TexMo: A Multi-language Development Environment

Rolf-Helge Pfeiffer and Andrzej Wąsowski

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

Abstract. Contemporary software systems contain a large number of artifacts expressed in multiple languages, ranging from domain-specific languages to general purpose languages. These artifacts are interrelated to form software systems. Existing development environments insufficiently support handling relations between artifacts in multiple languages.

This paper presents a taxonomy for multi-language development environments, organized according to language representation, representation of relations between languages, and types of these relations. Additionally, we present TexMo, a prototype of a multi-language development environment, which uses an explicit relation model and implements visualization, static checking, navigation, and refactoring of cross-language relations. We evaluate TexMo by applying it to development of a web-application, JTrac, and provide preliminary evidence of its feasibility by running user tests and interviews.

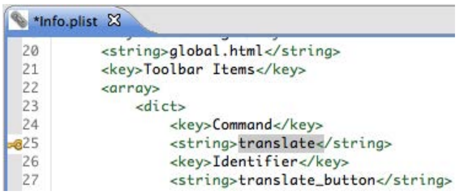
1 Introduction

Maintenance and enhancement of software systems is expensive and time consuming. Between 85% to 90% of project budgets go to legacy system operation and maintenance [6]. Lientz et. al. [19] state that 75% to 80% of system and programming resources are used for enhancement and maintenance, where alone understanding of the system stands for 50% to 90% percent of these costs [25].

Contemporary software systems are implemented using multiple languages. For example, PHP developers regularly use 1 to 2 languages besides PHP [1]. The situation is even more complex in large enterprise systems. The code base of OFBiz, an industrial quality open-source ERP system contains more than 30 languages including General Purpose Languages (GPL), several XML-based Domain-Specific Languages (DSL), config files, property files, and build scripts. ADempiere, another industrial quality ERP system, uses 19 languages. ECommerce systems Magento and X-Cart utilize more than 10 languages each.[1] Systems utilizing the model-driven development paradigm additionally rely on multiple languages for model management, e.g., meta-modelling (UML, Ecore, etc.) model transformation (QVT ATL, etc.), code generation (Acceleo, XPand, etc.), and model validation (OCL, etc.).[2]

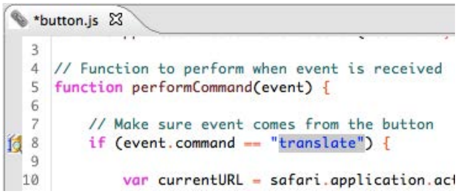
[1] See `ofbiz.apache.org`, `adempiere.com`, `magentocommerce.com`, `x-cart.com`

[2] See `uml.org`, `eclipse.org/modeling/emf`, `omg.org/spec/QVT`, `eclipse.org/atl`, `eclipse.org/acceleo`, `wiki.eclipse.org/Xpand`, `omg.org/spec/OCL` respectively.



```
    <string>global.html</string>
    <key>Toolbar Items</key>
    <array>
        <dict>
            <key>Command</key>
            <string>translate</string>
            <key>Identifier</key>
            <string>translate_button</string>
```

(a) Declaration of the *translate* command attached to a button.



```

// Function to perform when event is received
function performCommand(event) {

    // Make sure event comes from the button
    if (event.command == "translate") {

        var currentURL = safari.application.ac
```

(b) JavaScript code that is executed whenever the button is pressed.

Fig. 1. Declaration of a command and its use

We call software systems using multiple languages, *Multi-Language Software Systems (MLSS)*. Obviously, the majority of modern software systems are MLSSs.

Development artifacts in MLSS can be models, source code, property files, etc. To simplify presentation, we refer to all these as *mograms* [18]. Mograms in MLSS are often heavily interrelated. For example, OFBiz contains many hundreds of relations across its languages [23,13]. Unfortunately, relations across language boundaries are fragile. They are broken easily, as development environments neither visualize nor statically check them.

Consider the following scenario. For simplicity of presentation we use a small example. Our work, though, is not tight to a particular selection of languages, or the particular example system.

Example Scenario. Bob develops a *Safari* web browser *extension*. The extension contributes a button to Safari's menu bar. Pressing the button translates the current web-page to English using *Google translate* and presents it in a new tab. Browser extensions are usually built using HTML, CSS and JavaScript. Bob's extension consists of three source code files: *Info.plist*, *button.js*, and *global.html*.

Plist files serve as an interface for the extension. They tell Safari what the extension contributes to the UI. In Bob's extension, the *Plist* file contains the declaration of a `translate` command attached to a toolbar button (Fig. 1a). *JavaScript* code contains logic attached to buttons, menus, etc. Bob's *button.js* forwards the current URL to Google's translation service whenever the corresponding button is pressed (Fig. 1b). Every extension contains a *global.html* file, which is never displayed. It contains code which is loaded at browser start-up or when the extension is enabled. It is used to provide code for extension buttons, menus, etc. Bob's *global.html* file (not shown here) contains only a single `script` tag pointing to *button.js*.

