fREX: fUML-based Reverse Engineering of Executable Behavior for Software Dynamic Analysis

Alexander Bergmayr
Business Informatics Group
TU Wien
Vienna, Austria
bergmayr@big.tuwien.ac.at

Hugo Bruneliere
Inria, Mines Nantes & LINA
Ecole des Mines de Nantes
Nantes, France
hugo.bruneliere@inria.fr

Jordi Cabot
ICREA
Open University of Catalonia
Barcelona, Spain
jordi.cabot@icrea.cat

Jokin García
Inria, Mines Nantes & LINA
Ecole des Mines de Nantes
Nantes, France
jokin.garcia-perez@inria.fr

Tanja Mayerhofer
Business Informatics Group
TU Wien
Vienna, Austria
mayerhofer@big.tuwien.ac.at

Manuel Wimmer
Business Informatics Group
TU Wien
Vienna, Austria
wimmer@big.tuwien.ac.at

ABSTRACT

Reverse engineering is still a challenging process, notably because of the growing number, heterogeneity, complexity, and size of software applications. While the analysis of their structural elements has been intensively investigated, there is much less work covering the reverse engineering of their behavioral aspects. To further stimulate research on this topic, we propose fREX as an open framework for reverse engineering of executable behaviors from existing software code bases. fREX currently provides model discovery support for behavior embedded in Java code, employs the OMG's fUML standard language as executable pivot format for dynamic analysis, and uses model transformations to bridge Java and fUML. Thus, fREX also aims at contributing to explore the relationship between programming languages (e.g., Java) and executable modeling languages (e.g., fUML). In this paper, we describe the proposed fREX framework and its current reverse engineering support covering some core Java features. In addition we discuss how the framework can be used for performing different kinds of dynamic analysis on existing software, as well as how it could be extended in the future.

Keywords

Reverse Engineering; Executable Behavior; Dynamic Analysis; Programming Language; Executable Modeling Language; fUML

1. INTRODUCTION

Reverse engineering is fundamental in several software engineering activities and contributes to facilitate software comprehension in many ways. Over the years, it has proven its relevance and usefulness both during and after software design, development, maintenance, evolution or modernization tasks. However, despite the already important achievements in this area, significant challenges remain to be tackled [11]. Properly reverse engineering the executable behavior of existing software is one of them. Indeed, while there has been a strong focus on static analysis techniques for structural aspects of software, there have been (much) less attempts to target their behavioral aspects by dynamic analysis techniques.

A practical illustration is the ARTIST initiative, which is an international collaborative project gathering research institutions and industrial partners coming from 7 different European countries [1]. The project has been funded by the European Commission during 3 years, and has resulted in both an overall methodology and the related tooling aimed at providing a global model-based reengineering approach for migrating existing software more easily to novel cloud offerings [5]. Notably, this involves selecting a cloud storage solution given a set of persistence requirements derived from software implemented in a variety of programming languages. This in turn requires at least (i) to obtain a precise data model and (ii) to understand how application data is persisted and retrieved. However, statically producing a representation allowing to reason on structural aspects is not enough. On the contrary, it is highly required to dynamically analyze the behavioral aspects of the system for deriving improvements concerning non-functional aspects. Dealing with such a scenario highlighted the practical need for a dynamic/behavioral reverse engineering support, as well as the effort required to realize it separately for several different programming languages (e.g., Java or C# that were both in the scope of the project). This would imply duplicating the work, e.g., to instrument source code and produce the runtime information in terms of machine-interpretable execution traces.

Among the different paradigms and underlying approaches available to tackle these problems, Model Driven Reverse Engineering (MDRE) is a promising one [9] that has been applied in the ARTIST context. MDRE encourages the application of model driven engineering (MDE) principles and techniques to generate relevant model-based representations of existing software applications. To do so, the use of a multi-viewpoint modeling language such as UML is quite often considered [28]. It enables expressing model-based views of the applications independently from the programming languages that are used to implement them. Studying such mappings between programming and modeling languages (e.g., Java and UML) has a long tradition in both reverse and forward engineering. The work in this area focuses mainly on UML's class diagram to capture structural aspects of an application, while behavioral aspects are typically expressed (mostly partially) in terms of

MiSE'16 May 16-17 2016, Austin, TX, USA

ISBN 978-1-4503-4164-6/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2896982.2896984

sequence and statechart diagrams. With the relatively recent emergence of the fUML standard [24], UML's activity diagrams appear to be more appropriate for capturing behavioral aspects in a way that they can be executed directly at model-level.

