XCore: Support for Developing Program Analysis Tools

Alexandru Ştefănică and Petru Florin Mihancea
LOOSE Research Group
Politehnica University of Timişoara, Romania
Email: salexandru.alex@gmail.com, petru.mihancea@cs.upt.ro

***Abstract*—Building program analysis tools is hard. A recurring development task is the implementation of the meta-model around which a tool is usually constructed. The XCORE prototype supports this task by generating the implementation of the meta-model. For this purpose, developers will add directly into the source code of the tool under construction some meta-information describing the desired meta-model. Our demo presents some internal details of XCORE and emphasizes the advantages of our tool by describing the construction of a basic analysis instrument.**

***Index Terms*—meta-model, program analysis tool**

I. INTRODUCTION

Creating tools is an important goal of the research communities that elaborate analysis means to support software development. Unfortunately, building highly flexible, reusable and integrable program analysis tools is hard.

Figure 1 shows the generic architecture of an analysis instrument. The *back-end* extracts the raw artifacts needed by the *analyses* implemented in the tool (e.g., a parser creating the ASTs of the source code). Usually, *analyses* require artifacts that are more coarse-grained than what is provided by *back-end* extractors. Thus, some *builders* are needed to form higher-level artifacts (e.g., visit ASTs and capture all classes as first-class entities). These latter artifacts form the *model* of the analyzed system and it is structured according to a *meta-model* defined and implemented by the developers of the tool.

This is a recurring task for tool builders. IPLASMA [1] uses the MEMORIA meta-model to compute metrics. RANDOOP [2] captures the methods from a system and models random sequences of invocations in order to create tests. WALA [3] also defines and implements a meta-model capturing the program structure (e.g., existing classes) but also represents more complex entities like flow-graphs in order to perform flow analyses.

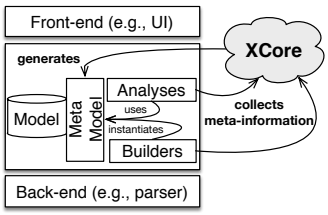

Fig. 1. XCORE Place in the Development of an Analysis Tool

One main concern when implementing a meta-model is to provide an implementation for its entities (e.g., writing the code of the meta-class whose instances will represent different types from the program under analysis). While this can be perceived as a simple task, even if it is time consuming from a development perspective, it is only half of the story. Like any other program, the analysis tool will evolve over time and include more and more sophisticated analyses requiring an enhancement to the initial meta-model. Consequently, dedicated mechanisms will have to be implemented from the beginning to allow easy meta-model extension. Similarly, other mechanisms could be required in connection with the meta-model to support easy reuse of simpler analyses in larger ones or to reuse analyses by inter-operating/integrating other tools. Unfortunately, all these additional mechanisms are difficult to design and implement, and different approaches have been proposed in the state-of-the-art literature to address them (e.g., [4], [5]). However, our tool offers an alternative.

XCORE (see Figure 1) provides developers with the means to add meta-information in the source code of the tool under development in order to describe the desired meta-model. Next, during the compile process of the tool, XCORE gathers all this meta-information and it automatically generates the implementation of the meta-model. Additionally, due to the way it was designed to be used, XCORE also facilitates other non-functional traits for the built tool such as the simplicity to extend its meta-model.

II. XCORE BASICS

To generate the implementation of a meta-model, XCORE needs a way to enable developers to describe it. Consequently, our tool needs a meta-meta-model.

A. The Meta-Meta-Model Used by XCORE

XCORE is based on the conceptual meta-meta-model described by Ganea et. al. in [5]. Figure 2a shows how this meta-meta-model has been employed into XCORE together with some extensions. To describe its components, we make use of the meta-model exemplified in Figure 2b.

XMethod and *XClass* are *types of entities* of the meta-model used to represent classes and methods from an object-oriented program. An entity has *properties* representing characteristics or analysis results associated to that entity (e.g., *noOfArgs* represents a metric for the method). An entity may also have

