Softw Syst Model (2015) 14:383–411
DOI 10.1007/s10270-013-0376-y

SPECIAL SECTION PAPER

The design space of multi-language development environments

Rolf-Helge Pfeiffer · Andrzej Wąsowski

Received: 8 November 2012 / Revised: 12 August 2013 / Accepted: 15 August 2013 / Published online: 20 September 2013

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.

Communicated by Dr. Antonio Vallecillo and Dr. Juha-Pekka Tolvanen.

R.-H. Pfeiffer (✉) · A. Wąsowski
Process and System Models Group, IT University of Copenhagen, Copenhagen, Denmark
e-mail: ropf@itu.dk

A. Wąsowski
e-mail: wasowski@itu.dk

Keywords Multi-language development environment · Multi-modeling · Cross-language relations

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 [93], 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 [83]. 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, ADempiere,[2] uses 19 languages. The eCommerce systems Magento[3] and X-Cart[4] 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, KM3,[5] etc.), modeling (DSLs, UML, CVL[6]), validation (OCL, EVL,[7] etc.), model-

[1] http://ofbiz.org, see also [44] on use of DSLs in OFBiz.
[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.variabilitymodeling.org.
[7] http://www.eclipse.org/epsilon/doc/evl/.

to-model transformation (QVT, ATL,[8] etc.), code generation (Acceleo,[9] XPand,[10] etc.), and scripting (MWE2,[11] 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 [25]. Usually, more than one language is needed, since non-trivial systems span multiple problem domains and multiple technical spaces [43]. Existing domain-specific and general-purpose languages are brought into the development in order to reuse existing frameworks, tools, and technology stacks [16]. Moreover, modern systems are rarely stand-alone and increasingly integrate with other systems, which require use of interface mechanisms and integration of their languages [62].

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, and properties files. To simplify presentation, we refer to all these as *mograms* [53] in this paper.

Mograms are often heavily interrelated. For example, OFBiz contains hundreds of relations across mograms in different languages [46,69]. 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 [70].

Example JTrac[12] is an open-source multi-language Web-based bug tracking system. JTrac's log-in page is implemented using mograms in three different languages. The log-in 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 log-in and password and their order. Since JTrac is built using the Apache Wicket[13] Web-development framework, the HTML code contains some 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 log-in page. The Java code provides logic for evaluating a log-in (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 log-in 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 (IDEs) 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)*, aims 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

[8] http://eclipse.org/atl.

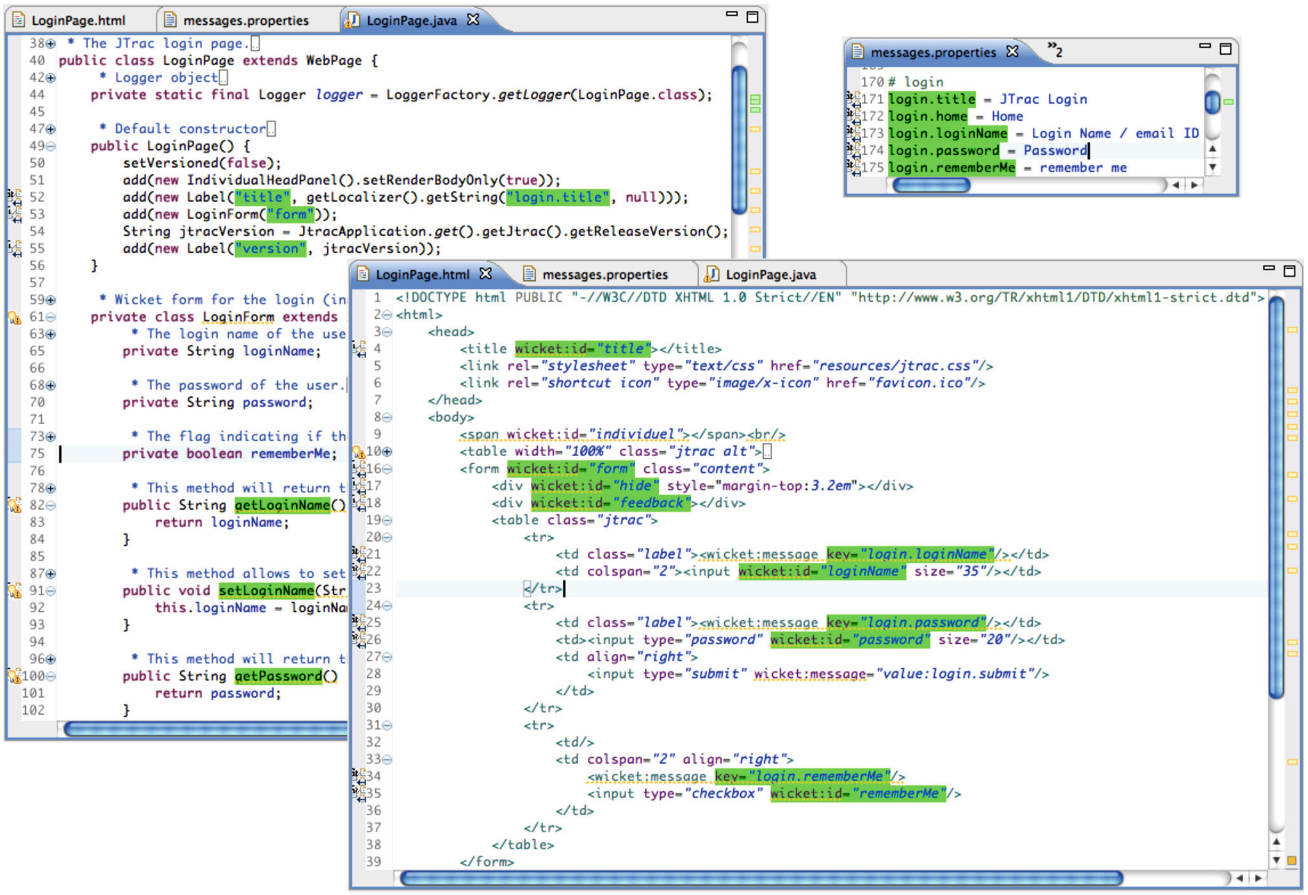
[9] http://eclipse.org/acceleo.

[10] http://wiki.eclipse.org/Xpand.

[11] http://help.eclipse.org/helios/topic/org.eclipse.xtext.doc/help/MWE2.html.

[12] http://www.jtrac.info/.