In Fig. 1a the `translate` command for the button is defined. Fig. 1b shows how the translate command is used in *button.js* in a string literal. This is an example of a *string-based reference* to *Info.plist*. Such string-based references are common in development of MLSSs.

Now, imagine Bob renaming the command in *Info.plist* from `translate` to its Danish equivalent `oversæt`. Obviously, the browser plugin will not work anymore since the JavaScript code in *button.js* is referring to a non-existing command. Symmetrically, the reference is broken whenever the "`translate`" string literal is modified in the button.js file, without the corresponding update to *Info.plist*. □

Existing Integrated Development Environments (IDE) do not directly support development of MLSSs. IDEs do not visualize cross-language relations (markers left to line numbers and gray highlighting in Fig. 1). Neither do they check statically for consistency of cross-language relations, or provide refactorings across mograms in multiple languages. We are out to change this and enhance IDEs into *Multi-Language Development Environments (MLDE)*.

This paper introduces a taxonomy of design choices for MLDEs (Sec. 2). The purpose of this taxonomy is twofold. First, it serves as requirements list for implementing MLDEs, and second it allows for classification of such. We argue for the validity of our taxonomy by a survey of related literature and tools.

As the second main contribution, the paper presents TexMo (Sec. 3), an MLDE prototype supporting textual GPLs and DSLs. It implements actions for *visualization* of, *static checking* of, *navigation* along, and *refactoring* of inter-language relations, and facilities to declare inter-language relations. Additionally, TexMo provides standard editor mechanisms such as syntax highlighting. We position TexMo in our taxonomy and evaluate it by applying it to development of an MLSS and user tests followed by interviews.

2 Taxonomy of Multi-language Development Environments

Popular IDEs like Eclipse or NetBeans implement separate editors for every language they support. A typical IDE provides separate Java, HTML, and XML editors, even though these editors are used to build systems mixing all these languages. Representing languages separately allows for an easy and modular extension of IDEs to support new programming languages. Usually, IDEs keep an *Abstract Syntax Tree (AST)* in memory and automatically synchronize it with modifications applied to concrete syntax. IDE editors exploit the AST to facilitate source code navigation and refactorings, ranging from basic renamings to elaborate code transformations such as *method pull ups*.

Inter-language relations are a major problem in development of MLSS [23,13,12]. Since they are mostly implicit, they hinder modification and evolution of MLSS. An MLDE is an IDE that addresses this challenge by not only integrating tools into a uniform working experience, as IDEs do, but also by integrating languages with each other. MLDEs support across language boundaries the mechanisms implemented by IDEs for every language separately.

We surveyed IDEs, programming editors[3], and literature to understand the kind of development support they provide. We realized that 4 features, that

[3] IDEs: Eclipse, NetBeans, IntelliJ Idea, MonoDevelop, XCode. Editors: MacVim, Emacs, jEdit, TextWrangler, TextMate, Sublime Text 2, Fraise, Smultron, Tincta, Kod, gedit, Ninja IDE. (See project websites at: `www.itu.dk/~ropf/download/list.txt`)

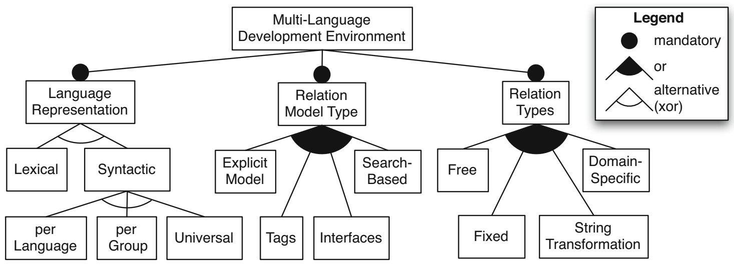


Fig. 2. Taxonomy for multi-language development environments

visualization, *navigation*, *static checking*, and *refactoring* are implemented by all IDEs and by some programming editors. Thus, in order to support developers best, MLSS need to consider delivering these features across language boundaries as their essential requirements:

1. *Visualization.* An MLDE has to highlight and/or visualize inter-language references. Visualizations can range from basic markers, as for instance in the style of Fig. 1 to elaborate visualization mechanisms such as treemaps [7].
2. *Navigation.* An MLDE has to allow navigating along inter-language relations. In Fig. 1, the developer can request to automatically open *button.js* and jump to line 8, when editing *Info.plist*. All surveyed IDEs allow to navigate source code. Further, IDEs allow for source code to documentation navigation, a basic multi-language navigation.
3. *Static Checking.* An MLDE has to statically check the integrity of inter-language relations. As soon as a developer breaks a relation, the error is indicated to show that the system will not run error free. All surveyed IDEs provide static checking by visualizing errors and warnings.
4. *Refactoring.* An MLDE has to implement refactorings, which allow easy fixing of broken inter-language relations. Different IDEs implement a different amount of refactorings per language. Particularly, rename refactorings seem to be widely used in current IDEs [21,31].