In this paper, we introduce fREX as a twofold contribution: *1)* an open extensible framework that is capable of automatically generating and executing fUML models from existing applications, and *2)* a base mapping between the UML's languages for activity/class diagrams (i.e., fUML) and the core language features of Java, putting the focus on behavioral aspects and their execution at model-level. To be able to obtain the required runtime information, fUML comes with a dedicated virtual machine (VM) that has been extended to provide execution traces as a runtime model [22]. We believe our proposed approach comes with the following interesting benefits:

- (i) genericity and reusability are fostered as developed analysis techniques can be potentially applied to arbitrary source code, assuming that a mapping from the employed language to fUML is provided;
- (ii) extensibility is made easier as new languages can be mapped at any time to the pivot (also making possible to combine several programs written with different languages into a same fUML model), as well as new analysis components provided on top of it;
- (iii) non-intrusiveness is ensured as instrumenting the source code and serializing the obtained runtime information is no longer required, execution traces being produced directly by the fUML VM.

In Section 2, we briefly introduce fUML and present the overall architecture of our fREX framework. Then in Section 3, we introduce our Java-to-fUML mapping by means of a concrete example. To provide more insights into fREX, we also show the runtime model produced by the fUML VM as a result of executing the obtained fUML model. In Section 4, we describe our current Eclipse-based implementation of the fREX framework. In Section 5, we list several possible application scenarios that can be realized on top of the fREX framework. Finally, we discuss the related work in Section 6 before we conclude with an outlook on next steps in Section 7.

2. THE fREX FRAMEWORK

A fundamental idea of the proposed fREX framework is the central use of a common representation format for all behavioral concerns. fUML [24], as a subset of UML focusing on executability aspects, plays the essential role of a pivot language in our solution. Thus, we briefly introduce fUML in Section 2.1 and then present the overall architecture of fREX in Section 2.2.

2.1 fUML in a Nutshell

Application behavior can be defined in UML either interaction-oriented by using sequence diagrams or state-oriented by using state machine diagrams. The behavior triggered by such interactions and states can be represented in details by means of activity diagrams. Recently, the formal semantic definition of two core sub-languages of UML, namely class and activity diagrams, has been considerably improved. fUML corresponds to this core subset of UML that has been identified as relevant for representing software behavior with the main purpose of executing it. In particular, the fUML standard [24] makes explicit the semantics of these two sub-languages for a dedicated virtual machine (fUML VM) that is able to interpret both class and activity diagrams [27]. Thus, similarly

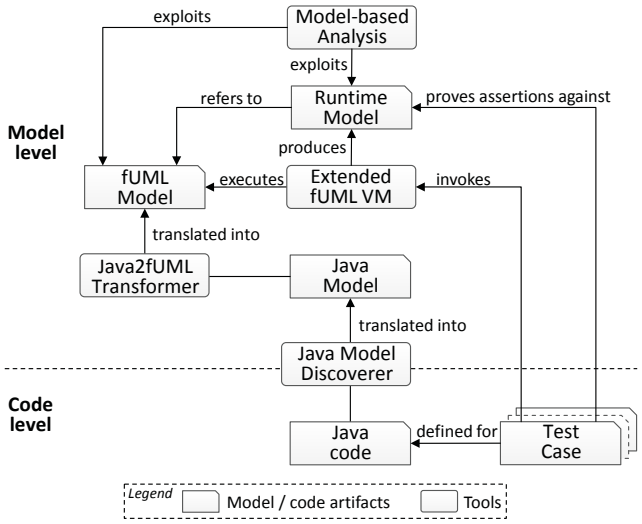

Figure 1: fREX Overall Architecture

to existing object-oriented programming languages (e.g., Java or C#), fUML provides concepts for defining classes with attributes and operations, abstract classes, multiple inheritance, enumerations as well as an extensible type system. Operation bodies are implemented by activities and via the action language provided by UML. It is a complete language which enables expressing manipulations and other computations. Hence, fUML appears to be a potential language for capturing the behavior of source code at model-level. As it is capable to represent the behavior in executable form, it enables dynamic analysis to be carried out directly at model-level instead of code-level. This is beneficial for realizing model-based analysis tools.