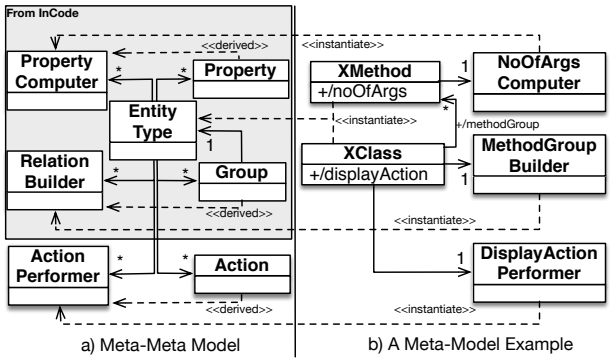

Fig. 2. The Conceptual Meta-Meta-Model and an Instantiation

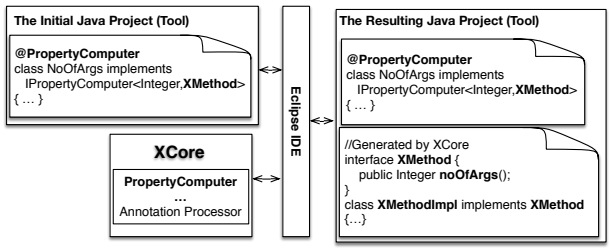

Fig. 3. Internals and Usage Overview

groups describing relations between entities (e.g., a class has a group of methods declared in that class). Last, but not least, an entity may have *actions* (e.g., *displayAction* opens the source code of the associated class in an editor). We emphasize that these properties, groups and actions are not explicitly defined. They are actually derived/obtained from their corresponding *PropertyComputer*, *RelationBuilder* or *ActionPerformer* objects. These last modelling elements are central from the XCORE internal and usage perspectives.

B. Internal Details and Usage Basics

XCORE is a tool for helping building tools and thus, it must work very closely with the developers. Consequently, it has been implemented as an ECLIPSE plugin (see Figure 3). Moreover, to generate the JAVA code for the meta-model of the tool under construction, XCORE is an annotation processor. Developers add classes to the tool project, and annotate them by specifying if they are a *property computer*, a *relation builder* or an *action performer*. When the project is built, ECLIPSE invokes XCORE asking it to process these annotations. As a result, XCORE injects the code for the found meta-model entities into the tool project (i.e., *XMethod* in Figure 3). This includes the required operations (i.e., *noOfArgs*) and their implementations that invoke the actual executants of those operations (i.e., the *NoOfArgs* property computer object).

III. BUILDING A TOOL WITH XCORE

To better understand the XCORE advantages we present in the following the main steps of developing a basic ECLIPSE plugin analysis tool using our meta-tool. This section assumes that XCORE is installed and that the ECLIPSE project of the built tool is properly configured[1]. The developed analysis instrument will model the classes and the methods from an object-oriented program as instances of *XClass* and *XMethod* meta-model entities. For each entity, the name will be provided. The tool will also compute the *NoOfArguments* metric for a method, and the *AvgMethodParameters* at the class level. Finally, the tool must facilitate the navigation from an *XClass* object to the source code of the modeled class.

A. Developing the DemoTool

Figure 4a shows the way we start describing the characteristics of an *XMethod*. We declare two classes annotated as *property computers*. They have to implement a predefined generic interface[2] and, based on the used type arguments, XCORE derives that the corresponding properties should be associated to an *XMethod*. However, ECLIPSE complains that there is no such type in the *DemoTool* project (as can be seen in left side of Figure 4).

When we will build the project, XCORE will generate the code of the *XMethod* type as can be seen in Figure 5a[3]. This includes the declaration of the corresponding operations in the *XMethod* interface (i.e., *toString()* and *noOfArguments()*) and their implementations (i.e., class *XMethodImpl*) which delegate a request to the corresponding property computer. Next, the way these properties are actually implemented must be provided by the developers.

XCORE does not depend on the usage of a particular back-end/fact extractor tool. However, it enables developers to associate a meta-model entity to an entity from the desired back-end. In this case (Figure 5b), an *XMethod* is associated to a *jdt.core.IMethod* type declared in the *Java Development Tool* component of ECLIPSE. This object is accessible via the *getUnderlyingObject* method present in the interface of all XCORE generated types. Consequently, the developers can implement the properties of an *XMethod* using the functionalities provided by the back-end (see Figure 5c).

In a similar fashion, we can declare and implement the characteristics of an *XClass*. The relation between a class and its group of declared methods can be represented using the *RelationBuilder* annotation (see Figure 6a). Based on the type arguments of the required generic interface, XCORE associates this group to an *XClass* and it will be clear that the group contains *XMethod* entities.

