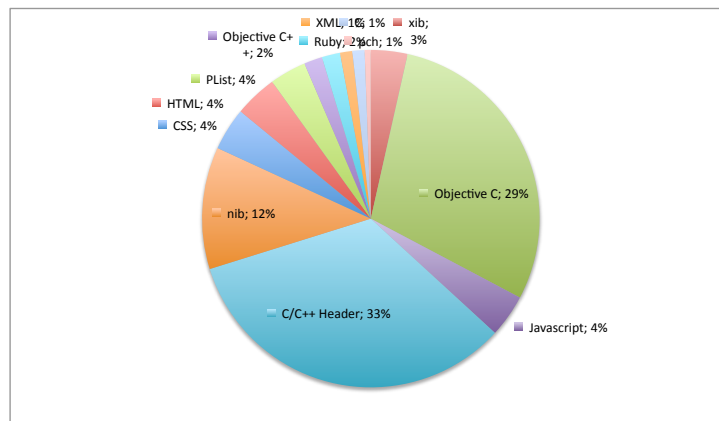


(c) *mootools-core* <https://github.com/mootools/mootools-core>

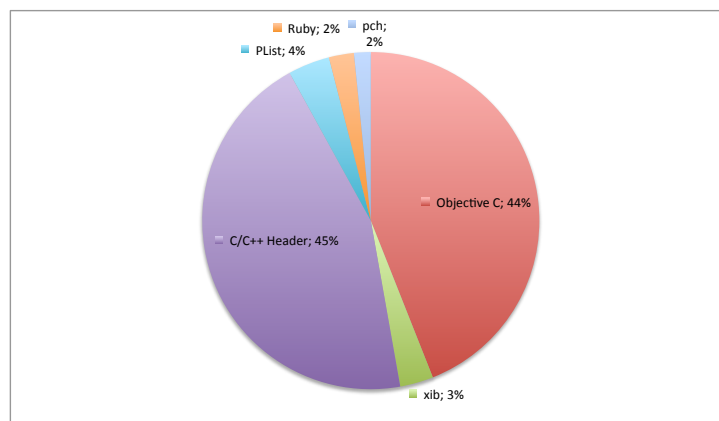
Figure B.2: The twelve most interesting projects on GitHub and the constituting programming languages (continued from Figure B.1)



(a) Restfulx <https://github.com/dima/restfulx>

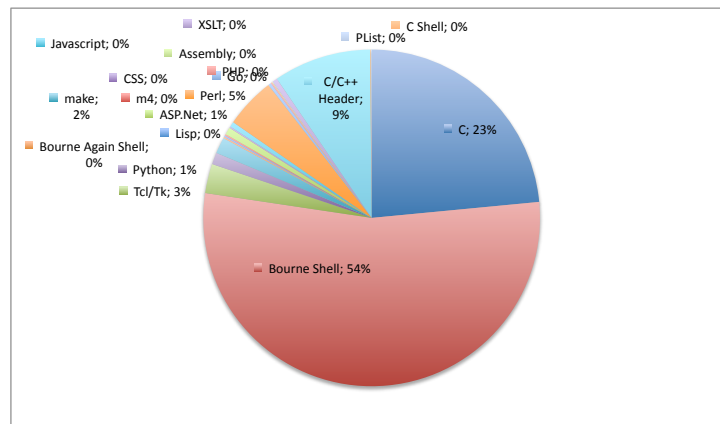


(b) GitX <https://github.com/pieter/gitx>

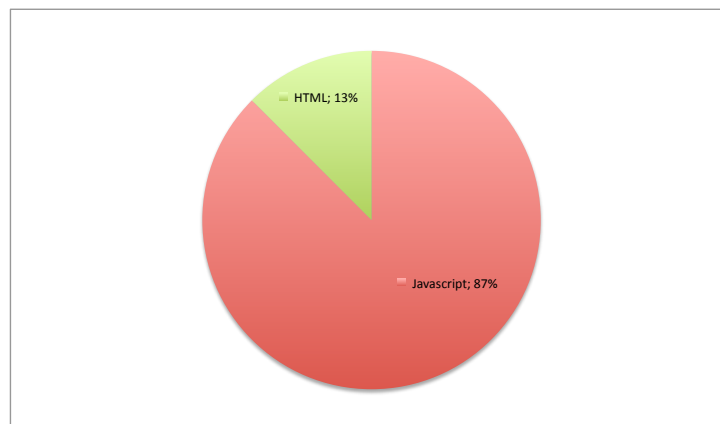


(c) asi-http-request <https://github.com/pokeb/asi-http-request>

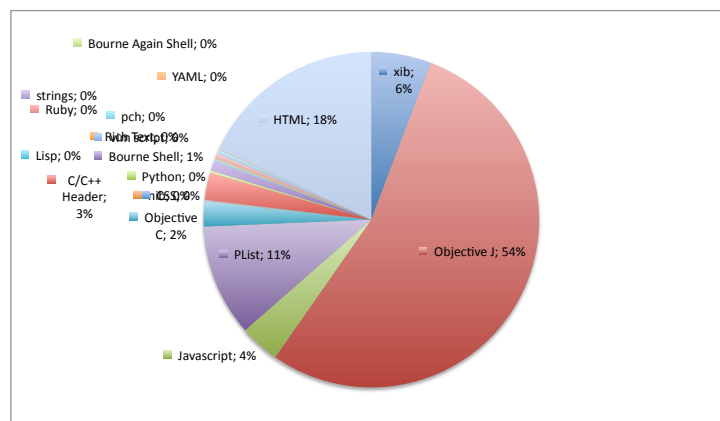
Figure B.3: The twelve most interesting projects on GitHub and the constituting programming languages (continued from Figure B.2)



(a) *Git* <https://github.com/git/git>



(b) *Raphael* <https://github.com/DmitryBaranovskiy/raphael>



(c) *Cappuccino* <https://github.com/cappuccino/cappuccino>

Figure B.4: The twelve most interesting projects on GitHub and the constituting programming languages (continued from Figure B.3)