To address these requirements one needs to make three main design decisions: **a)** *How to represent different programming languages*? **b)** *How to inter-relate them with each other?* **c)** *Using which kind of relations?*

Systematizing the answers to these questions led us to a domain model characterizing MLDEs. We present this model in Fig. 2 using the feature modeling notation [5,16]. An MLDE always represents mograms based on the their language (*Language Representation*). Furthermore, an MLDE has to represent inter-language relations (*Relation Model Type*). This feature is essential for augmenting an IDE to an MLDE. Finally, an MLDE associates types to inter-language relations (*Relation Types*). An IDE first becomes an MLDE if it supports inter-language relations, i.e., as it implements an instance of this model.

The following subsections detail and exemplify the fundamental MLDE characteristics of our taxonomy. References to the surveyed literature are inlined.

2.1 Language Representation Types

We consider two main types of language representation, lexical and syntactic language representation. The former always works on an artifact directly without constructing a more elaborate representation, whereas the latter is always based on a richer data-structure representing mograms in a certain language. Syntactic language representation can represent mograms per language, per language group, or universally.

Lexical Representation. Most text editors, such as EMacs, Vim, and jEdit, implement lexical representation. Mograms are loaded into a buffer in a language agnostic manner. Syntax highlighting is implemented solely based on matching tokens. Due to lack of sufficient information about the edited mogram such editors provide limited support for static checking, code navigation, and refactoring.

Syntactic Representation. Per Language. Typical IDEs represent mograms in any given language using a separate AST, or a similar richer data structure capturing a mogram's structure; for instance Eclipse, NetBeans, etc. Unlike lexical representation, a structured, typed representation allows for implementation of static checking and navigation within and between mograms of a single language but not across languages. The advantage using per language representation, compared to per language group and universal representation, is that IDEs are easily extensible to support new languages.

Using models to represent source code is getting more and more popular[4]. This is facilitated by emergence of language workbenches such as EMFText, XText, Spoofax, etc.[5] The MoDisco [4] project, a model-driven framework for software modernization and evolution, represents Java, JSP, and XML source code as EMF models, where each language is represented by its own distinct model. These models are a high-level description of an analyzed system and are used for transformation into a new representation. The same principle of abstracting a programming language into an EMF model representation is implemented in JaMoPP [11]. Similarly, JavaML [3] uses XML for a structural representation of Java source code. On the other hand, SmartEMF [12] translates XML-based DSLs to EMF models and maps them to a Prolog knowledge base. The EMF models realize a per language representation. Similarly, we represent OFBiz' DSLs and Java using EMF models to handle inter-component and inter-language relations [23].

Syntactic Representation. Per Language Group. A single model can represent multiple languages sharing commonalities. Some languages are mixed or embedded into each other, e.g., SQL embedded in C++. Some languages extend others, e.g., AspectJ extends Java. Furthermore, languages are often used together, such as JavaScript, HTML, XML, and CSS in web development. Using

[4] Language workbenches mostly use modeling technology to represent ASTs. Therefore, we use the terms AST and model synonymously in this paper.
[5] See `www.languageworkbenches.net` for the annual language workbench competition.

a per language group representation allows increased reuse in implementation of navigation, static checks and refactoring in MLDEs, because support for each language does not need to be implemented separately.

For example, the IntelliJ IDEA IDE (`jetbrains.com/idea`), supports code completion for SQL statements embedded as strings in Java code. X-Develop [28,27] implements an extensible model for language group representation to provide refactoring across languages. AspectJ's compiler generates an AST for Java as well as for AspectJ aspects simultaneously. Similarly, the WebDSL famework represents mograms in its collection of DSLs for web development in a single AST [8]. *Meta*, a language family definition language, allows the grouping of languages by characteristics, e.g., object-oriented languages in *Meta(oopl)* [14]. The Prolog knowledge base in [12] can be considered as a language group representation for OFBiz' DSLs, used to check for inter-language constraints.

Syntactic Representation. Universal. Universal representations use a single model to capture the structure of mograms in any language. They can represent any version of any language, even of languages not invented yet. Universal representations use simple but generic concepts to represent key language concepts, such as blocks and identifiers or objects and associations. A universal representation allows the implementation of navigation, static checking, and refactoring only once for all languages. Except for TexMo, presented in Sec. 3, we are not aware of any IDE implementing a universal language representation.

The per group and the universal representations are generalizations of the per language representation. Both represent multiple languages in one model. Generally, there are two opposing abstraction mechanisms: *type abstraction* and *word abstraction* [29]. Type abstraction is a unifying abstraction, whereas word abstraction is a simplifying abstraction.

For example, both Java and C# method declarations can include modifiers, but the set of the actual modifiers is language specific. The synchronized modifier in Java has no equivalent in C#. Under the type abstraction, Java and C# method declarations can be described by a *Method Declaration* type and an enumeration containing the modifiers. In contrast, under the word abstraction, Java and C# method declarations would be described by a common simple *Method Declaration* type that neglects the modifiers. Obviously, in the type abstraction Java and C# method modifiers are distinguishable, whereas in the more generic word abstraction this information is lost.