2.2 fREX Architecture

The fREX framework is intended to facilitate the construction of several structural and behavioral views on a given software, and this at different levels of abstraction depending on the reverse engineering needs. It is made easier notably thanks to the core use of models, which allows directly benefiting from related modeling techniques such as multi-viewpoint modeling and model transformation (e.g., for view computation, refinement, slicing, querying, execution, to name just a few). Thus fREX follows the typical two-phase process of many MDRE frameworks [9]:

- **Model Discovery** generates from the software artifacts and/or their executions the needed initial models representing the raw behavior of the considered software. In our present case, base fUML [24] models are automatically discovered from Java source code.
- **Model Understanding** further analyzes the previously obtained fUML models by producing derived traces and/or models proposing different additional relevant views. In our present case, we employ a fUML VM to execute these models and test the produced traces.

The overall architecture of fREX and its current Java support is presented in Figure 1.

Producing fUML-based representations from source code requires both overcoming different encodings and resolving language heterogeneities [7]. Thus, instead of directly translating plain code into fUML, a two-step approach is preferable for the fUML model

discovery phase. Firstly, the source code is *translated into* a code model (a Java model in the present case) using a low-level specific *model discoverer*. The obtained model conforms to a metamodel of the programming language that precisely describes its terminology and structures. Secondly, this code model is *translated into* a fUML model that resolves language heterogeneities by relying on the correspondences between the given language (here Java) and fUML metamodels. This is implemented as a so-called *transformer*.

Having obtained a proper fUML model, it can then be directly *executed* by the fUML VM in a model understanding phase. In previous work, we incorporated additional tracing support into an existing fUML VM [22]. In particular, we elaborated on a metamodel allowing to capture the runtime behavior of fUML models in terms of execution traces and extended the fUML VM for recording execution traces as instances of this metamodel. Hence, as a result of the model execution onto the fUML VM, a runtime model is *produced* capturing execution traces *referring* to the executed fUML model. They provide information on executed activities and their actions including information about their call hierarchy, and the chronological and logical order of their execution, as well as information on the runtime states of the model during the execution. The generated runtime model along with the previously discovered fUML model can then be *exploited* by model-based analysis techniques. These include model refinement, slicing or view generation for instance (see Section 5 for different possible application scenarios).

In addition, in order to check the validity of the produced fUML models, we apply a test-driven approach. The base idea is to define and run unit tests for asserting that the discovered fUML models actually capture the original behavior of the Java code. Actually, we compare for a given input the result of a given fUML model execution (i.e. a runtime model) against the result of running the corresponding piece of code. We apply this approach to continuously validate new language correspondences that are implemented by the available transformers (e.g. the Java-to-fUML one, cf. Section 3). Please note that the code-level test cases are for now manually translated into model-level test cases and that model-level test cases are—like the fUML VM—implemented with Java. However, an automated translation of test cases is in principle possible if all programming language constructs needed for defining test cases are supported by the model discoverer.

To conclude and recall from the introduction, the proposed architecture for the fREX framework (including notably the use of fUML as a pivot representation format) comes with several interesting benefits from a reverse engineering point of view. Firstly it allows *extensibility* from the model discovery perspective, as new model discovery components targeting fUML can be implemented from various kinds of software inputs. For instance, different fUML model discoverers could be built for supporting behavioral reverse engineering from both Java and C# source code. Secondly it also permits *genericity* and *reusability* from the model understanding and analysis perspective, as existing components consuming fUML models can be reused independently from the original nature of the treated software. This way, the same execution capabilities and/or analysis transformations can be used indifferently on all fUML models. Moreover, the fact of considering only (fUML) models for execution and analysis provides an interesting *non-intrusiveness* property to the framework. Hence no modifications (e.g., for code instrumentation purposes) are required at source-level anymore, as everything can be performed at model-level (e.g., via utilizing the used fUML VM).