In order to construct such a group, two things are required. First, we need a way to determine all the methods declared in a particular class from the analyzed source code. For this purpose, we have to rely on the back-end and thus, an *XClass* is associated to the *jdt.core.IType* from *JDT*. Second, we need a way to instantiate *XMethod* objects from our meta-model. To

[1]Installation instructions and other tutorials (including a screencast and configuration details) are available at https://goo.gl/4BzB4Z

[2]While we could have relied only on class/method annotations, we also used interfaces to achieve structural uniformity in the meta-elements definitions

[3]Be aware that the folder in which Eclipse records the generated code is usually hidden

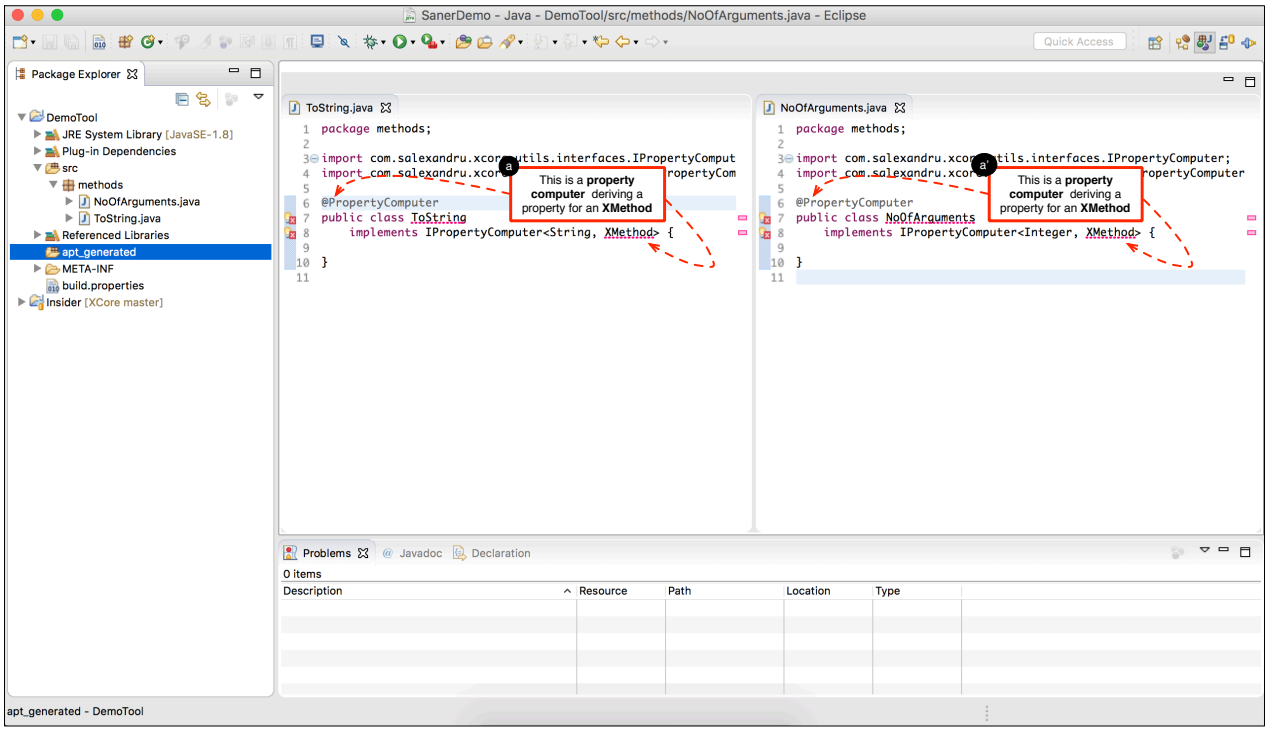

Fig. 4. Step 1 - Declaring The Characteristics Of An *XMethod*

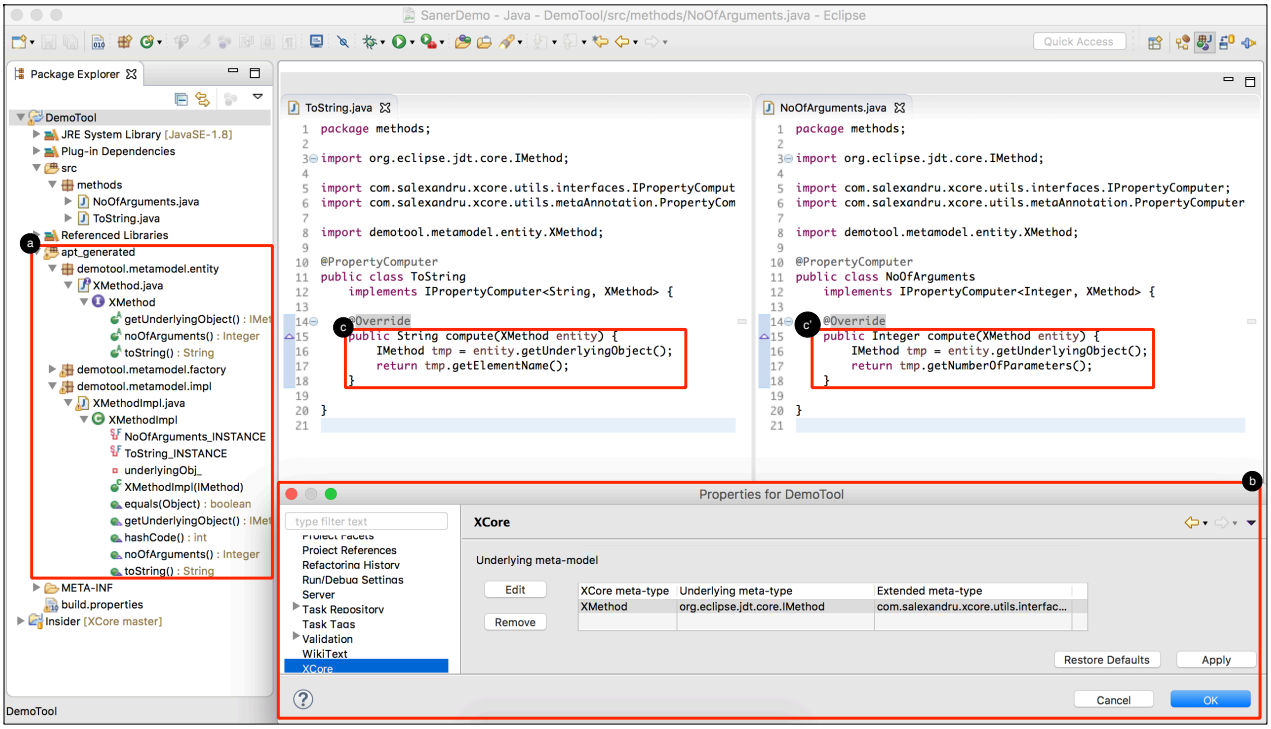

Fig. 5. Step 2 - Implementing The Characteristics Of An *XMethod*

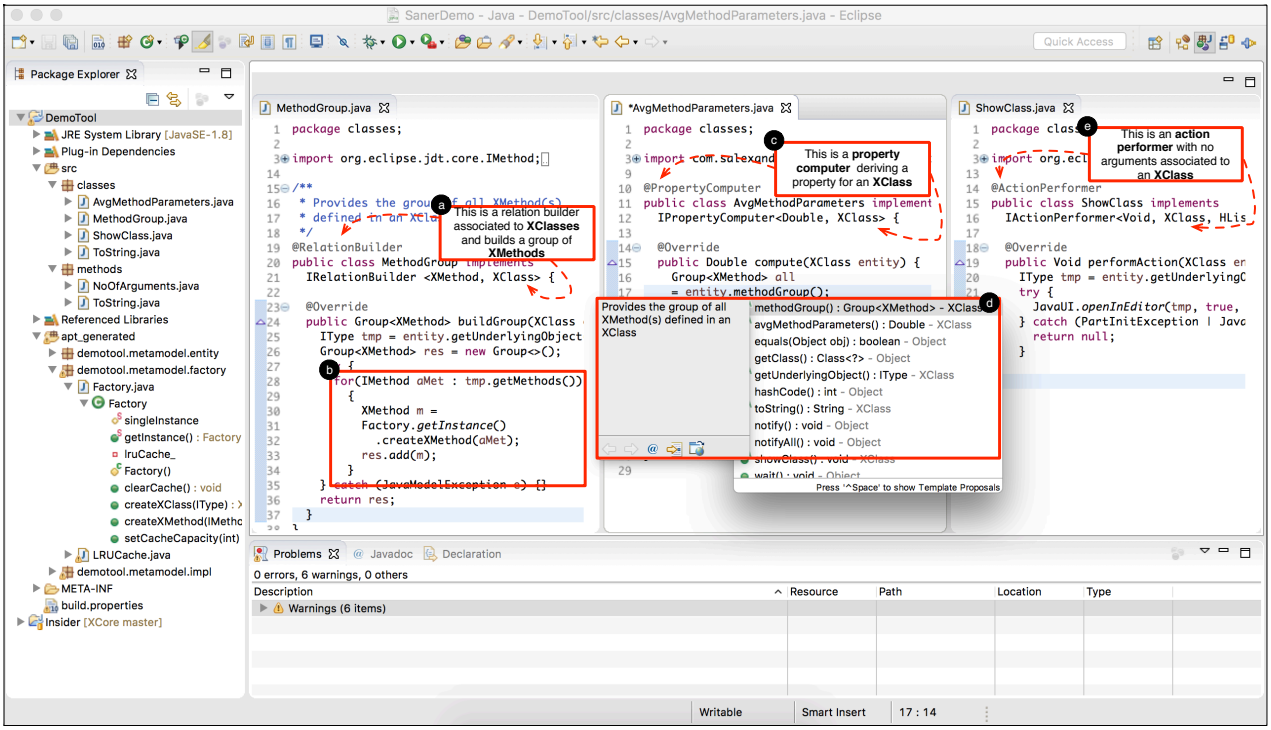

Fig. 6. Step 3 - Declaring and Implementing The Characteristics Of An *XClass*

support this task, XCORE also generates a *Factory* of objects that also enforces the association of a meta-model element to its underlying back-end object (e.g., an *XMethod* must be associated to a *jdt.core.IMethod* object as in Figure 6b).

