Architecture Conformance Checking of Multi-Language Applications

Razieh Rahimi
School of Electrical and
Computer Engineering
University of Tehran
North Karegar Avenue, Tehran, Iran
Email: r.rahimi@ece.ut.ac.ir

Ramtin Khosravi
School of Electrical and
Computer Engineering
University of Tehran
North Karegar Avenue, Tehran, Iran
Email: rkhosravi@ece.ut.ac.ir

***Abstract*—As the development in a software project goes on, the structure of the implemented code diverges from the intended architecture. To prevent this, architecture conformance methods are used to check if the source code complies with the architecture. In the development of today's enterprise applications, general-purpose programming languages are used along with a number of domain specific languages. So, there is a need for a conformance checking method to support multi-language source artifacts. We present a model-based approach for checking cross-language architecture conformance rules. Our method is extensible, in the sense that it is independent of the specific set of languages used in the project.**

I. Introduction

Software architecture is the primary artifact designed during software development for reasoning about software properties either functional or non-functional such as availability or modifiability. Architectural or design decisions have effects on several implementation artifacts. This may cause the developers to mistakenly violate these decisions, resulting in architecture erosion. This gap between implementation and architecture causes the system to fail to satisfy some of the intended non-functional properties. So, we need the implementation to be in conformance with the architecture.

Today, one can rarely find an enterprise-scale application fully written in a single programming language. The widespread use of reusable frameworks makes developers of the project use several small languages besides the main programming language. The frameworks usually allow the developers to describe a part of the configuration or definition of the application in a separate artifact outside the main application source code. This allows the developers to change the description dynamically without re-compiling (or even restarting) the application. Also, the syntax of the description language is usually in a declarative style and is more readable than the equivalent code written in a general-purpose programming language. For example, in an enterprise information system, the architect may decide to use a third-party framework that allows the developers to define the flow between user-interface pages in a language with a state machine-like structure. As an example, Fig. 1 shows part of such a description in the Spring Web Flow [1] framework.

```
<flow>
    <on-start>
        <evaluate expression="bookingService.createBooking(
                                    hotelId, currentUser.name)"
                  result="flowScope.booking" />
    </on-start>

    <view-state id="enterBookingDetails">
        <transition on="submit" to="reviewBooking" />
    </view-state>

    <view-state id="reviewBooking">
        <transition on="confirm" to="bookingConfirmed" />
        <transition on="revise" to="enterBookingDetails" />
        <transition on="cancel" to="bookingCancelled" />
    </view-state>

    <end-state id="bookingConfirmed" />

    <end-state id="bookingCancelled" />
</flow>
```

Fig. 1. A simple flow description in Spring Web Flow [1]

Moreover, the rising use of Domain-Specific Languages (DSLs) in software development industry [2], adds to the number of languages used in a typical project. For example, the developers may define a set of business rules in an external DSL to allow the domain experts to dynamically modify some aspects of the program behavior during the application maintenance phase. These languages are DSLs related to business domains (e.g. financial or health) while the configuration or description languages mentioned before can be considered as DSLs related to technical domains (e.g. persistence or web flow). Throughout this paper, the term DSL is referred to the languages in both classes in general.

As several languages are used in developing software systems, architectural rules may affect artifacts of multiple languages. For example, in an application using Spring Web Flow, there may be a rule such as "*when making a transition from a web page in a subsystem to a web page in another subsystem, the* leaveContext *method of the source subsystem's session façade shall be called*". Checking this rule requires inspecting both the flow description artifacts as well as the Java source code of the subsystems. Therefore, in addition to rules over individual languages, there are some rules across several languages. To support cross-language rules, we need to

have a method to build a unified model from different artifacts.

Moreover, as any project may use a different set of DSLs, a requirement for an effective conformance checking method for enterprise-scale application is to be independent of the specific languages. In other words, such a method should have the capability of being customized according to the set of languages used in the project, which means the ability to dynamically add new languages to the toolset. Most of the existing methods focus on checking the compliance of source code written in one programming language with the architecture. So, they may not be adequate for enterprise-scale applications.