Type abstraction is preferable for per group representations. Word abstraction is preferred for universal representations. The choice of abstraction influences the specificity of the representation, affecting the tools. Word abstractions are more generic than type abstractions. For instance, more cross-language refactorings are possible with the per group representation, while the refactorings in the systems relying on the universal representation automatically apply to a wider class of languages.

2.2 Relation Model Types

Software systems are implemented using multiple mograms. At the compilation stage, and often only at runtime, a complete system is composed by relating all the mograms together. Each mogram can refer to, or is referenced by, other mograms. An MLDE should maintain information about these relations. We observe four different techniques to express cross-language relations:

Explicit model. For example, *mega-models* [15], *trace models* [22,9], *relation models* [23], or *macromodels* [24]. All these are models linking distributed mograms together.

Tags. Hypertext systems, particularly HTML code links substructures or other artifacts with each other by tags. Tags define anchors and links within an artifact [10]. Hypertext systems interpret artifacts, anchors, and links. first after interpretation a link is established.

Interfaces. Interfaces are anchors decoupled from artifacts. An interface contains information about a development artifact's contents and corresponding locations. For example, OSGi manifest files or model and meta-model interfaces describe component and artifact relations [13].

Search-based. There is no persistent representation of relations at all. Possible relation targets are established after evaluating a search query. Search-based relations are usually used to navigate in unknown data. For example, in [30] relations across documents in different applications are visualized on user request by searching the contents of all displayed documents.

2.3 Relation Types

Here we elaborate on relations between mograms in different languages. Since we consider only textual languages all the following relation types relate strings.

Free relations are relations between arbitrary strings. They rely solely on human interpretation. For example, natural language text in documentation can be linked to source code blocks highlighting that certain requirements are implemented or that a programmer should read some documentation. Steinberger et. al. describe a visualization tool allowing to interrelate information across domains, even across concrete syntaxes [26]. Their tool visualizes relations between diagrams and data.

Fixed relations: Relations between equal strings are fixed relations. Fixed relations occur frequently in practice. For example, the relation between an HTML anchor declaration and its link is established by equality of a tag's argument names. Figure 1 shows an example of a fixed relation across language boundaries.

Waldner et. al. discuss visualization across applications and documents [30]. Their tool visualizes relations between occurrences of a search term matched in different documents.

String-transformation relations are relations between similar strings, or functionally related strings. For example, a Hibernate configuration file (XML) describes how Java classes are persisted into a relational database. The Hibernate framework requires that a field specified in the XML file has a corresponding get and set method in the Java class. A string fieldName in a Hibernate configuration file requires a getter with name getFieldName in the corresponding Java class. Depending on the direction, a string-transformation relation either attaches or removes get and capitalizes or decapitalizes fieldName.

Domain-Specific Relations (DSRs) are relations with semantics specific to a given domain or project. DSRs are always typed. Additionally, DSRs can be free, fixed or string-transformation relations. For example, a requirements document can require a certain implementation artifact, expressing that a certain requirement is implemented. At the same time, some Java code can require a properties file, meaning that the code will only produce expected results as soon as certain properties are in place. We consider any relation type hierarchy domain-specific, e.g., trace link classification [22].

The first three relation types, free, fixed, and string-transformation relations are untyped. They are more generic than DSRs, since they only rely on physical properties of relation ends. Fixed, string-transformation, and domain-specific relations can be checked automatically, which allows to implement tools supporting MLSS development, such as error visualization and error resolution.

3 TexMo as an MLDE Prototype

TexMo[6] addresses the requirements listed in Sec. 2 and it implements an instance of our MLDE taxonomy. TexMo uses a *key-reference* metaphor to express relations. In the example of Fig. 1, the command *declaration* takes the role of a *key* (Fig. 1a) and its uses are *reference* (Fig. 1b). TexMo relations are always many-to-one relations between *references* and *keys*. We summarize how TexMo meets the requirements presented in Sec. 2:

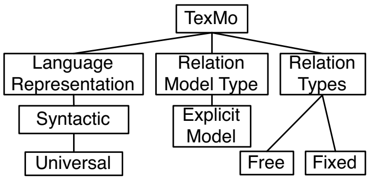
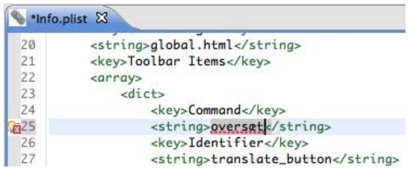


Fig. 3. The feature model instance describing TexMo in our taxonomy of MLDEs

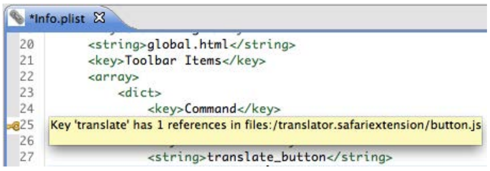
1. *Visualization.* TexMo highlights keys and references using gray boxes, see line 25 in Fig. 1a and line 8 Fig. 1b. Keys are labeled with a key icon and references are labeled by a book icon; see Fig. 1 left to line numbers. Inspecting markers reveals detailed information, e.g., how many references in which files refer to a key, see Fig. 4b.