[13] http://wicket.apache.org/.



```java
 * The JTrac login page.
public class LoginPage extends WebPage {
     * Logger object
    private static final Logger logger = LoggerFactory.getLogger(LoginPage.class);

     * Default constructor
    public LoginPage() {
        setVersioned(false);
        add(new IndividualHeadPanel().setRenderBodyOnly(true));
        add(new Label("title", getLocalizer().getString("login.title", null)));
        add(new LoginForm("form"));
        String jtracVersion = JtracApplication.get().getJtrac().getReleaseVersion();
        add(new Label("version", jtracVersion));
    }

     * Wicket form for the login (in
    private class LoginForm extends
         * The login name of the use
        private String loginName;

         * The password of the user.
        private String password;

         * The flag indicating if th
        private boolean rememberMe;

         * This method will return t
        public String getLoginName()
            return loginName;
        }

         * This method allows to set
        public void setLoginName(Str
            this.loginName = loginNa
        }

         * This method will return t
        public String getPassword()
            return password;
        }
```

```properties
# login
login.title = JTrac Login
login.home = Home
login.loginName = Login Name / email ID
login.password = Password
login.rememberMe = remember me
```

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<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="individuel"></span><br/>
        <table width="100%" class="jtrac alt">
        <form wicket:id="form" class="content">
            <div wicket:id="hide" style="margin-top:3.2em"></div>
            <div wicket:id="feedback"></div>
            <table class="jtrac">
                <tr>
                    <td class="label"><wicket:message key="login.loginName"/></td>
                    <td colspan="2"><input wicket:id="loginName" size="35"/></td>
                </tr>
                <tr>
                    <td class="label"><wicket:message key="login.password"/></td>
                    <td><input type="password" wicket:id="password" size="20"/></td>
                    <td align="right">
                        <input type="submit" wicket:message="value:login.submit"/>
                    </td>
                </tr>
                <tr>
                    <td/>
                    <td colspan="2" align="right">
                        <wicket:message key="login.rememberMe"/>
                        <input type="checkbox" wicket:id="rememberMe"/>
                    </td>
                </tr>
            </table>
        </form>
```

Fig. 1 Mograms in three languages describing JTrac's log-in page shown in Coral user interface

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 the 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 such as TexMo and Coral can clearly advance the state of the art in tooling for language integration. An important aspect of both TexMo and Coral is 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 [71]. We also adapt some elements from [70]. 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 (Sects. 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 Java, HTML, and property files. The semantic rules underlying the Web-application framework Wicket establish the cross-language relations between the files.

The popular IDEs, such as 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 [35]. Mostly, 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 the development of multi-language systems, obstructing their modification and evolution [42,46,69]. Unlike IDEs, which just integrate development tools, a MLDE integrates different languages by relating mograms across language boundaries. This way, MLDEs are able to address the challenge of modification and evolution of multi-language systems.

We surveyed IDEs, programming editors,[14] 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) [70]:

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 [19].
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 navigation of 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 [61,91].

To address the same requirements in an MLDE, in a cross-language fashion, one needs to make three fundamental design decisions:

(a) *How to represent different programming languages?*
(b) *How to relate them?*
(c) *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 [18,51]. 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, and properties files written in various diverse languages.

Definition 1 *Mograms* are all files that are created, edited, or modified by humans or machines with the purpose of developing, customizing, or modifying a software system. Such files may contain source code, models, plain text, etc.

[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/.

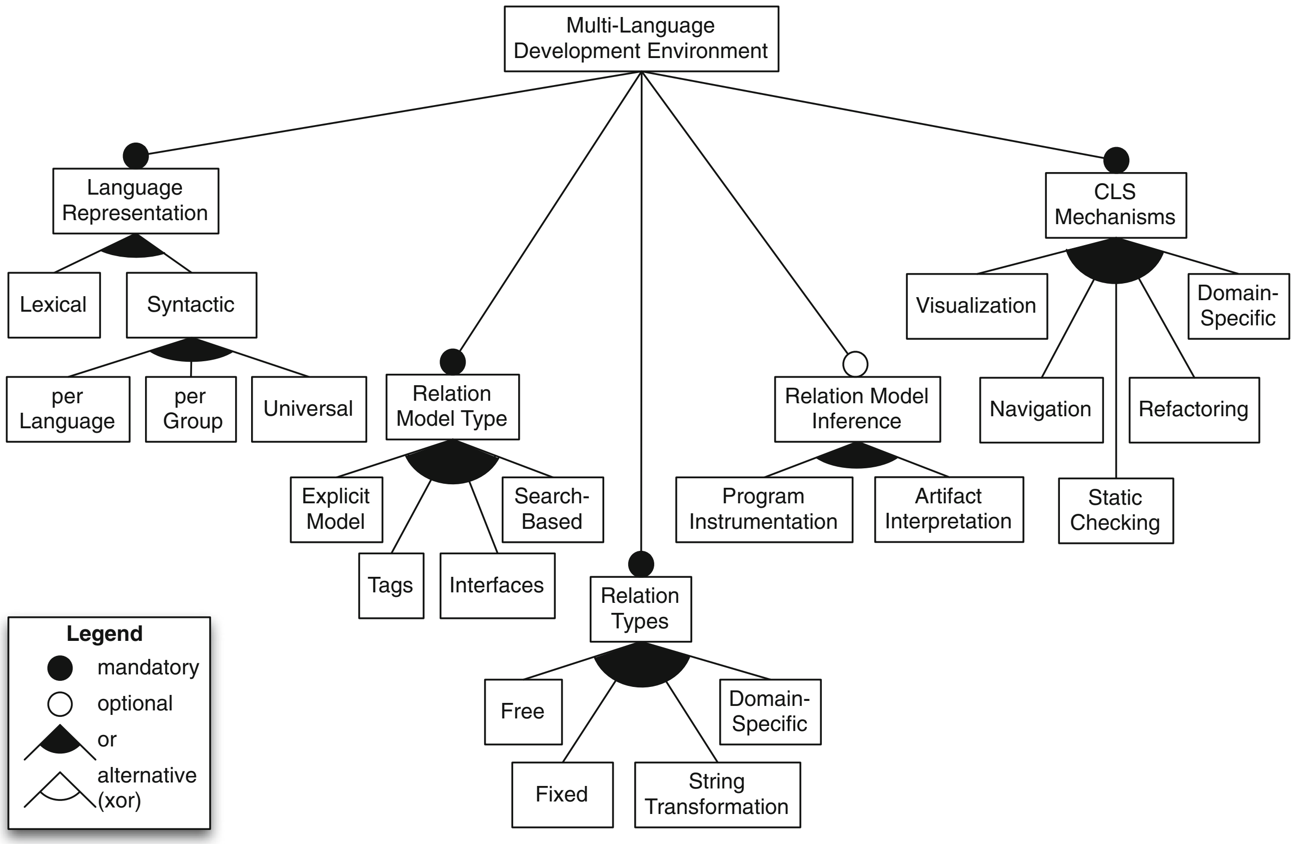


Fig. 2 Taxonomy for multi-language development environments

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 such as trees and graphs to describe concepts and their relations. This work is strongly influenced by the credo *"Everything is a model"* [13]. Often, metamodels are used for specification of syntactic language representations. ASTs 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

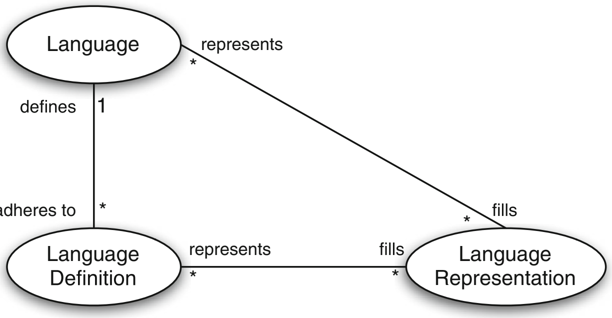


Fig. 3 The concepts of language, language definition, language representation, and their relations

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 [78] (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 [82] formally defines 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 the 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, and a metamodel using data structures such as trees and graphs.

Typical modern IDEs, such as Eclipse or NetBeans, represent mograms in any given language using a separate AST, 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 [38], Xtext [26], and Spoofax [52][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 [15] 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 [12] 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 [39]. Also, JavaML [11] uses XML for a structural representation of Java source code. On the other hand, SmartEMF [42] 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 [69].

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 such as trees and 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. 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.

[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.

X-Develop [80,81] implements an extensible model for language group representation to provide refactoring across object-oriented and markup 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 syntax tree [32]. *Meta*, a language family definition language, allows the grouping of languages by characteristics, e.g., object-oriented languages in *Meta(Oopl)* [47]. The Prolog knowledge base in [42] can be considered as a language group representation for OFBiz' DSLs, used to check for cross-language constraints. The Generic Intermediate Metamodel in [29] 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 such as trees and 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 most language group representations are suggestive of being universal representations. However, when discussing schemes of tool integration, Meyers [59] 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 [71] 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, which distinguishes them from plain IDEs. We have identified four different techniques to express cross-language relations in MLDEs:

2.2.1 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 [90] 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* [16,50], *trace models* [21,33,49,56,65,67], or *macromodels* [74]. 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 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 are identified by uniform resource identifiers (URIs) (lines 11–13 and 18–20, respectively).

Listing 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−5
          d54−4183−9dcf−ecd4f378f28d[FIXED]
25  ...
26 }
```