In this paper, we present a model-based approach to architecture conformance checking capable of handling cross-language rules. The proposed approach uniformly represents various languages in the system in the form of a meta-model of the relevant source artifacts at the desired level of detail. Rules are described in terms of the elements of this high-level uniform representation. We follow the well-known standards Meta-Object Facility (MOF) [3] and Object Constraint Language (OCL) [3] to represent the meta-model and the rules respectively. Each source artifact is abstracted into a separate model as specified by its meta-model. Then, these models are integrated into a unified model according to the integrated meta-model of the system. The rules are specified in terms of OCL expressions, which are evaluated on the unified model to check the conformance of the source code to the architecture.

The rest of this paper is organized as follows. The next section provides a background on conformance checking. Section III presents an example case. Section IV explains our approach and the implementation method. Section V presents a selected subset of architectural constraints and their specifications. Section VI briefly reviews the related work. Finally, we conclude the paper and discuss the benefits and drawbacks of our method in Section VII.

II. Conformance Checking of Architecture

Architecture is the set of design decisions that the architect or expert developers would like to get right as early as possible in the development of systems. These include the decomposition of system into modules and specification of allowed interactions among them [4]. The implementation is constrained by these decisions and is supposed to exhibit the architecture to guarantee the intents behind the decisions. As developers implement the designed elements and make lower-level decisions, they may mistakenly violate the designed structure. Even they may do this intentionally, for example, they may introduce a direct dependency to improve performance. But through their lack of an overall view of the system design, they violate an architectural decision that affects the system's maintainability. Therefore, without source code architecture conformance, architecture becomes less useful.

Conformance checking is a process that can investigate consistency between different artifacts in a wide scope. The main use is in ensuring that the software is implemented according to the architecture, which is the foremost high-level artifact. Model-driven approaches generate multiple models by converting high-level model to models at lower levels of abstraction. Consistency between models at different levels of abstraction can also be a part of conformance checking process.

Architecture is described using different views and each imposing constraints on the implementation. Violations of these constraints can be categorized according to the architectural aspect that is the concern of the corresponding view. For instance, structural violations are usually related to the *module viewpoint* of architecture in SEI documentation model [5] or *development view* of the "viewpoints and perspectives" method [6]. Structural violations can be resulted from constraints like "Presentation layer must call domain services through controllers". Some constraints address runtime aspects of the system like imposing certain topological constraints on the components and connectors of the system. Some need runtime information along with static information to be checked, like "At the end of each control path of every service implementation method, if a transaction is opened, it must be eventually committed or rolled back". Some may constrain naming convention of software elements like "The name of control classes must end with `ctrl`". An architecture conformance approach can investigate compliance with different architecture viewpoints.

Architecture conformance approaches can also have features like traceability of violations to source code so that the part of the source code that causes the violation can be identified and corrected easily. Another useful feature is incremental refinement during development process. If violations can be detected as early as possible by continuously checking the conformance, their ripple effects will be less and its source can be corrected more simply. The approach can be integrated with the IDE to automatically checks the source code.

Several approaches have been proposed to *enforce* architectural rules or *check* the compliance of source code with the architecture [7]. They can be categorized as either static or dynamic according to the time that they can be applied [8]. Static conformance checking is done without executing code, while dynamic approaches use execution information. For instance, Reflexion model [9] is a static approach that checks compliance of the source model with the architecture model. Architecture-centric Model-driven development approaches [10] can also be used to enforce the architecture, but in cases that the generated code is completed manually by the developers, implementation may diverge from the architecture. The feature of defining compile time declaration of Aspect Oriented Programming (AOP) can also be used to check the architecture compliance [7].

III. An Example Case

This section introduces a simple case that is referred to in the other sections to explain the use of the proposed approach. This case study consists of a small programming language called *μpl* and a DSL called *μflow*. The programming

```
module Adder {
    var sum
    fun add(num1 num2) {
        sum = doAdd(num1 num2)
        return 2
    }
    fun again() {
        return 1
    }
    fun doAdd(a b) {
        return a + b
    }
}
```

Fig. 2. *μpl* sample script

```
flow addition(Adder) {
    page(1) {
 input num1
    input num2
    button add
}
    page(2) {
        output sum
        button again
    }
}
```

Fig. 3. *μflow* sample script

language is used to develop domain logic and the DSL has been designed to specify a flow of user interface pages and the related module handlers of each page action.