[6] TexMo's source code including the text model and the relation model is available online at: `www.itu.dk/~ropf/download/texmo.zip`



```
<string>global.html</string>
<key>Toolbar Items</key>
<array>
    <dict>
        <key>Command</key>
        <string>oversæt</string>
        <key>Identifier</key>
        <string>translate_button</string>
```

(a) A broken relation between command declaration and its use, see Fig. 1b.



```
<string>global.html</string>
<key>Toolbar Items</key>
<array>
    <dict>
        <key>Command</key>
Key 'translate' has 1 references in files:/translator.safariextension/button.js
        <string>translate_button</string>
```

(b) Detailed relation information attached to a key marker.

Fig. 4. Visualization and information for inter-language relations

2. *Navigation.* Users can navigate from any reference to the referred key and from a key to any of its references. Navigation actions are called via the context menu.
3. *Static checking.* Fixed relations in TexMo's relation model (RM) are statically checked. Broken relations, i.e., fixed relations with different string literals as key and reference, are underlined red and labeled by a standard error indicator in the active editor, see Fig. 4a.
4. *Refactoring.* Broken relations can be fixed automatically using quick fixes. TexMo's quick fixes are key centric rename refactorings. Applying a quick fix to a key renames all references to the content of the key. Contrary, applying a quick fix to a reference renames this single reference to the content of the corresponding key.

On top of these multi-language development support mechanisms, TexMo provides syntax highlighting for 75 languages. GPLs like Java, C#, and Ruby, as well as DSLs like HTML, Postscript, etc. are supported. Standard editor mechanisms like undo/redo are implemented, too.

Universal Language Representation. The Text Model. TexMo implements a universal language representation since such an MLDE is easily applicable for development of any MLSS.

All textual languages share a common coarse-grained structure. The text model (Fig. 5), an AST of any textual language, describes blocks containing paragraphs, which are separated by new lines and which contain blocks of words. Words consist of characters and are separated by white-spaces. The only model elements containing characters are word-parts, separators, white-spaces, and line-breaks. Blocks, paragraphs, and word blocks describe the structure of a mogram. Separators are non-letters within a word, e.g., '/','.', etc., allowing represent of typical programming language tokens as single words.

TexMo treats any mogram as an instance of a textual DSL conforming to Fig. 5. For example, a snippet of JavaScript code `if(event.command ==` , line 8 in Fig. 1b, looks like: Block(Paragraph[WordBlock(Word[WordPart("if"), SeperatorPart(content:"("), WordPart("event"), SeperatorPart("."), WordPart("command"), WhiteSpace(" ")]), ...]) (using Spoofax [17] AST notation).

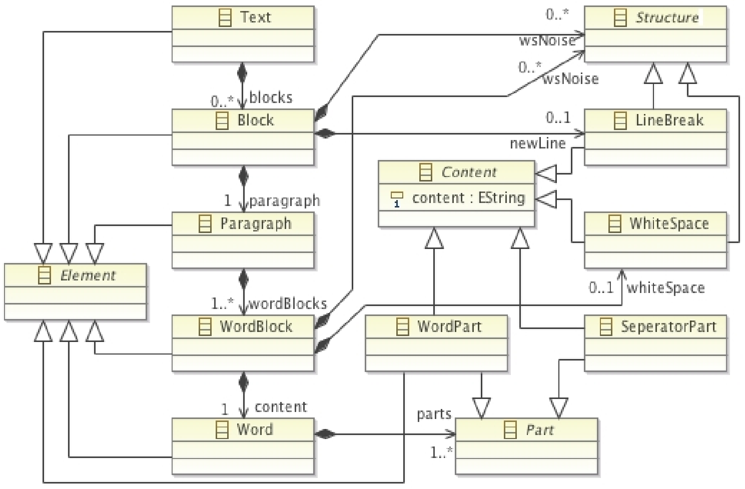


Fig. 5. The open universal model for language representation

An Explicit Relation Model. TexMo uses an instance of the Relation Model (RM) presented in Fig. 6 to keep track of relations between multi-language mogram code. Our RM allows for relations between mogram contents (ElementKey and ElementReference), between mogram contents and files (Artifacts) or components (Components), and between files and components. This allows for example to express relations in case mogram code requires another file, which occurs frequently, e.g., in HTML code.

The RM instance is kept as a textual artifact. The textual concrete syntax is not shown here, since the RM is not intended for human inspection. TexMo automatically updates the RM instance whenever developers modify interrelated mograms. That is, TexMo supports evolution of MLSS. Currently, the RM is created manually. TexMo provides context menu actions to establish relations between keys and references. Future versions of TexMo will integrate pattern based mining mechanism [23,9] to supersede manual RM creation.