Figure 6c presents the declaration of the *AvgMethodParameters* metric as a property computer associated to an *XClass*. The most important thing to note here is that, to compute a metric, one can reuse the previously defined metrics and groups. In our case, to implement *AvgMethodParameters* we can reuse the *NoOfArguments* metric and the *MethodGroup* of an *XClass*. As you can see in Figure 6d, through the *XClass* interface the metrics and groups are accessible together with their documentation.

Finally, Figure 6e exemplifies an *action performer*. In this case, the *ShowClass* action is associated to an *XClass* and it can be used to open the implementation of the modeled class into an ECLIPSE editor.

B. DemoTool *At Work*

Figure 7 shows the minimalist tool built with XCORE at runtime. INSIDER is a utility plugin we provide as a generic front-end for tools developed with XCORE. Via a pop-up menu, users can access the properties/groups/actions of any model object from registered tools.

IV. PROS, CONS AND OTHER FEATURES

In the previous section, we have shown how XCORE can be used during the development of an analysis tool to automatically generate the implementation of the tool meta-model. However, this is just one of the advantages offered by

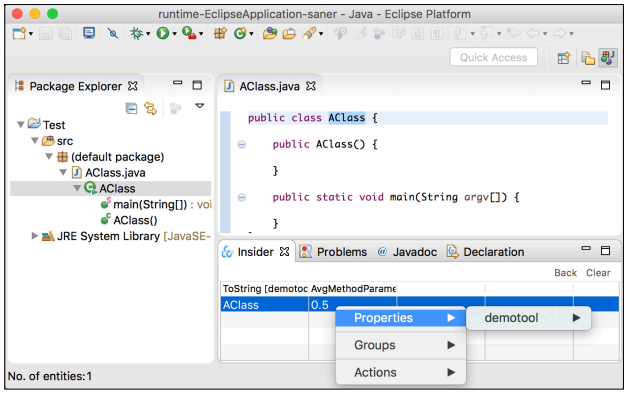

Fig. 7. Step 4 - DEMOTOOL at Runtime

XCORE. In addition, *extending* the meta-model of an XCORE-based tool is easy: programmers just add the annotated source code (e.g., a new relation builder class) into the ECLIPSE project and rebuild the tool. As a result, XCORE will detect the new meta-model element and it will properly integrate it in the regenerated meta-model implementation.

Another pro is the possibility of simple *integration* of an XCORE-based tool with another one built also with XCORE. By mixing them into the same ECLIPSE project, XCORE will generate the common meta-model that satisfies the expectations of each tool in part. Consequently, the combined tools will be able to interact programmatically. To simplify this merging activity we plan a new feature that is briefly described in Section VI.

Meanwhile, another XCORE feature goes in this direction: we can declare an analysis instrument (i.e., a sub-tool) as extending another tool (super-tool) built with XCORE. The effect is that the meta-model entities of the sub-tool will extend where possible the meta-model entities of the super-tool. In other words, the super-tool features will be "inherited" by the sub-tool where they can be (re)used. The drawback is that a copy of the super-tool must be packed together with the sub-tool into a single deployment unit. (i.e., the super-tool cannot be shared dynamically by different sub-tools because all of them must use the same meta-model implementation which is not possible when sub-tools are not aware of each other).