Fig. 2 shows a part of a script written in *μpl* that defines an `Adder` module with three methods called `add`, `again` and `doAdd`. All variables are of Integer type. Fig. 3 shows a script in *μflow* language. In this script, a flow of user interface pages is specified that includes two pages. The `addition` flow uses `Adder` module to handle user requests. When user presses a button in a page, a function of the specified module with the same name as the button will be called to take appropriate action. The return value of that function will determine the ID of a page of flow to be represented next. For example, when user presses the `add` button of page 1 the `add` function of `Adder` module will be called. This function calculates the sum of the two inputs and returns 2, so that when the control returns to the flow, the page with ID 2 will be represented. Several cross-language rules on source code written in these two languages will be described in Section V.

IV. Our Approach to Conformance Checking

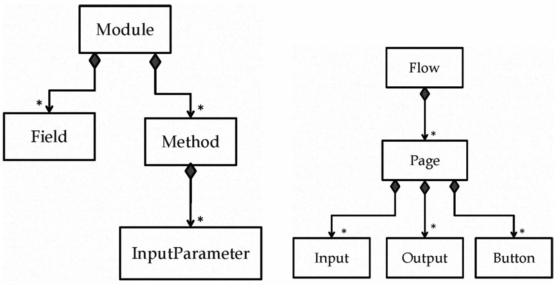
We assume that the target software is comprised of a main codebase in a general-purpose programming language and a set of artifacts in form of domain-specific descriptions. Some of these descriptions are based on off-the-shelf reusable frameworks while others are based on domain-specific languages specifically defined for the particular software under development. Our goal is to provide a tool-based approach for the software architects to validate cross-language architectural rules. We intend to use the proposed approach for modeling the source code and specifying rules independently of the languages used in developing, so that it can be customized to support domain-specific languages that are being designed specifically for a system. The approach should provide a uniform way for the inclusion of newly designed languages so the architects/designers can add them dynamically in design phase to express cross-language conformance rules.

This paper introduces a two-phase method to check conformance rules on multi-language software systems. In the first phase, the architect is supposed to provide the infrastructure to use the approach for the target system. The architect defines the languages used in the project and provides an integrated system-wide meta-model and specifies architectural rules. In the next phase, the approach can be used to check the rules anytime during the system development. The code base is processed to extract elements related to the rules and build a higher level object model of source code. Then, the rules are checked on the generated object model. The detailed steps of each phase are described in the following subsections.

The implementation is often more detailed than the architecture. Some of these details are irrelevant to a specific architecture viewpoint. So, conformance check needs inspection of the source code to build a higher-level model. Following valuable practices of Model Driven Software Development (MDSD) [10], we choose to model the source code based on a meta-model that describes the structural view of architecture. MDSD allows construction of models of a system according to the meta-models and transforms them automatically to get models at the desired level of abstraction. Following a similar approach in conformance checking, the architect describes the structural aspect of a system in a meta-model and then specifies rules in terms of its elements. It brings the need for a uniform description of the meta-model. One way is to follow the widely-used standard for modeling, Meta-Object Facility (MOF) developed by the Object Management Group (OMG). MOF provides a generic framework to define meta-models of different modeling languages and lies at the heart of Model Driven Architecture (MDA) [3]. We decided to adopt MOF as a language for specifying structural view of architecture. It provides a unified schema for expressing and checking structural rules. Once the architecture is modeled using MOF, the rules can be expressed in terms of its elements. We choose Object Constraint Language (OCL), another standard by OMG, as a platform-independent language to describe constraints over the system meta-model. OCL is a declarative language that can be used in conjunction with UML or any MOF-based meta-model to express constraints and object query expressions on their meta-elements and thus precisely define their semantics.

Formulating the inputs of conformance checking in standard languages (MOF and OCL) has other advantages in addition to uniformity such as vendor independence, extensive tool support and short learning curve. So, it would be more easily accepted by the architect to specify an architecture and the rules in this way compared to using a newly defined syntax.

As various languages used in development have their own structure, the system meta-model can be obtained by integrating the individual language meta-models. These meta-models should be designed at an appropriate level of abstraction



(a) *μpl* meta-model (b) *μflow* meta-model

Fig. 4. languages' metamodels

according to the corresponding rules. To provide an integrated meta-model for a system, the architect should follow the steps below.