2.2.2 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 2 and 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 [34]. Hypertext systems interpret tags within mograms as anchors, and links. After interpretation, a relation is established. HyperPro [63,66] is a programming environment that treats mograms in a software system as hypertext. That is, mograms can be enriched with tags linking fragments across language boundaries.

Listing 2 An excerpt of a Java class with link tags

```
public class LoginPage {
 private static final Logger logger =...

 public LoginPage() {
   setVersioned(false);
   add(new IndividualHeadPanel().setRenderBodyOnly(true));
   add(new Label(@link(in(../LoginPage.html), target(wicket:title)),
     getLocalizer().getString("login.title", null)));
   add(new LoginForm(@link(in(../LoginPage.html),target(wicket:form)
)));
   String jtracVersion = JtracApplication.get().getJtrac().
           getReleaseVersion();
   add(new Label("version", jtracVersion));
 }
 ...
}
```

Listing 3 An excerpt of HTML code with relation anchor tags

```
<html>
 <head>
  <title @anchor(wicket: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>
 ...
   <form @anchor(wicket:form) class="content">
       ...
   </form>
   ...
 </body>
</html>
```

DEFT [89], 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 [40,41] is a composition framework for invasive composition. Components encoding various concerns are defined separately and composed when a system is specified. Both works [40,41] 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. [56] 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.

2.2.3 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.

Definition 11 *Interface-based relation models* explicitly define fragments and their relations in interfaces. Interfaces are separate artifacts accompanying interrelated mograms.

Listings 4 and 5 illustrate two interfaces for the interrelated Java and HTML mograms of Fig. 1. The interfaces are expressed in the Tengi interface DSL [68]. Tengi interfaces define relation ends in corresponding mograms (ENTITY) as ports (LOCATOR). Outports (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.

Alfaro and Henzinger [3] 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. Based on this work, Hessellund and Wąsowski [46] define interfaces for interrelated models and metamodels to explicitly describe relations between models crossing language boundaries. Compared to the interfaces in [46], OSGi interfaces [58] are more coarse

Listing 4 A Tengi interface corresponding to LoginPage.java

```
TENGI LoginLogic ENTITY "LoginPage.java" [
 IN: { loginTitleHTML, loginFormHTML }; CONSTRAINT: loginTitleHTML &
        loginFormHTML;
 OUT: { loginTitleJava, loginFormJava}; CONSTRAINT: loginTitleJava &
        loginFormJava;
](
 LOCATOR loginTitleJava IN "LoginPage.java" OFFSET 198 LENGTH 5;
 LOCATOR loginFormJava design IN "LoginPage.html" OFFSET 278
        LENGTH 4;
}
```

Listing 5 A Tengi interface corresponding to LoginPage.html

```
TENGI LoginView ENTITY "LoginPage.html" [
 IN: { loginTitleJava, loginFormJava}; CONSTRAINT: loginTitleJava &
        loginFormJava;
 OUT: { loginTitleHTML, loginFormHTML}; CONSTRAINT: loginTitleHTML
        & loginFormHTML;
](
 LOCATOR loginTitleHTML IN "LoginPage.html" OFFSET 27 LENGTH 17;
 LOCATOR loginFormHTML design IN "LoginPage.html" OFFSET 244
        LENGTH 16;
}
```

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' [78] tag 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.

2.2.4 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 6 illustrates a search-based relation model. It is expressed in 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 unknown data in open systems. For example, in [88] relations across documents in different applications are visualized on user request by searching the contents of all displayed documents. In [22] consistency rules for models in different UML languages are evaluated to find inconsistencies in interrelated models. Hessellund and Sestoft [45] apply code flow analysis to statically check interrelated XML and Java source code. Cross-language relations are formalized into consistency constraints checking properties of ASTs of parsed XML files and Java source code. PAMOMO [33] utilizes triple graph patterns to define constraints, i.e., relations between models. The tool allows the specification of 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 constitute a search-based relation model.

Also, GPLs are used to express search-based relation models. For example, in SmartEMF [42] 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 constitutes a search-based relation model.