Relation Types. TexMo's RM currently implements fixed and explanatory relations. Explanatory relations are free relations in our taxonomy. Keys and references of fixed relations contain the same string literal. Figure 1 shows a fixed relation and Fig. 4 shows a broken fixed relation. Explanatory relations allow to connect arbitrary text blocks with each other, for example documentation information to implementation code.

4 Evaluation

In this section we discuss TexMo's applicability. First, we evaluate TexMo's language representation mechanism, i.e., its representation of mograms as text

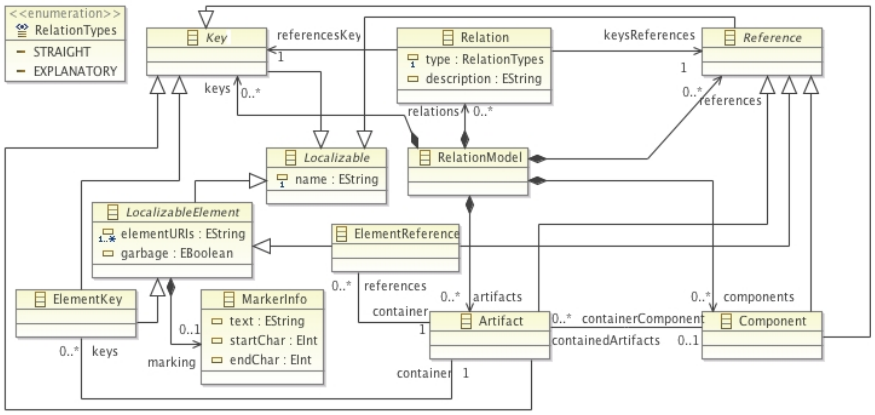


Fig. 6. TexMo's explicit relation model

models. Second, we provide preliminary evidence on the feasibility of TexMo by testing user acceptance. Furthermore, we discuss applicability of TexMo's relation model with respect to keeping inter-language relations while testers are using TexMo.

The subject used for this evaluation is the open-source web-based bug-tracking system, JTrac. JTrac's code base consists of 374 files. The majority of files, 291, contain source code in Java (141), HTML (65), property files (32), XML (16), JavaScript (8), and 29 other source code files such as Shell scripts, etc. Similar to many web-applications, JTrac implements the model-view-controller (MVC) pattern. This is achieved using popular frameworks: Hibernate (`hibernate.org`) for OR-Mapping and Wicket (`cwiki.apache.org/WICKET/`) to couple views and controller code. The remaining 83 files are images and a single jar file. We did not consider these files in our evaluation since they do not contain information in a human processable, textual syntax. Clearly, JTrac is an MLSS.

4.1 Universal Language Representation

To evaluate TexMo's universal language representation, we manually opened all 291 mograms with the TexMo editor to check if a correct text model can be established. By correct we mean that any character and string in a source code artifact has a corresponding model element in the text model, which in turn allows the RM to interrelate mograms in different languages. The files used are available at: `www.itu.dk/~ropf/download/jtrac_experiment.zip`.

We concluded that all 291 source code files can be opened with the TexMo editor. For all files a correct text model has been established.

4.2 User Test

To test user acceptance, we let 11 testers perform three typical development tasks. The testers included 4 professional developers, 3 PhD students, and 4 undergraduate students, with median 3 years of working experience as software developers.

Using only a short tutorial, which explains TexMo's features the testers had to work on the JTrac system. First, they had to find and remove a previously injected error, a broken fixed relation. Second, they had to rename a reference and fix the now broken relation. Third, they had to replace a code block, which removes two keys. We captured the screen contents and observed each tester. After task completion, each tester filled out a questionnaire. Questions asked for work experience, proficiency in development of MLSS using Java, HTML, and XML. Additionally, two open questions on the purpose of the test and on the usefulness of TexMo where asked. After the completion of questionnaires we had a short, open discussion about TexMo where we took notes on tester's opinions.

We conclude that the testers understand and use MLDE concepts. Seven testers applied inter-language navigation to better understand the source code, i.e., to inspect keys and references whenever an error was reported. Furthermore, another seven used rename refactorings to securely evolve cross-language relations in JTrac. All testers were able to find all errors and to fix them. In the following we quote a selection of the testers arguing about usefulness of TexMo (we avoid quoting complete statements for the sake of brevity). Their statements indicate that visualization, static checking, navigation and refactoring across language boundaries are useful and that such features are missing in existing IDEs.