A. Phase 1: Building the Infrastructure

Step 1: Generating Language Parsers

Building the infrastructure means to specify how to process the language scripts for constructing their model. The tool requires a model extractor for each language used. These models are based on each language meta-model. So, in addition to the extractor, their meta-models should also be defined. Because these meta-models will be integrated to build the system meta-model, they should have a unified format. The Eclipse Modeling Framework (EMF) is a meta-modeling framework that implements MOF. Being based on EMF, our toolset requires language meta-models to be represented in EMF. EMF also has the facility for generating Java code corresponding to the meta-model which will be used to generate the object model of language scripts. For the explained case study, the meta-model of *μpl* and *μflow* languages should be specified independently in EMF format. EMF meta-model can be generated from either a Rational Rose model or a set of annotated Java interfaces and classes. Fig. 4 shows the meta-models of *μpl* and *μflow*.

The architect/designer is free to determine the abstraction level of meta-model by putting any element that is significantly related to the structural view of architecture in the meta-model. The set of architectural rules give a good criteria to decide on elements of the meta-model. Assuming that we only have one rule on the example case study, that "the getter methods should have no input parameter", we just need to have *method* and *inputParameter* elements from the *μpl* meta-model shown in Fig. 4.(a). In some cases that the rules include more implementation details, the provided meta-model can tend towards Abstract Syntax Tree (AST).

The model extractor will populate the semantic model of the scripts of a language according to its specified meta-model. There is a module corresponding to each input language to derive its model. We provide different ways for the architect to define the model extractor of the new languages. The architect can provide a parser that directly populates the model in a single pass or a two-pass parser having AST as an intermediate model. In the second case, the extractor performs syntactic analysis on the input scripts to produce the AST. It then uses an associated tree walker to generate the model. For AST generation, the architect is free to introduce the abstract syntax grammar or to provide a parser. In the case that a grammar is used, a corresponding parser will be generated using ANTLR [11] which requires to have a grammar that is compatible with ANTLR BNF grammar format along with AST construction operators or rewrite rules. If there is not any tree construction rule in the grammar, the parser will return an AST that has nodes for each term in the right-hand side of the rules. The extractor sets the output option to "AST" and the type of generated tree node to our custom implemented "`TreeNode`" which extends "`CommonTree`". Our "`TreeNode`" implementation provides some functions that facilitate traversing of AST, like "`getChildren()`", which returns the children of a node.

ANTLR requires an adaptor as a tree navigator so that a related adaptor that creates the defined node would be set as the generated parser tree adaptor to inform ANTLR of its use. For our case study, the descriptions of *μpl* and *μflow* grammar in BNF format are shown in Fig. 5 and Fig. 6 respectively. In each rule, what follows "→" is the syntax for tree construction. Using these grammar definitions, the tool will produce a parser for each language that generates AST as output. The second rule of *μpl* grammar, will result in creation of a `microModule` node and its name and its declared fields and methods as its children. The generated ASTs of scripts are given to another program to populate its model. This program generates object model of the scripts by taking appropriate actions according to each AST node's type. For example, we have implemented a method that instantiates an object of microModule element of meta-model for each `microModule` node.

In another case, the architect can provide a parser that directly populates the model in a single pass, but the previous case is more common. The way that the tool infrastructure is set, facilitates supporting new languages. Providing parsers that populate the semantic model of languages is a straightforward task, especially for domain-specific languages, because they are supposed to ease the generation of a part of the overall domain model.

Step 2: Define Meta-model Integration Scheme

After parsing the scripts and populating their semantic models, they should be integrated based on the system meta-model. We need a unified way of describing relationships between their meta-models to be able to automatically integrate the models of different scripts. Model elements that refer to elements of other models are supposed to extend a specially provided EObject of EMF, called `Reference`. For example, in our case study, the *μflow* flow elements that refer to *μpl* module elements should extend `Reference` element. This way, we can have an integrated meta-model, generated automatically from individual meta-models.