Listing 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.commons.
              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 wickedIDsInHTML
7    is info display "A wicketID to property key relation.";
8
9  string transformation: Key in properties <––> StringReference in
          java with wickedIDsInJava
10   is info display "A wicketID to property key relation.";
11
12 fixed: StringReference::value in java <––> StringValParameter::
          value in html with wickedIDsInJavaConstructors
13   is info display "Wicket IDs in Java constructor call.";
14
15 string transformation: NamedElementName in java <––>
          StringValParameter in html with getterMethods
16   is info display "Wicket IDs require a getter method in Java";
17
18 string transformation: NamedElementName in java <––>
          StringValParameter in html with setterMethods
19   is info display "Wicket IDs require a setter method in Java";
```

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 Listings 4 and 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 us to identify fragments by path expressions navigating the data structure of the language representation. For example, the explicit relation model in Listing 1 utilizes URIs to specify relation ends in mograms.
Query Evaluation Alternatively, mograms with syntactic language representations allow us to identify fragments via queries. For example, the search-based relation model in Listing 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, and a code generator, 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. [79] describe a visualization tool allowing interrelation of information across domains, even across concrete syntaxes. 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. [88] discuss visualization across applications and documents. 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 [67], 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 the implementation of 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 semiautomatically. The inference may exploit either static properties of a system, i.e., its mograms, or

[17] www-01.ibm.com/software/awdtools/doors.

its dynamic behavior [31]. 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* [14,31,85,86] in the model-driven development community and *schema matching* [72,77] in the database community aim 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 subgraphs 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 links between model elements or objects which are in relation because of a transformation directive. For example, Epsilon Transformation Language (ETL) [55] automatically generates a trace model for each model transformation guarded by a post condition. Atlas Transformation Language (ATL) [92] establishes a trace model via a similar mechanism. Also, the QVT [64] transformation language has built-in support for traceability [8]. 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 [65], 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 [49] automatically merges traceability rules into existing ATL transformation rules before their execution populating a trace model. Grammel and Kastenholz [30] 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 of 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 [15], as instance of the JaMoPP Java 5 model [39], or as instance of our Java 5 model [69]. 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 to as *ontological metamodeling* or *logical metamodeling*, and second *word abstraction*, also referred to as *linguistic metamodeling* or *physical metamodeling* [9,10,87]. Type abstraction is a unifying abstraction that 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 [17], the fundamental difference in 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. Considering 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

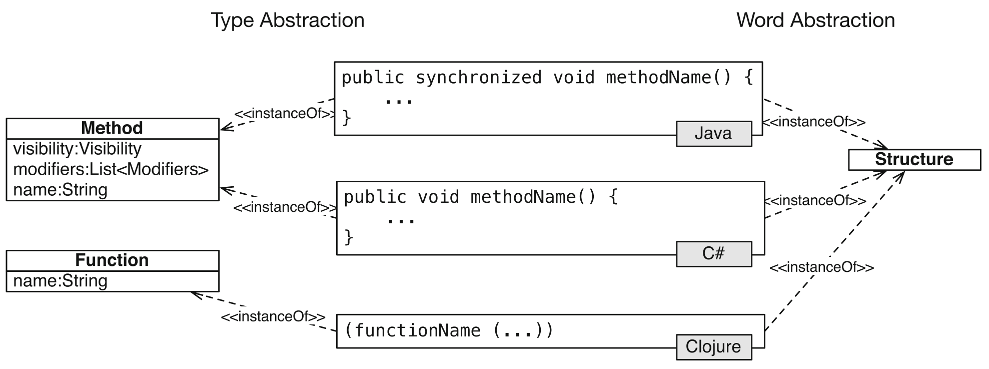


Fig. 4 Type abstraction and word abstraction, two orthogonal abstraction mechanisms

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 us 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 dimensions using the relative measures as low, medium, and high. The purpose of this discussion is to raise awareness toward the impact of design decision using two dimensions as examples.

Adaptability is the ability of an MLDE to be used for the 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 MLDE should be used for the 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 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 hardwired. 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 metalevel (language level). Search-based relation models can be reused for the 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

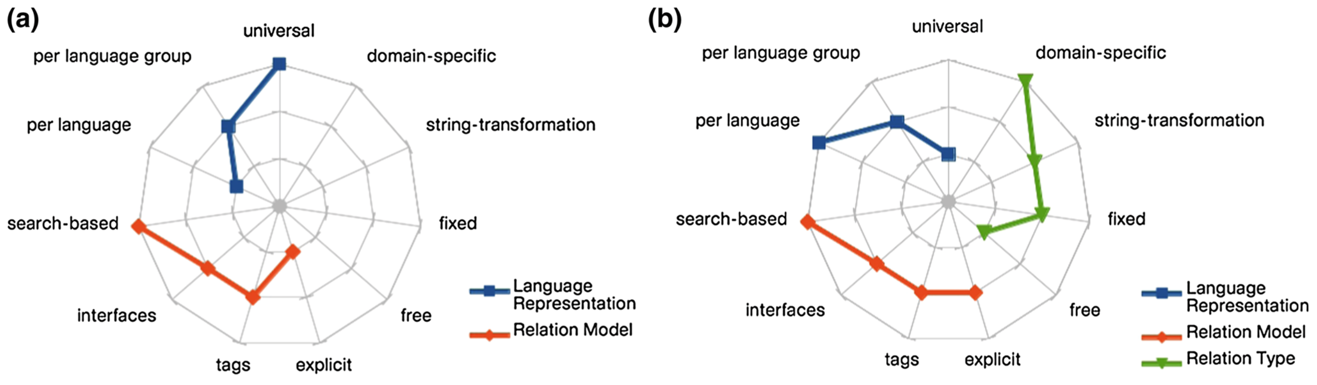