3. THE JAVA-TO-fUML EXAMPLE

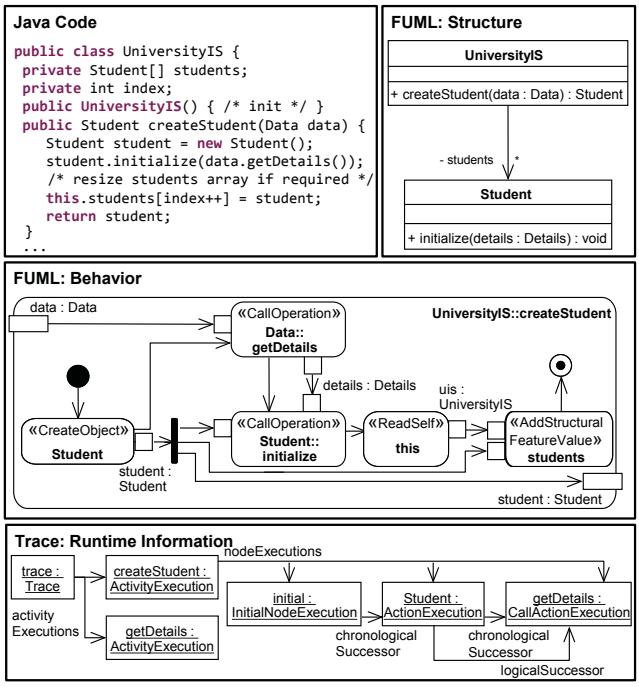

Figure 2: Java code expressed and executed by means of fUML

To demonstrate the capabilities of the fREX framework for an extensively used programming language, we decided to start working on the Java case. As an example, Figure 2 gives an overview of the different artifacts and models considered and produced by fREX from a given piece of Java code. Notably, some Java code and a corresponding (reverse engineered) fUML model are depicted there. Application structure and behavior are captured by a class diagram and activity diagram, respectively. An excerpt of the traces resulting from the execution of the illustrated activity (i.e. the runtime model) is shown beneath the diagrams. Due to the high complexity of a complete mapping between Java and fUML, we started by addressing a subset of Java called MiniJava[1]. Thus, we decided to voluntarily delay the treatment of some other aspects of the language (cf. Section 7). Our current Java-to-fUML mapping is inspired from initial work within the standard fUML specification [24] which we refined, extended, and implemented in terms of a Java-to-fUML model transformation (cf. Section 4).

Table 1 introduces the conceptual mapping required to discover an fUML model form the Java code of Figure 2. It shows the rules for translating the statements of the `createStudent` method into corresponding fUML model elements. The concepts on the left hand side of the table refer to the terminology of the Java Language Specification (JLS) [25], whereas in the right hand side are corresponding concepts defined by the fUML metamodel [24]. In this present work, the focus is set on behavioral aspects by capitalizing on the structural mapping realized in JUMP [6] and by complementing it with new behavioral elements.

From a structural perspective, a method declared in Java corresponds to an operation in UML. In order to capture its behavioral elements at model-level, it is also mapped to an activity that is linked to the operation (see `specification` property). The name of the activity is derived from the method signature. Furthermore, formal parameters and the return type defined by the method

[1]http://www.cambridge.org/us/features/052182060X/

signature are mapped to parameter nodes of the corresponding activity. As an activity explicitly defines control nodes at which the execution starts and ends when it is invoked, those nodes, i.e., `InitialNode` and `FinalNode`, are created by default for each activity. If a `FinalNode` has been executed, the activity execution terminates. The activity also terminates if no activity node is enabled anymore. After the termination, the activity execution collects the object tokens residing on output activity parameter nodes and provides them as output (see the `student` object).

A created instance variable (see the `student` instance) is mapped to an fUML action that creates an object (i.e., `CreateObjectAction`). The action's name and classifier are derived from the type (Java class) that is instantiated. Additionally, an output pin is created at which the action puts the instantiated object at runtime. The instantiated object is distributed to possibly several other actions via a fork node. It is connected to the action's output pin via an object flow edge. The latter ensures that the objects are offered to the successor activity nodes once the current node has been executed.

A method invocation is mapped to an fUML action for calling operations (i.e., `CallOperationAction`). Its main properties (i.e., `name` and `operation`) are derived from the signature of the method that is invoked. Input pins and the respective object flow edges are created for the target object of the invocation[2] and for the values passed to the parameters of the invoked method. Also, an output pin is created if the invoked method returns a value.