Q: *"Do you think TexMo could be beneficial in software development? Why?"*
A$_1$: *"TexMo's concepts are really convincing. I would like to have a tool like this at work."*
A$_2$: *"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."*
A$_3$: "[TexMo] *improves debugging time by keeping track of changes on source code written in different programming languages that are strongly related. I do not know any tool like this."*
A$_4$: *"I see* [TexMo] *useful, especially when many people work on the same project, and, of course, in case the projects gets big."*
A$_5$: *"I did development with Spring and a tool like TexMo would solve a lot of problems while coding."*
A$_6$: *"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.*
A$_7$:"[TexMo] *solves* [a] *common problem experienced when software project involves multiple languages."*

Robustness of the Relation Model. To run the user test and to demonstrate that the RM can express inter-language relations in an MLSS, we established a RM relating 9 artifacts containing 51 keys, 87 references, via 87 fixed relations with each other. The RM relates code in Java, HTML, and properties files with each other. We did not aim for a *complete* RM, since we focus on demonstrating TexMo's general applicability. After the testers had finished their development tasks, we inspected the RMs manually to verify that they still correctly interrelate keys and references.

We conclude that TexMo's RM is robust to modifications of the MLSS. After modification operations, all relations in the RM correctly relate keys and references across language boundaries.

A common concern of the testers related to replacing a code block containing multiple keys with a new code block, where TexMo complains about a number of created dangling references in corresponding files. We did not implement a feature to automatically infer possible keys out of the newly inserted code, since we consider this process impossible to automate completely.

4.3 Threats to Validity

The code base of JTrac might be to small to allow to generalize that any textual mogram in any language can be represented using TexMo's text model. However, we think that nearly 300 source code files in 15 languages gives a rather strong indication. The RM used for the user tests might be to small and incomplete. We were not interested in creating a complete RM, but only concerned about its general applicability.

To avoid direct influence on the testers in an oral interview, we used a written questionnaire. All quotes in the paper are taken from this written data.

5 Related Work

Strein et. al. argue that contemporary IDEs do not allow for analysis and refactoring of MLSS and thus are not suitable for development of such. They present X-Develop an MLDE implementing an extensible meta-model [28] used for a syntactic per language group representation. The key difference between X-Develop and TexMo is the language representation. TexMo's universal language representation allows for its application in development of any MLSS regardless of the used languages. Similarly, the IntelliJ IDEA IDE implements some multi-language development support mechanisms. It provides multi-language refactorings across some exclusive languages, e.g., HTML and CSS. Unlike in TexMo, these inter-language mechanisms are specific to particular languages since IntelliJ IDEA relies on a per language representation.

Some development frameworks provide tools to enhance IDEs. Our evaluation case, JTrac relies on the web framework Wicket. *QWickie* (`code.google.com/p/qwickie`), an Eclipse plugin, implements navigation and renaming support between inter-related HTML and Java files containing Wicket code. The drawback of framework-specific tools is their limited applicability. QWickie cannot be used for development with other frameworks mixing HTML and Java files.

Chimera [2] provides hypertext functionality for heterogeneous *Software Development Environments (SDE)*. Different programs like text editors, PDF viewers and browsers form an SDE. These programs are viewers through which developers work on different artifacts. Chimera allows for the definition of anchors on views. Anchors can be interrelated via links into a hyperweb. TexMo is similar in that models of mograms can be regarded as views where each model element can serve as an anchor for a relation. Chimera is not dynamic. It does not automatically evolve anchors while mograms are modified. Subsequent to modifications, Chimera users need to manually reestablish anchors and adapt the links to it. Contrary, TexMo automatically evolves the RM synchronously to modifications applied to mograms. Only after deleting code blocks containing keys, users need to manually update the dangling references.

Meyers [20] discusses integrating tools in multi-view development systems. One can consider language integration as a particular flavor of tool integration. Meyers describes basic tool integration on file system level, where each tool keeps a separate internal data representation. This corresponds to the per language representation in our taxonomy. Meyers' *canonical* representation for tool integration corresponds to our universal language representation. Our work extends Meyers work by identifying a per language group representation.

6 Conclusion and Future Work

We have presented a taxonomy of multi-language development environments, and TexMo, an MLDE prototype implementing a universal language representation, an explicit relation model supporting free and fixed relations. The taxonomy is established by surveying related literature and tools. We have also argued that implementation of TexMo meets is design objectives and evaluated adequacy of its design. By itself TexMo demonstrates that design of useful MLDEs is feasible and welcomed. We reported very positive early user experiences.

To gather further experience, we plan to extend TexMo with string-transformation and domain-specific relations and compare it to an MLDE using a per language representation. We realized that it is costly to keep an explicit RM updated while developers work on a system, especially the larger a RM grows. Therefore, we will experiment with a search-based relation model. This will also overcome the vulnerability of an explicit RM to changes applied to mograms outside the control of the MLDE.

Note, TexMo's RM does not only allow the interrelation of mograms of different languages but also of mograms in a single language. We do not focus on this fact in this paper. However, this ability can be used to enhance and customize static checks and visualizations beyond those provided by current IDEs without extending compilers and other tools.