Fig. 5 The impact of design choices of MLDEs along the relative measures low (*inner ring*), medium (*central ring*), or high (*outer ring*) with respect to adaptability and feature richness. **a** The impact of design choices on adaptability of a MLDE. Relation types are not included, as they have no impact on adaptability. **b** The impact of design decisions on richness of functionality of a MLDEs

keep arbitrary information about the reason for their existence; thus, they allow for MLDEs with rich domain-specific functionality.

3.3 TexMo

TexMo[18] 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 us to easily deploy TexMo for the 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 developer's perspective. Alone the survey by Winkler and Pilgrim [90] 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 unidirectional relations using a *key-reference* metaphor. For example, login.title on line 171 in Fig. 1 is a key 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 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. Contrarily, applying a quick fix to a reference renames this single reference to the content of the corresponding key.

On top of these CLS mechanisms, TexMo provides syntax highlighting for 75 languages. GPLs such as Java, C#, and Ruby, as well as DSLs such as HTML are supported. Standard editor mechanisms such as 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 markup 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.

[18] http://www.itu.dk/~ropf/download/texmo.zip.

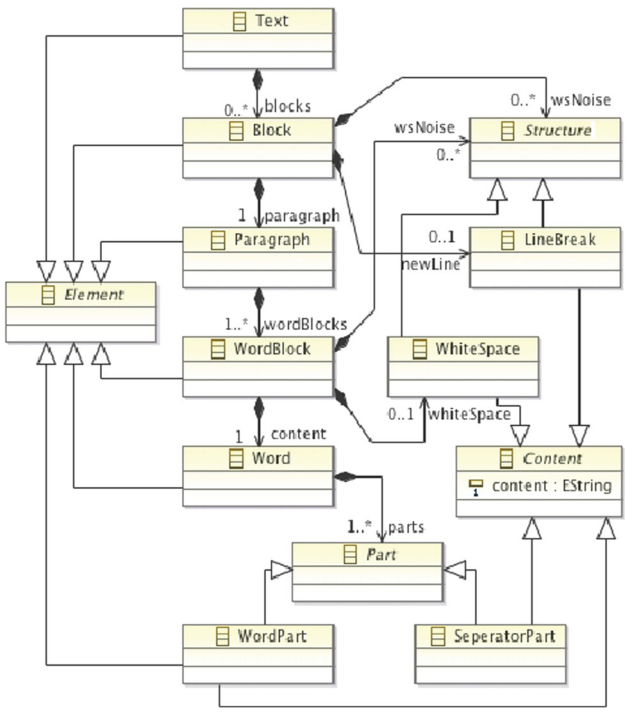


Fig. 6 Text Model—an example of universal language representation as used in TexMo

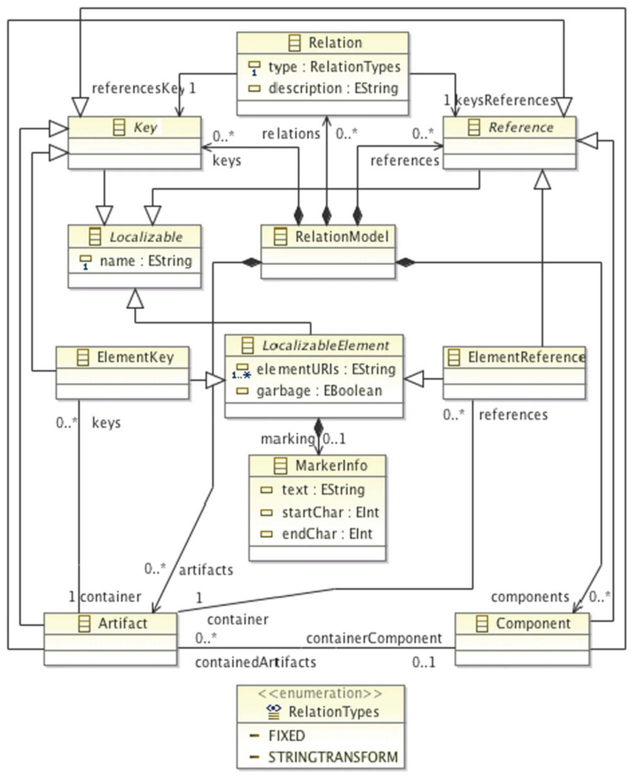


Fig. 7 TexMo's explicit relation model

But, all textual languages share a common coarse-grained structure. The text model (Fig. 6), an AST[19] 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 space. 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., '/' and '.', 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 Spoofax [52] 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 between components (Components).

The relation model instance is kept as a textual artifact storing relations between mogram instances. Listing 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 Listing 1), are identified by URLs on the language representation model (lines 11–13, 18–20 Listing 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 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

[19] The grammar rules for TexMo's universal language representation can be found in the file TexMo.cs in the TexMo sources.

references. The inference mechanism presented in Sect. 3.4 could be adapted to semiautomatic 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 us to connect arbitrary text blocks with each other, for example, documentation information with implementation code.

3.4 Coral

Coral[20] 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] and QWickie, an Eclipse plug-in for Wicket.[24] The main reason for the 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 (Fig. 10).

Coral supports both unidirectional and bidirectional 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 us to reveal detailed information, e.g., the location of the opposite relation end in another file (see Fig. 8 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. 8 top right).
4. *Refactoring* Broken relations can be fixed automatically using refactorings. Currently, Coral provides initial basic rename refactorings that rename all opposite relation ends to the content of the relation end to which the refactoring is applied. Coral uses Refactory [5,73], 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 9 illustrates excerpts of language representation models for Java (Fig. 9a), HTML (Fig. 9b), and properties files (Fig. 9c). All three language representation models rely on type abstraction. They contain abstractions over a mogram's contents. The language representations for parametrizing the Coral framework are generated using EMFText[25] [5], a concrete syntax mapper for EMF models. Technically, the Coral framework can be parametrized with other language representations 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 [39]), XML, Hibernate XML, HTML, properties files,[26] 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 that need to be installed to Eclipse and registered to a Coral plug-in containing the constraint libraries.

[20] http://www.itu.dk/~ropf/coral.html.

[21] http://www.eclipse.org/ajdt.

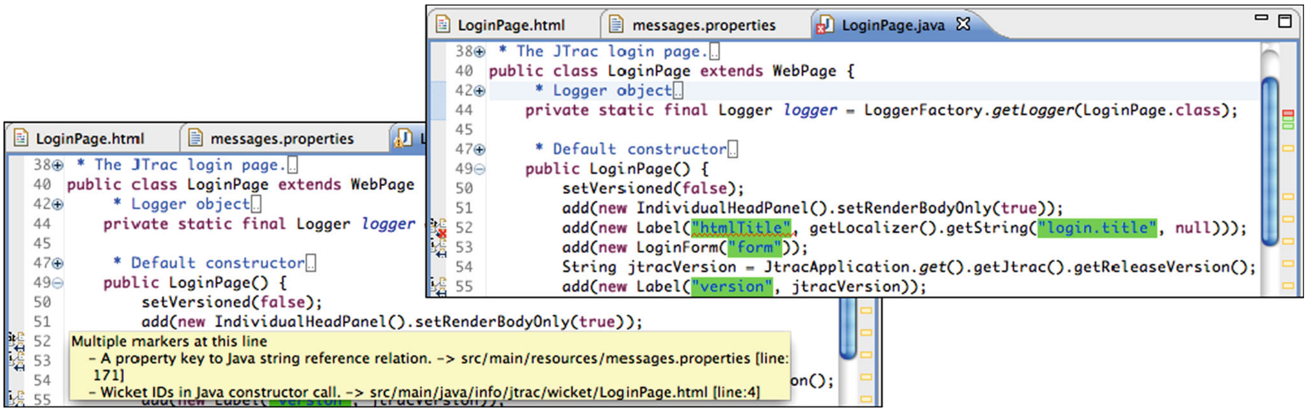
[22] http://www.springsource.org/sts.

[23] http://www.hibernate.org/subprojects/tools.html.

[24] http://code.google.com/p/qwickie.

[25] http://emftext.org.

[26] A modified version of http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Properties_Java.