A value assignment to a multi-valued Java variable (e.g., an array of students) is mapped to a named fUML action that adds a value to a structural feature[3] (i.e., `AddStructuralFeatureAction`). The latter is referenced accordingly by the action (see its `structuralFeature` property). Again, input pins and the respective object flow edges are created for the left hand side as well as the right hand side of the assignment statement.

Finally, a `ReadSelfAction` along with an output pin are created when Java's "this" keyword is used to refer to the member of the current object from within an instance method[4].

Java Concept	fUML Concept
MethodDeclaration md	**add** Activity a
	a.name = md.name
	a.specification = -- *infer respective Operation from structural part*
ReturnType rt	**add** ActivityParameterNode rapn
	rapn.name = "return"
	rapn.type = rt.type
	rapn.parameter = -- *infer respective Paramter from structural part*
FormalParamter fp	**add** ActivityParameterNode fapn
	fapn.name = fp.name
	fapn.type = fp.type
	fapn.paramter = -- *infer respective Paramter from structural part*
Block b	**add** InitialNode in, FinalNode fn, StructuredActivityNode san
	-- *infer control flow from b.statements*
VariableDeclaration vd, ClassInstanceCreation cic	**add** CreateObjectAction createOA
	createOA.name = vd.type.name
	createOA.classifier = vd.type
	add OutputPin op, ObjectFlow of, ForkNode fn
	of.source = op, of.target = fn
MethodInvocation mi	**add** CallOperationAction callOA
	callOA.name = mi.method.name
	callOA.operation = mi.method
	add InputPin ip, ObjectFlow of -- *for target object, e.g., student*
	add InputPin ip, ObjectFlow of **foreach**
	FormalParameter fp **in** mi.method
	-- *infer source and target of ObjectFlows*
	add OutputPin op **for** ReturnType rt **in** mi.method
Assignment a **switch**(a.leftHandSide) **case:** ArrayAccess	**add** AddStructuralFeatureValueAction asfva
	asfva.name = -- *infer name from left hand side*
	asfva.structuralFeature = -- *infer feature from left hand side*
	add InputPin ip, ObjectFlow of **fo**r a.leftHandSide
	add InputPin ip, ObjectFlow of **fo**r a.rightHandSide
	-- *infer source and target of ObjectFlows*
ThisExpression	**add** ReadSelfAction rsa
	rsa.name = "this"
	add OutputPin op

Table 1: Mapping between MiniJava and fUML

4. fREX TOOLING IN ECLIPSE

The current implementation of the fREX framework relies on the combined use and integration of several components (i.e., plugins) from the Eclipse Modeling Project. For interoperability purposes, all models created and handled by fREX are based on Eclipse Modeling Framework (EMF)[5]. Moreover, the UML2[6] EMF-based reference implementation is used for representing the discovered (f)UML models.

The initial low-level Java model discovery step (from a source Java project) is automatically performed by reusing the corresponding MoDisco[7] component. Then, the previously introduced Java-to-fUML mapping is implemented as the current version of our transformer by using the ATL[8] model-to-model transformation language and tooling. As mentioned before, the extended fUML VM developed in the Moliz[9] project is utilized to provide the required model execution capabilities. Finally, JUnit test cases have been implemented to ensure that the produced fUML models behave (i.e. execute) as expected.

We checked the completeness and correctness of our implementation, concerning both the model discovery/mapping and the model understanding/execution steps, with the following practical testing methodology: (i) develop Java examples that use the aforementioned Java structures, (ii) discover Java models from them, (iii) transform these Java models into fUML ones, (iv) execute these fUML models and the original Java code via unit tests, and (v) compare the outputs produced by executing the fUML models and the original Java code.

In addition to these core aspects, a couple of UI plugins providing fREX-specific contextual actions have also been implemented. They offer to users simple ways of launching the different steps of the reverse engineering process from the Eclipse workbench they are familiar with.

The source code of the fREX implementation as well as a corresponding demo/video (applying our testing methodology on a concrete example), is available at our project Web page [16].

5. APPLICATION SCENARIOS

Having fUML models that represent behavioral aspects of existing software, the way is paved for further software understanding and analysis carried out directly at model-level. We describe in this section some concrete scenarios practically (re)using these fUML models. In order to give an initial impression of the applicability of our fREX framework, we consider hereafter three main families of model-based analysis techniques.