Step 3: Describing Rules

Once we have an integrated meta-model that specifies the structure of a system, the constraints that it imposes, the rules on its elements and relationships among them can be

```
microPL : microModule* -> ^(MODULE_LIST microModule*);
microModule : 'module' n=ID '{' fieldDecl* methodDecl* '}' -> ^(MICROMODULE $n ^(FIELDS fieldDecl*) ^(METHODS methodDecl*));

fieldDecl : 'var' n=ID -> ^(FIELD $n);
methodDecl : 'fun' n=ID '(' params* ')' '{' body '}' -> ^(METHOD $n ^(PARAMS params*) ^(EXPRESSIONS body));


params  : n=ID -> ^(PARAM $n);
body : statement*;
statement : conditional | repetitive | assignment | returnSts | '{' statement* '}';
conditional : 'if' '(' condExpr ')' statement 'else' statement;
repetitive : 'do' '(' condExpr ')' statement;
assignment : ID '=' expr;
returnSts : 'return' expr;
condExpr : expr '==' expr;
expr : ID | ID '(' ID* ')' | DIGIT | ID '+' ID | ID '-' ID | n=ID '.' ID '(' expr* ')' -> ^(DEPENDENCY $n);
```

Fig. 5. *μpl* grammar

```
pavaFlow : pFlow* -> ^(FLOW_LIST pFlow*);
pFlow : 'flow' n=ID '(' m=ID ')' '{' page* '}' -> ^(FLOW $n ^(MODULENAME $m) ^(PAGES page*));

page : 'page' '(' n=ID ')' '{'  elem* '}' -> ^(PAGE $n ^(ELEMENTS elem*)) ;
elem : input -> ^(INPUT input*) | output -> ^(OUTPUT output*) | button -> ^(BUTTON button*);

input : 'input' n=ID -> ^($n);
output : 'output' n=ID -> ^($n);
button : 'button' n=ID -> ^($n);
```

Fig. 6. *μflow* grammar

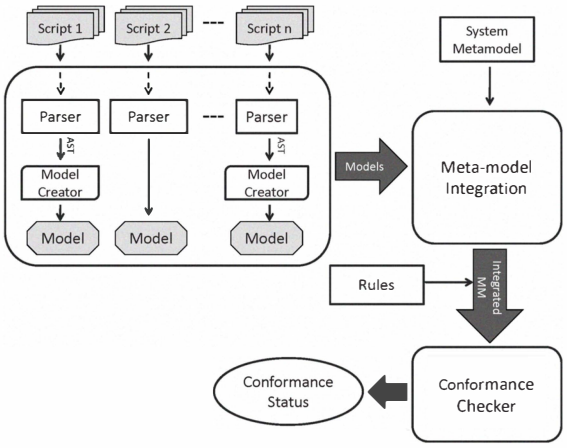


Fig. 7. Conformance checking process

expressed in OCL. For instance, one rule on the described meta-model for our example case can be derived from the fact that user interface pages in a flow are referred to by their ID attribute, so they need to have a unique ID. This rule can be specified in OCL as: "`context Page inv: Page.allInstances() -> isunique (id)`". The rule specification is described in more details in Section V.

B. Phase 2: Rule Checking

After building the infrastructure, the tool can be used to inspect the source code to check if it follows the specified rules. This checking is done in three steps, as illustrated in Fig. 7.

Step 1: Building Models for Each Artifact

In the first step, the appropriate parser for each input script will be called independently. Assuming that we have scripts shown in Fig. 2 and Fig. 3, *μpl* parser will interpret the scripts of this language and the associated *μflow* parser will parse *μflow* source files. As explained before, the output of these parsers is an AST. Thereafter, another programs that take these ASTs will be called to generate the semantic model of scripts. After executing the associated programs we have models of each individual language.

Step 2: Integrating Models

The second step deals with integrating the generated models. The code for linking the objects will be generated in accordance with the association between the reference elements in the languages' models. After populating the object models of the entire scripts, the linking code is executed.

Step 3: Checking Rules

In this step, the rules will be examined on the generated object model of the code base. For parsing and evaluation of the OCL constraints and queries, we use MDT-OCL plug-in of Model Development Tools (MDT) [12].

V. Specifying Rules

In this section, we present some examples to illustrate the use of OCL for specifying rules. Based on the level of abstraction of the rules, they are divided into the following three categories.