```
 * The JTrac login page.
public class LoginPage extends WebPage {
    * Logger object
   private static final Logger logger = LoggerFactory.getLogger(LoginPage.class);

    * Default constructor
   public LoginPage() {
       setVersioned(false);
       add(new IndividualHeadPanel().setRenderBodyOnly(true));
       add(new Label("htmlTitle", getLocalizer().getString("login.title", null)));
       add(new LoginForm("form"));
       String jtracVersion = JtracApplication.get().getJtrac().getReleaseVersion();
       add(new Label("version", jtracVersion));
```

```
Multiple markers at this line
  - A property key to Java string reference relation. -> src/main/resources/messages.properties [line:
    171]
  - Wicket IDs in Java constructor call. -> src/main/java/info/jtrac/wicket/LoginPage.html [line:4]
```

Fig. 8 The CLS mechanisms' visualization and static checking in Coral

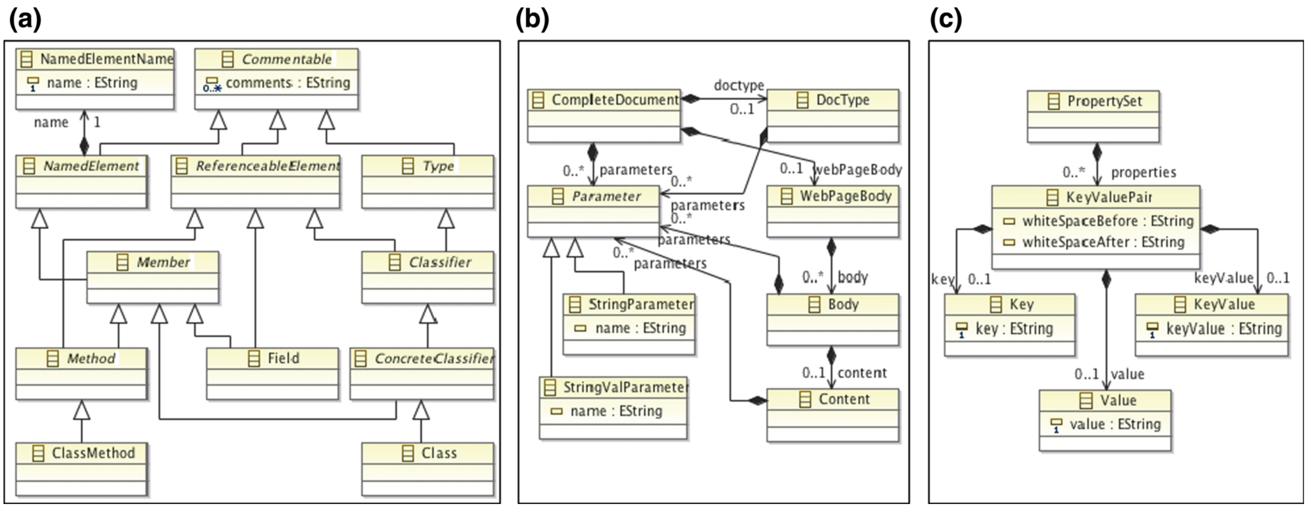


Fig. 9 Per language representation models for three languages. **a** An excerpt of the language representation model for Java code. **b** An excerpt of the language representation model for HTML code. **c** The language representation model for properties code

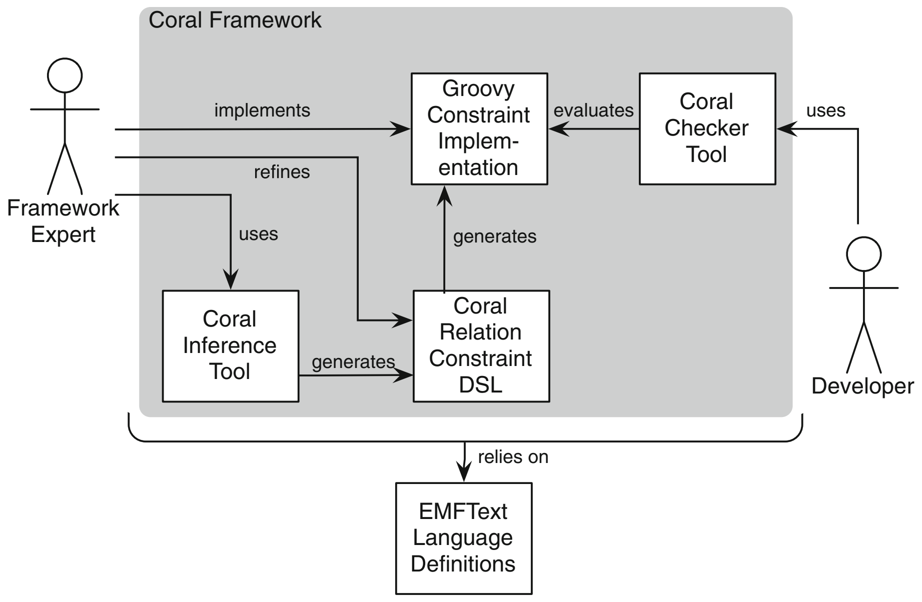
Note that Coral employs a lazy approach when representing mograms with models. That is, only when static checking and refactoring are invoked, the model representations for the corresponding mograms are present in memory.

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 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 contribute in a relation (lines 1–4). Imported language elements can be found in Fig. 9. For Java, these are, for example, element names (NamedElementName), and for HTML, these are string-value parameters (StringValParameter). The Coral DSL allows naming 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 and fixed (lines 6, 9, 12, 15, 18). A constraint connects two language elements of two distinct languages in a unidirectional (←, not shown in the example) or bidirectional (↔) 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 implemented either in an implementation block (not shown in the example) or by provision of a method name referring to a Groovy [20] method implementing the constraint, such as

Fig. 10 Coral's architecture and its user groups



wickedIDsInHTML (the Groovy code is not illustrated here but on the project page[27]). 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 automatically generating a library with possible constraints.

Relation types Coral implements all four relation 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 in LoginPage.html and a string-transformation relation between line 82 in LoginPage.java and line 22 in LoginPage.html. A broken relation is shown in Fig. 8 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. 10). First, the Coral DSL allows for declarative specification of constraint libraries. Second, the Coral checker tool 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 Web site.

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, and 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

[27] http://www.itu.dk/~ropf/coral.html#Constraints.

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 element 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 that are enclosed entirely by a language element's attribute in both languages are potential relation ends. NamedElementName, StringParameter, or Key in Fig. 8 are examples for 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 pairwise inference. When applied to an entire project, the inference

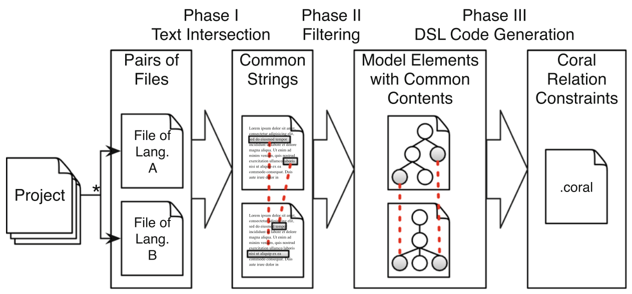


Fig. 11 The four phases to inferring CLRC libraries

tool runs pairwise 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, and 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 hardwired 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 semiautomatic 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; and third, the two dimensions, adaptability and richness of functionality.

Both TexMo and Coral implement the same CLS mechanisms (see Table 1). Visualization, navigation, and static checking have similar look and feel for the users of the tools.

Table 1 Comparison of the two MLDE prototypes with respect to three criteria

	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

Both MLDEs implement rename refactorings that maintain consistency of relation end points. 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 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 is available online.[28] We have run the inference on a 2.9GHz Intel i7 Mac Book with 8GB of RAM, of which 4GB was assigned to the Java 6 virtual machine.

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), Java-Script (8), and 29 other source code files such as shell scripts. Similar to many Web applications, JTrac implements the model-view-controller (MVC) pattern. This is achieved using popu-

[28] http://www.itu.dk/~ropf/coral/tech_experiment.zip.

[29] http://sourceforge.net/projects/j-trac/files/jtrac/2.1.0.

Table 2 Cross-language relations established by Coral inference for JTrac

	Hibernate	Wicket	Total
# cross-language relations	165	4,776	4,941
# of checks	700	33,900	34,600
Total time (min)	0.04	1.27	1.31
Average time (ms)	3.79	2.24	2.27
# of relations per file			13.21
False positives	0/165	0/578	

lar frameworks: Hibernate[30] 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 Table 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 ms.

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 subclusters 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 30 ms to check one mogram, open in an editor, for the relations in which it participates. Standard UI research indicates that response time for the visualization of results of computation actions happening without display of any progress indicators should never exceed 2 s [28,60]. So even if the density of relations would be much higher in other projects, it is very likely that they can be checked within acceptable time.

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 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 that, 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 with 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 [70,71], 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

[30] http://hibernate.org.

development supported by the four fundamental CLS mechanisms from a user perspective. Here we focus on reporting qualitative feedback. See [70] for full experiment results and analysis.

Method We run a single factor with two alternative experiments. 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 a maximum of 13 years (average work experience: around 3 years, median 3 years). Two PhD and one MSc students 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 the 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.

Table 3 Language integration survey questions and quantitative results

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%		

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 online.[31] Twelve of the questions and the corresponding results are given in Table 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?

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?

[31] http://www.itu.dk/people/ropf/survey.zip.

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 the 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

[32] www.eclipse.org/forums/index.php?t=thread\&frm_id=27.

[33] emftext-users@mail-st.inf.tu-dresden.de.

[34] antlr-interest@antlr.org.

[35] users@javacc.java.net.

[36] http://users.parboiled.org.

[37] pyparsing-users@lists.sourceforge.net.

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-fifths mention that they construct GPLs (Q IV). All constructed languages are applied in the 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 admit that they also construct languages that 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 toward 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 support for their languages (Q X), and two-fifths 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 provide neither any language integrations nor tools to enhance IDEs with language integration knowledge.