As a disadvantage, incremental compilation of an XCORE-based tool is difficult because our tool must know the *entire* description of the meta-model (i.e., all the relevant annotations) in order to generate entirely the required implementation. As another con, at this moment XCORE does not directly express inheritance between meta-entities in order to factor out common properties/relations/actions.

V. RELATED WORK

Our tool is related to INCODE [5] and IPLASMA [1] by the way they express and handle their meta-models. On one hand, XCORE uses in part the same meta-meta-model as the previous two analysis infrastructures. Even if their meta-models are easy to extend, these meta-models are not explicit. In other words, they exist only at runtime as instances of some classes representing the meta-meta-model. Consequently, the compiler cannot be aware of the static structure of the meta-model making the type checks of Java useless when acting on a meta-model entity (i.e., the compiler cannot know what operations can actually be performed on a particular meta-type). This proved to be problematic for complex meta-models because it is difficult for a developer to remember exactly all the characteristics of each entity from a meta-model.

By contrast, XCORE solves this issue because it generates the Java code of a meta-model and thus, makes it explicit. As a result, we can rely on the compiler to verify that an operation can be executed by the meta-model entity we invoke. Also, a developer can rely on ECLIPSE's code completion support to see the documentation of a meta-type operation at its invocation site.

Similarly, our tool is related to the MOOSE [6] analysis infrastructure and, more precisely, to the way its meta-model is manipulated. MOOSE uses the FAMIX meta-model described using the FM3 meta-meta-model and handled by the FAME tool [4] also available (at least partially) in Java. In short, JAVAFAME [7] generates the Java code of a meta-model taking as input the description of that meta-model from an external *mse* file. By contrast, XCORE lets developers describe the meta-model directly in the source code using Java annotations. One advantage of our approach is that the actual information is kept in a single place reducing the risk of de-synchronisation. Moreover, having the meta-model description in an external configuration file is problematic for refactoring engines. XCORE does not have this problem because the meta-model description is expressed in Java code constructs.

We also mention that Java annotations are also present in JAVAFAME. However, to the best of our understanding, they are not used to produce code. They are used as runtime annotations marking and categorizing the elements of a generated meta-model in order to build, for example, a generic browser to interact with the model [4]. It happens that we also use a similar mechanism to categorize the elements of the XCORE-generated meta-model (e.g., as a group, a property, or an action operation) in order to build the INSIDER generic UI. However, XCORE also uses the Java annotation processing mechanism to describe and trigger the meta-model code generation including the "linking" between the meta-model operations and their actual implementations.

VI. CONCLUSIONS AND FUTURE WORK

In this demo, we introduced XCORE, a meta-tool aimed to support the development of program analysis tools. We showed its feasibility and expected advantages by presenting the construction of a basic analysis instrument in an (almost) step by step fashion.

As future work, we plan to add the *Merge Analysis Tool* feature. In essence, this will help a developer to automatically copy the source code of an XCORE-based tool into the ECLIPSE project of another XCORE-based tool and to merge all the specific settings. As a result, compiling the destination tool will make XCORE generate the common meta-model of both tools enabling their interaction even at a programmatic level. In other words, the new feature will help to merge two distinct XCORE-based tools into a single one.

We also plan to support the integration of other tools that are not based on XCORE by offering the possibility to have multiple underlying objects at one time in an instance from an XCORE meta-model. Generating a caching mechanism for properties and groups is also considered for performance purposes. Finally, evaluating the generalisation power of XCORE with respect to different types of analyses is another TODO.

REFERENCES

[1] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel, "iPlasma: An integrated platform for quality assessment of object-oriented design," in *Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary*, 2005, pp. 77–80.

[2] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 815–816.

[3] "Wala - T. J. Watson libraries for analysis," http://wala.sourceforge.net/wiki/index.php/Main_Page, accessed: 2016-03-03.

[4] S. Ducasse, T. Gîrba, A. Kuhn, and L. Renggli, "Meta-environment and executable meta-language using Smalltalk: An experience report," *Journal of Software and Systems Modeling (SoSyM)*, vol. 8, no. 1, pp. 5–19, 2009.

[5] G. Ganea, I. Verebi, and R. Marinescu, "Continuous quality assessment with inCode," *Science of Computer Programming*, vol. 134, pp. 19–36, 2017.

[6] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of Moose: An agile reengineering environment," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 1–10, Sep. 2005.

[7] "FameJava," https://github.com/girba/FameJava, accessed: 2016-11-20.