While working with TexMo we realized that a universal language representation is favorable if an MLDE has to be quickly applied to a wide variety of systems with respect to the variety of used languages. Furthermore, there is a trade-off between the language representation mechanism and the richness of the tools an MLDE can provide. Basic support, like visualization, highlighting, navigation and rename refactorings, can be easily developed on any language representation, with very wild applicability if the universal representation is used. More complex refactorings require a per group or a per language representation.

In future we plan to build support to automatically infer inter-language relations. Fixed and string-transformation relations can be automatically established by searching for equal or similar strings. This process is not trivial as soon as a language provides for example scoping. Then inferring inter-language relations has to additionally consider language specific scoping rules. Inferring domain-specific relations has to rely on additional knowledge provided by developers, for example as patterns [23], which explicitly encode domain knowledge. Inferring free relations is probably not completely automatable but relying on heuristics and search engines could result in appropriate inter-language relation candidates.

Acknowledgements. We thank Kasper Østerbye, Peter Sestoft and David Christiansen for discussion and feedback on models for representation of language groups and for feedback on the TexMo prototype. EMFText developers have provided technical support during TexMo's development. Chris Grindstaff has developed the Color Editor (`gstaff.org/colorEditor`), parts of which were reused for TexMo's syntax highlighting. Last but not least, we also thank all the testers participating in the experiment.

References

1. Zend Technologies Ltd.: Taking the Pulse of the Developer Community, `http://static.zend.com/topics/zend-developer-pulse-survey-report-0112-EN.pdf` (February 2012)
2. Anderson, K.M., Taylor, R.N., Whitehead Jr., E.J.: Chimera: Hypermedia for Heterogeneous Software Development Enviroments. ACM Trans. Inf. Syst. 18 (July 2000)
3. Badros, G.J.: JavaML: A Markup Language for Java Source Code. Comput. Netw. 33 (June 2000)
4. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In: Proc. of the IEEE/ACM International Conference on Automated Software Engineering (2010)
5. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications (2000)
6. Erlikh, L.: Leveraging Legacy System Dollars for E-Business. IT Professional 2 (May 2000)
7. de Figueiredo Carneiro, G., Mendonça, M.G., Magnavita, R.C.: An experimental platform to characterize software comprehension activities supported by visualization. In: ICSE Companion (2009)
8. Groenewegen, D.M., Hemel, Z., Visser, E.: Separation of Concerns and Linguistic Integration in WebDSL. IEEE Software 27(5) (2010)
9. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-modelling: From Theory to Practice. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 376–391. Springer, Heidelberg (2010)
10. Halasz, F.G., Schwartz, M.D.: The Dexter Hypertext Reference Model. Commun. ACM 37(2) (1994)
11. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 374–383. Springer, Heidelberg (2010)
12. Hessellund, A.: SmartEMF: Guidance in Modeling Tools. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (2007)

13. Hessellund, A., Wąsowski, A.: Interfaces and Metainterfaces for Models and Metamodels. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 401–415. Springer, Heidelberg (2008)
14. Holst, W.: Meta: A Universal Meta-Language for Augmenting and Unifying Language Families, Featuring Meta(oopl) for Object-Oriented Programming Languages. In: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (2005)
15. Jouault, F., 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)
16. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (1990)
17. Kats, L.C.L., Visser, E.: The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In: OOPSLA (2010)
18. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels (2008)
19. Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. Commun. ACM 21 (June 1978)
20. Meyers, S.: Difficulties in Integrating Multiview Development Systems. IEEE Softw. 8 (1991)
21. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: Proc. of the 31st International Conference on Software Engineering (2009)
22. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering. Softw. Syst. Model. 10 (October 2011)
23. Pfeiffer, R.-H., Wąsowski, A.: Taming the Confusion of Languages. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 312–328. Springer, Heidelberg (2011)
24. Salay, R., Mylopoulos, J., Easterbrook, S.: Using Macromodels to Manage Collections of Related Models. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 141–155. Springer, Heidelberg (2009)
25. Standish, T.A.: An Essay on Software Reuse. IEEE Trans. Software Eng. (1984)
26. Steinberger, M., Waldner, M., Streit, M., Lex, A., Schmalstieg, D.: Context-Preserving Visual Links. IEEE Transactions on Visualization and Computer Graphics (InfoVis 2011) 17(12) (2011)
27. Strein, D., Kratz, H., Lowe, W.: Cross-Language Program Analysis and Refactoring. In: Proc. of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (2006)
28. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An Extensible Meta-Model for Program Analysis. IEEE Trans. Softw. Eng. 33 (September 2007)
29. Wagner, S., Deissenboeck, F.: Abstractness, Specificity, and Complexity in Software Design. In: Proc. of the 2nd International Workshop on the Role of Abstraction in Software Engineering (2008)
30. Waldner, M., Puff, W., Lex, A., Streit, M., Schmalstieg, D.: Visual Links Across Applications. In: Proc. of Graphics Interface (2010)
31. Xing, Z., Stroulia, E.: Refactoring practice: How it is and how it should be supported — an Eclipse case study. In: Proc. of the 22nd IEEE International Conference on Software Maintenance (2006)