A. Architectural Rules

A typical example of architectural rules is the specification of allowed dependencies between layers. Following this, we consider a rule that states that "Data layer should not be dependent on the UI layer". The mapping of layers to *μpl* modules is specified as follows:

i) *Controller* modules are those associated with the flow declaration, *ii*) UI layer is the collection of *controller* modules, *iii*) Data access modules are those that are directly dependent on the DB module, *iv*) Data layer is the collection of data access modules plus the DB module. We assume that DB is the only module that accesses the database directly.

Architectural level dependencies result from dependencies between the implementation artifacts. Therefore, to compute the dependency of architectural elements, like layers, we need information about the dependencies between implementation level elements belonging to them. The syntax for expressing dependencies between implementation level elements like method calls varies among languages. Due to the purpose of being independent of languages, we assume that the expressions causing dependencies will be determined from the grammar of the languages. For instance, the format of method call in *μpl* is based on the "`expr : ID '.' ID '(' expr* ')'`" grammar rule. We should notify the model generator to model the dependency based on this rule, so that we complete it using the tree construction rule "`^(DEPENDENCY $n)`" to let the parser add DEPENDENCY nodes to the AST. The model generator sets the dependency of model element instances according to DEPENDENCY nodes.

The above rule imposes a constraint on a collection of modules that constitute the data layer. Hence, before expressing the constraint we should define how we obtain this collection. For this purpose, we use OCL query expressions. The query that results in the data layer can be expressed as "`MicroModule.allInstances() -> select (c: MicroModule| c.depends -> exists (d: MicroModule | d.name = 'DB'))`". Then we specify the mentioned constraint on this group of entities as "`context MicroModule inv: MicroModule.allInstances() -> collect (depends) -> intersection (MicroModule.allInstances() -> select (c: MicroModule | c.flows -> size() >= 1)) -> isEmpty()`" and we evaluate the constraint on the result collection of the query.

B. Design Rules

Design rules are mostly derived from design patterns and they address elements directly realized in the source code. Some examples of design rules are as follows.

Existence of handler module: each flow of a user interface page defines a module as a handler of users' requests. The reference to this module is just by its name, so there is a rule that a module with that name must exist. This rule can be specified in OCL as: "`context Flow inv: self.moduleName = self.microModule.name`".

Existence of a corresponding action handler: when users make their requests by pushing a button of a page, a method with the same name as the button in the corresponding module of the page's flow would be called to handle the request. Then, the contents of the text boxes, which are defined by the `input` keywords, are passed to the related method as input parameters. This leads to the following rule: "*For each button, there must be a method in the corresponding module, with the same name, where the number of parameters are equal to the number of inputs in the page*". This constraint can be explained in OCL as: "`context Button inv: self.page.flow.microModule.methods -> exist (m | m.name = self.name and m.inputparameters -> size() = self.page.input -> size())`".

Existence of appropriate attributes to reveal the results: after the called method finishes execution, the control should return to the flow, and the result of the action would be displayed through the output attributes of the called page. These attributes get their values from the fields of the corresponding module with the same names. This behavior results in the addition of another rule: "*For every output in a page, there should exist a variable in the corresponding module with the same name*". This rule is formalized in OCL as: "`context Output inv: self.page.flow.microModule.field -> (exist f | f.name = self.name)`".

C. Implementation rules

There are implementation level rules such as naming convention rules and coding policies. To illustrate this, we consider a rule that states: "The name of the input elements of a page should end with `input`". This rule can easily be expressed in OCL using String operations.

VI. Related Work

Reflexion model [9] is an early work on this topic and has been applied to some recent empirical studies on architecture conformance checking [13] [14]. In this method, the generated model of the source code is checked against the architectural model according to a mapping which is defined by the architect. Some approaches have refined this model [15] [16], for instance, by introducing a semi-automatic way to derive mappings between the source model and the architectural model through clustering. Reflexion model tool [17] uses an extractor tool to obtain the source model. Deriving such an extractor for the systems that are developed using various languages and may be specific to that system, seems not to be straightforward; but once we have it, the approach can be used to validate architectural rules.