Regarding RQ7 Even though many language developers provide tools that check for correct language interrelation (over two-thirds, 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 that 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), and (**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 that 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 few responses, we decided to let the survey open for responses. We will update the data on the survey's home page to reduce this risk.

5 Related work

5.1 Taxonomy

The IEEE Standard Glossary of Software Engineering Terminology [84] 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 [1] *...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 and Pilgrim [90] 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 [90]. We propose to use several types of language representation models, which allow models to be related in a generic manner.

In [21,24] a traceability metalanguage 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. [1] 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's 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. [1] 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 in 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. [81] realize that IDEs do not allow for analysis and refactoring of multi-language system and thus are not suitable for the 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 the 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 [6] provides hypertext functionality for heterogeneous *Software Development Environments (SDEs)*. Different programs such as 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 re-establish 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. Coral's constraints are just re-evaluated as soon as a mogram is modified. Thereby, relations do not have to be manually re-established.

Jarzabek [48] describes specification of MLDEs 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 re-evaluated. Unfortunately, the described IDE is VAX-based seems to be discontinued.

Meyers [59] 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 [4,27] 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 [75] is an editor for literate programming integrating natural language and Unified Modeling Language (UML) models via Object Constraint Language (OCL) constraints. Similar to [42], 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 [71]. 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) [54], EMF-IncQuery [37], or Prolog [42] 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* [14,31,85,86] 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 subgraphs is fulfilled, relations, mostly trace links, are automatically created. Second, there is *schema matching* [72,77] in the database community, which 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* [2,7,23,36,57,76] 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 that 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: visualization, 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 with the quick checks. Therefore, an effective, incremental loading strategy has to be researched and implemented in an upcoming version of Coral. Along these 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 [68] 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 suggested surveying 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. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Syst. J. **45**(3), 515–526 (2006)
2. Alexander, I.: Towards automatic traceability in industrial practice. In: Proceedings of the 1st International Workshop on Traceability, pp. 26–31 (2002)
3. Alfaro, L.D., Henzinger, T.A.: Interface theories for component-based design. In: EMSOFT (2001)
4. Altheide, F., Dörr, H., Schürr, A.: Requirements to a framework for sustainable integration of system development tools. In: Proceedings of the 3rd European Systems Engineering Conference (EuSEC), pp. 53–57 (2002)
5. 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. Softw. Syst. Model., 1–37 (2012). doi:10.1007/s10270-012-0284-6
6. Anderson, K.M., Taylor, R.N., Whitehead, Jr., E.J.: Chimera: hypermedia for heterogeneous software development environments. ACM Trans. Inf. Syst. **18**(3), 211–245 (2000)
7. 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 (2002). doi:10.1109/TSE.2002.1041053
8. Aranega, V., Etien, A., Dekeyser, J.L.: Using an alternative trace for QVT. Electron. Commun. EASST **42** (2011)
9. 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, London, UK (2001). http://dl.acm.org/citation.cfm?id=647245.719475
10. Atkinson, C., Kühne, T.: Model-driven development: a metamodeling foundation. IEEE Softw. **20**(5), 36–41 (2003). doi:10.1109/MS.2003.1231149
11. Badros, G.J.: JavaML: a markup language for Java source code. Comput. Netw. **33**(1-6), 159–177 (2000)
12. 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
13. Bézivin, J.: On the unification power of models. Softw. Syst. Model. **4**(2), 171–188 (2005)
14. 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
15. 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 (2010)
16. 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)
17. Colburn, T.: Philosophy and Computer Science. Explorations in Philosophy. M.E. Sharpe (2000). http://books.google.dk/books?id=luF4ElMxqg4C
18. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications (2000)
19. de Figueiredo Carneiro, G., Mendonça, M.G., Magnavita, R.C.: An experimental platform to characterize software comprehension activities supported by visualization. In: ICSE Companion, IEEE (2009)
20. Dearle, F.: Groovy for Domain-Specific Languages, 1st edn. Packt Publishing (2010)
21. Drivalos, N., Kolovos, D.S., Paige, R.F., Fernandes, K.J.: Software language engineering. chap. In: Engineering a DSL for Software Traceability, pp. 151–167. Springer, Berlin, Heidelberg (2009). doi:10.1007/978-3-642-00434-6_10

22. Egyed, A.: Automatically detecting and tracking inconsistencies in software design models. IEEE Trans. Softw. Eng. **37**(2), 188–204 (2011)
23. 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. 163–171. ASE '02, IEEE Computer Society, Washington, DC, USA (2002). http://dl.acm.org/citation.cfm?id=786769.787006
24. Espinoza, A., Garbajosa, J.: The need for a unifying traceability scheme. In: ECMDA-TW 2005. SINTEF ICT Norway, Nuremberg, Germany (November, 2005). http://www.modelbased.net/drupal/node/19
25. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, Reading, MA (2004)
26. 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). doi:10.1145/1869542.1869625
27. Freude, R., Königs, A.: Tool integration with consistency relations and their visualisation. In: ESEC/ FSE Workshop on Tool Integration in System Development (2003)
28. Galletta, D.F., Henry, R.M., McCoy, S., Polak, P.: Web site delays: how tolerant are users? J. AIS **5**(1), 1–28 (2004)
29. Gómez, P.M., Sánchez, M., Flórez, H., Villalobos, J.: Co-creation of models and metamodels for enterprise architecture projects. In: XM 2012—Extreme Modeling, Workshop (2012)
30. 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)
31. 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). doi:10.1007/978-3-642-33666-9_39
32. Groenewegen, D.M., Hemel, Z., Visser, E.: Separation of concerns and linguistic integration in WebDSL. IEEE Softw. **27**(5), 31–37 (2010)
33. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-modelling: from theory to practice. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I, pp. 376–391 (2010)
34. Halasz, F.G., Schwartz, M.D.: The Dexter hypertext reference model. Commun. ACM **37**(2) (1994)
35. Hammond, J.S., Schwaber, C., D'Silva, D.: IDE Usage Trends (2008). http://www.forrester.com/Research/Document/Excerpt/0,7211,43181,00.html
36. 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
37. Hegedüs, Á., Horváth, Á., Ráth, I., Varró, D.: Query-driven soft interconnection of emf models. In: France et al. [29], pp. 134–150
38. 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, Berlin, Heidelberg (2009). doi:10.1007/978-3-642-02674-4_9
39. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the gap between modelling and java. In: Proceedings of the 2nd International Conference on Software Language Engineering (SLE 2009), Revised Selected Papers (2010)
40. Heidenreich, F., Johannes, J., Zschaler, S.: Aspect orientation for your language of choice. In: Workshop on Aspect-Oriented Modeling (AOM at MoDELS) (2007)
41. Henriksson, J., Johannes, J., Zschaler, S., Aßmann, U.: Reuseware—adding modularity to your language of choice. J. Object Technol. **6**(9), 127–146 (2007)
42. Hessellund, A.: SmartEMF: guidance in modeling tools. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, pp. 945–946 (2007)
43. Hessellund, A.: Domain-Specific Multimodeling. Ph.D. thesis, IT University of Copenhagen (2009)
44. Hessellund, 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)
45. Hessellund, 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, Berlin, Heidelberg (2008). doi:10.1007/978-3-540-70592-5_13
46. Hessellund, 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)
47. 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)
48. Jarzabek, S.: Specifying and generating multilanguage software development environments. Softw. Eng. J. **5**(2), 125–137 (1990). doi:10.1049/sej.1990.0015
49. 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)
50. 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)
51. 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)
52. Kats, L.C.L., Visser, E.: The spoofax 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
53. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels (2008)
54. 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)
55. 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, Berlin, Heidelberg (2008)
56. Kolovos, D.S., Paige, R.F., Polack F.A.C.: On-Demand Merging of Traceability Links with Models (2006)

57. 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
58. McAffer, J., VanderLei, P., Archer, S.: OSGi and Equinox: Creating Highly Modular Java Systems, 1st edn. Addison-Wesley Professional, Reading, MA (2010)
59. Meyers, S.: Difficulties in integrating multiview development systems. IEEE Softw. **8**(1), 49–57 (1991)
60. Miller, R.B.: Response time in man-computer conversational transactions. In: Proceedings of the December 9–11, 1968, Fall Joint Computer Conference, Part I, vol. 33, pp. 267–277. AFIPS '68 (Fall, part I), ACM, New York, NY, USA (1968). doi:10.1145/1476589.1476628
61. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: Proceedings of the 31st International Conference on, Software Engineering (2009)
62. 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 (2006). http://www.sei.cmu.edu/uls/downloads.html
63. Nørmark, K., Østerbye, K.: Representing programs as hypertext. In: Lund Institute of Technology, pp. 11–24. Lund University (1994)
64. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, V1.1. http://www.omg.org/spec/QVT/1.1/ (2011)
65. Oldevik, J., Neple, T.: Traceability in model to text transformations. In: Proceedings of ECMDA Traceability Workshop ECMDA Traceability Workshop (ECMDA-TW) (2006)
66. Østerbye, K., Nørmark, K.: An interaction engine for rich hypertexts. In: Ritchie, I., Guimarães, N. (eds.) ECHT, pp. 167–176. ACM (1994)
67. 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**(4), 469–487 (2011)
68. Pfeiffer, R.H., Wąsowski, 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)
69. Pfeiffer, R.H., Wąsowski, A.: Taming the Confusion of Languages. In: Proceedings of the 7th European Conference on Modelling Foundations and Applications, pp. 312–328. ECMFA'11, Springer, Berlin, Heidelberg (2011). http://dl.acm.org/citation.cfm?id=2023522.2023552
70. Pfeiffer, R.H., Wąsowski, A.: Cross-language support mechanisms significantly aid software development. In: France et al. [29], pp. 168–184
71. 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, Berlin, Heidelberg (2012). doi:10.1007/978-3-642-31491-9_15
72. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB J. **10**(4), 334–350 (2001). doi:10.1007/s007780100057
73. 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)
74. 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 (2009)
75. 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
76. 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)
77. Shvaiko, P., Euzenat, J.: A survey of schema-based matching approaches. In: Spaccapietra, S., (ed) Journal on Data Semantics IV, Lecture Notes in Computer Science, Springer, **3730**, 146–171 (2005)
78. 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). doi:10.1145/800209.806466
79. Steinberger, M., Waldner, M., Streit, M., Lex, A., Schmalstieg, D.: Context-preserving visual links. IEEE Trans. Vis. Comput. Graph. (InfoVis '11), **17**(12), 2249–2258 (2011)
80. Strein, D., Kratz, H., Lowe, W.: Cross-language program analysis and refactoring. In: Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation, pp. 207–216 (2006)
81. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An extensible metamodel for program analysis. IEEE Trans. Softw. Eng. **33**(9), 592–607 (2007)
82. Sufrin, B.: Formal specification of a display-oriented text editor. Sci. Comput. Program. **1**(3), 157–202 (1982). http://www.sciencedirect.com/science/article/pii/0167642382900144
83. THE OPEN SOURCE DEVELOPER REPORT—2010 Eclipse Community Survey. http://eclipse.org/org/press-release/20100604_survey2010.php (2011), seen: Mar. 2012
84. The Institute of Electrical and Electronics Engineers: IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard (1990)
85. 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)
86. Voigt, K., Ivanov, P., Rummler, A.: Matchbox: combined metamodel 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). doi:10.1145/1774088.1774563
87. Wagner, S., Deissenboeck, F.: Abstractness, specificity, and complexity in software design. In: Proceedings of the 2nd International Workshop on the Role of Abstraction in Software Engineering (2008)
88. Waldner, M., Puff, W., Lex, A., Streit, M., Schmalstieg, D.: Visual links across applications. In: Proceedings of Graphics Interface (2010)
89. 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)
90. Winkler, S., Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development. Softw. Syst. Model. **9**(4), 529–565 (2010). doi:10.1007/s10270-009-0145-0
91. Xing, Z., Stroulia, E.: Refactoring practice: how it is and how it should be supported—an eclipse case study. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance (2006)

92. Yie, A., Wagelaar, D.: Advanced traceability for ATL. In: 1st International Workshop on Model Transformation with ATL, pp. 78–87 (2009)
93. Zend Technologies Ltd.: Taking the Pulse of the Developer Community. http://static.zend.com/topics/zend-developer-pulse-survey-report-0112-EN.pdf, seen: Feb. 2012

Author Biographies

Rolf-Helge Pfeiffer is currently working as research associate in the Process and System Models Group at IT University of Copenhagen. His research interests are in model-driven development, aspect-oriented programming, and their application in tools. He received a PhD from IT University of Copenhagen for his work on multi-language development environments in 2013. Earlier, he worked as a Junior Research Engineer at Intershop Communications AG Jena, Germany, where he worked on model-driven software product line engineering. He holds a Diplom-Informatiker degree from Friedrich-Schiller University Jena, Germany (2008).

Andrzej Wąsowski is Associate Professor at the IT University of Copenhagen. Earlier, he worked also at Aalborg University in Denmark, and as visiting professor at INRIA Rennes and University of Waterloo, Ontario. His interests are in semantic foundations and tool support for model-driven development, especially for software product lines and component-based systems. Many of these projects involve commercial or open-source partners, primarily in the domain of safety-critical embedded systems. Wąsowski holds a PhD degree from the IT University of Copenhagen, Denmark (2005), and a MSc Engineering degree from the Warsaw University of Technology, Poland (2000). He is a recipient of the *Sapere Aude* Research Leadership Award from The Danish Council for Independent Research (2012).