Feijs et al. extended relational algebra to Relation Partition Algebra (RPA) [18] and used it to formulate architectural rules [19]. The *Part-of* relations model the hierarchical structure of a system and *call* relations express the dependencies between functions. A lifting operator is defined to find out the dependencies between software elements at different levels of software hierarchy using predefined relations. Eichberg et al. [20] propose an approach based on RPA to define and continuously check dependencies. Source elements such as Class, Method, etc. are recognized and represented by relations. A logic programming language is used to define the

groups of source elements called *ensembles* and the constraints on dependencies between them. These two approaches use relation based representations of the source program. If these relations can be defined for a multi-language system, the dependencies will be extracted accordingly. Defining these relations without the help of an automatic tool is a complicated task.

Structural Constraint Language (SCL) [21], based on first order logic, is a language for describing a wide variety of design intents. A graph model of the source program is traversed according to quantifiers of SCL expressions to obtain sets of entities and check constraints against them. So far, it only supports Java and C++ as the programming language. Kellens [22] used a logic-based language to describe executable queries on the source code to form logical groups of source elements called *Intentional Views*. To evaluate constraints, sets resulting from executing multiple queries describing a same view are matched against each other.

Aspect-Oriented Programming can use static pointcut that allows declaration of compile-time warning or error messages which can be used to check architectural rules like structural dependencies, naming conventions and pattern usage [7]. Checking for structural dependencies is conceivable through adding aspect code that specifies unintended dependencies in point cuts and will introduce a compile-time warning or error message when finding join points that match them.

Dependency Structure Matrix [23], [24] has also been used to visualize the hierarchical structure and inter-module dependencies in a scalable way to help architect recognize unintended dependencies.

ArchJava [25] is an approach that statically enforces architecture by introducing new types to Java language to denote architectural elements like component. This way component and connector view of architecture can be imposed.

In another point of view, our approach is related to works that investigate an aspect of a software system that is defined through different languages. The approach proposed in [26], supports architecture described in different languages. This approach is different from ours in the way that we suppose a unified representation of architecture is provided through the system meta-model, but the code can be developed in heterogeneous languages and needs to be abstracted in a unified way.

Moise et al. [27] propose an approach to discover cross-language dependencies. This work can be considered as a method to check the dependencies between the basic elements in different programming languages (e.g. C and Java); but it only supports dependencies and does not provide the facility for description of abstract elements (e.g. layer). So, it should be extended to be able to extract other necessary information to support various kinds of architectural rules.

There is a similar approach to our method in the Model Driven Software Development tool, openArchitectureWare (oAW) [28]. Although our method follows a model-based approach, it differs from oAW. oAW is mainly used for code generation while our method checks compliance of the source code with the architecture.

VII. Conclusion

We proposed a uniform model-based approach to check the compliance of source code with the architecture by examining the cross-language conformance rules. Our proposed method is not bound to a specific language, rather, it can be extended to adopt any language that is used in developing software.

We have implemented our method in Java and ANTLR, but the proposed method is more general and is not restricted to the specific tools used. The approach focuses on structural constraints imposed by the architecture. Dependencies between elements are the main subject of structural constraints. These dependencies between elements in the multi-language software are of two kinds: inter-dependencies (cross-language dependencies) and intra-dependencies. Inter-dependencies are dependencies between elements of different languages and intra-dependencies refer to dependencies between elements of a single language. Our approach supports intra-dependencies that can be specified by the artifacts of the corresponding language, while inter-dependencies are supported through integration of the language models.

Our method is intended to be independent of the language used in developing software in order to support cross-language rules. We obtain independence of domain specific languages in a uniform straightforward way but independence of general-purpose languages is complicated because of their large-scale abstract syntax. Once we define the abstract syntax grammar and the model generator of a language in the way described in Section IV, they can be reused in any other system that uses that language. This way, a repository of languages can be formed. So in the infrastructure building phase, we can reuse available language definitions.

We proposed a generic approach to be applied on a multi-language source code; hence we did not focus on providing high-level primitives on language structures. An example of such a primitive is a function that investigates source code and extracts control dependencies between the expressions in the source code. Our method can be extended with libraries, each of which provides such facilities for a different language.

In our approach, rules are expressed in OCL, which is a standard, declarative language. It provides the possibility of reusing rules at design and implementation levels. Architecture level rules that include abstract elements (e.g. layer) are dependent on the meta-model and can only be reused along with it. This way, we can also have a repository of rules that reduces the cost of defining conformance rules for a project. This is particularly important in the context of software product lines in which multiple products share a common architecture.

Acknowledgment

The authors would like to thank Mohammad Javad Izadi and Mr. Aguile for their helpful comments.

REFERENCES

[1] Spring web flow reference guide, ver. 2.0.8. [Online]. Available: http://www.springsource.org/webflow

[2] M. Fowler. Language workbenches: The killer-app for domain specific languages? [Online]. Available: http://www.martinfowler.com/articles/languageWorkbench.html

[3] Object management group, the model driven architecture resources page (2003). [Online]. Available: http://www.omg.org/mda

[4] F. Martin, "Who needs an architect?" *IEEE Software*, vol. 20, no. 5, pp. 11–13, 2003.

[5] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives.* Addison-Wesley Professional, 2005.

[6] N. Rozanski and Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives.* Addison-Wesley Professional, 2005.

[7] P. Merson, "Using aspect-oriented programming to enforce architecture," Software Engineering Institute, Tech. Rep. CMU/SEI-2007-TN-019, September 2007.

[8] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," *Software Architecture, Working IEEE/IFIP Conference on*, vol. 0, p. 12, 2007.

[9] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, 2001.

[10] M. Völter and T. Stahl, *Model-Driven Software Development.* Wiley & Sons, May 2006.

[11] Antlr parser generator. [Online]. Available: http://www.antlr.org/

[12] Eclipse modeling-mdt. [Online]. Available: http://www.eclipse.org/modeling/mdt/

[13] J. Knodel, D. Muthig, U. Haury, and G. Meier, "Architecture compliance checking - experiences from successful technology transfer to industry," in *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 43–52.

[14] J. Rosik, A. Le Gear, J. Buckley, and M. Ali Babar, "An industrial case study of architecture conformance," in *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement.* New York, NY, USA: ACM, 2008, pp. 80–89.

[15] R. Koschke and D. Simon, "Hierarchical reflexion models," in *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering.* Washington, DC, USA: IEEE Computer Society, 2003, p. 36.

[16] A. Christl, R. Koschke, and M.-A. Storey, "Equipping the reflexion method with automated clustering," *Reverse Engineering, Working Conference on*, vol. 0, pp. 89–98, 2005.

[17] jrmtool eclipse plug-in. [Online]. Available: http://jrmtool.sourceforge.net/

[18] L. M. G. Feijs and R. C. van Ommering, "Relation partition algebra — mathematical aspects of uses and part-of relations," *Sci. Comput. Program.*, vol. 33, no. 2, pp. 163–212, 1999.

[19] R. C. van Ommering, R. L. Krikhaar, and L. M. G. Feijs, "Languages for formalizing, visualizing and verifying software architectures," *Comput. Lang.*, vol. 27, no. 1/3, pp. 3–18, 2001.

[20] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *ICSE '08: Proceedings of the 30th international conference on Software engineering.* New York, NY, USA: ACM, 2008, pp. 391–400.

[21] D. Hou and H. J. Hoover, "Using scl to specify and check design intent in source code," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 404–423, 2006.

[22] A. Kellens, "Maintaining causality between design regularities and source code," Ph.D. dissertation, Vrije Universiteit Brussel, 2007.

[23] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," *SIGPLAN Not.*, vol. 40, no. 10, pp. 167–176, 2005.

[24] S. Huynh, Y. Cai, Y. Song, and K. Sullivan, "Automatic modularity conformance checking," in *ICSE '08: Proceedings of the 30th international conference on Software engineering.* New York, NY, USA: ACM, 2008, pp. 411–420.

[25] J. Aldrich, "Using types to enforce architectural structure," in *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008).* Washington, DC, USA: IEEE Computer Society, 2008, pp. 211–220.

[26] M. Leclercq, A. E. Ozcan, V. Quema, and J.-B. Stefani, "Supporting heterogeneous architecture descriptions in an extensible toolset," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 209–219.

[27] D. L. Moise and K. Wong, "Extracting and representing cross-language dependencies in diverse software systems," *Reverse Engineering, Working Conference on*, vol. 0, pp. 209–218, 2005.

[28] "openarchitectureware." [Online]. Available: http://www.openarchitectureware.org/