[2]Here, the assumption is that an instance (i.e., non-static) method is invoked.

[3]The upper value of its multiplicity is assumed to be unbounded (i.e., 0..*).

[4]The "this" keyword may not only be used in the context of a method declaration but also a constructor declaration.

[5]https://eclipse.org/modeling/emf

[6]https://eclipse.org/modeling/mdt/?project=uml2

[7]https://eclipse.org/MoDisco

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

[9]http://www.modelexecution.org

5.1 Model Refinement

A first way of dealing with the obtained fUML models is to refine them using one or several model transformations. One of the objectives may be to insert additional information into the fUML models, possibly computed and/or coming from other models. Thus, in the proposed framework we are able to complement the initially generated fUML models by using runtime information coming from the trace models produced by the fUML VM (cf. the one from our example in Figure 2). Other interesting refinements could be achieved too at fUML-level. For example, transformations could be proposed in order to explore automatically, based on model executions, the refinement of associations into bi-directional associations or compositions with more accurate multiplicity constraints in the fUML models. This requires an analysis of the execution traces to observe if changes on one of the two potential unidirectional associations are always replicated on the other, suggesting that they are indeed representing the same concept. We have already implemented a first version of such a fUML model-to-model transformation for exploration purposes.

5.2 Model Slicing

The obtained models convey many types of information that are more explicit than in source code, e.g., associations between classes as discussed before. However they may not scale well in some cases, notably when the volume of represented information becomes too large. Thus, model-based slicing techniques [3] can help in capturing only relevant parts of a larger model for a given purpose. The class diagram depicted in our example can already be considered as a slice because it shows a reduced part of the whole university information system. With the dynamic approach in our framework, slices can be produced that contain only model elements required for a specific execution, e.g., creating a student entity, facilitating the comprehension of the parts of the model behavior relevant to specific functionalities. Complementary to this, model slicing could be extended by chaining different transformations computing distinct slices. For instance, in a first step, models capturing behavioral aspects are sliced according to a given slicing criterion. Then, in a second step, the structure influenced by the execution of the sliced behavior may be obtained. The latter can be achieved by computing a model slice according to the type information of the produced objects. Additionally, these slices may be propagated again to other UML viewpoints such as architectural ones (e.g., in UML component diagrams).

5.3 View Generation

Generating different useful views on existing software is one of the major purposes in reverse engineering [11]. A view enables turning the focus on certain concerns where a pertinent viewpoint specifies the conventions for representing such a view. As our approach relies on fUML and as its parent (i.e., UML) is a multi-viewpoint language, several interesting views are naturally conceivable for our example. For instance, in order to represent high-level interactions relevant in the context of creating a student in the university information system, a dedicated view based on UML sequence diagrams may be produced using trace analysis techniques, e.g., cf. [8] by converting the fUML VM produced traces to UML sequence diagrams. Deriving partial (and usually more abstract) representations of the software behavior would allow the right amount of information to be conveyed to each stakeholder involved in the system. Finally, we also foresee the potential application of domain-specific languages for behavioral analysis, highlighting aspects which are not straightforward to represent in pure UML models. To this intent, more generic (in the sense of metamodel-independent) model view approaches that allow relating together models which conform to different metamodels could be reused, e.g., cf. [10].

6. RELATED WORK

In this section, we consider three main lines of research related to our ongoing work on fREX. At first, we discuss existing approaches for representing software behavior in terms of models. As we propose an automated approach for discovering fUML models from application code, we then compare fREX to existing reverse engineering approaches with a particular emphasis on systematic mappings between Java and UML (from both a reverse and forward engineering perspective). Finally, we discuss how fREX differs from existing approaches supporting dynamic analysis.

6.1 Modeling Software Behavior

As mentioned in the introduction, there are already significant results as far as modeling structural aspects of software is concerned. However, there has been less initiatives really focusing on modeling precisely software behaviors. In addition to fUML, which we have already deeply discussed in this paper, Micro-KDM [26] is also capable of representing application behavior in a language-independent way at model-level. However, there is currently no explicit semantic specification and execution engine for Micro-KDM. Another possibility would be to extend other languages used for measurement and metric calculation such as the FAMIX language from Moose [13] or M^3 [4] developed within the OSSMETER project[10] with an action language such as the one already provided by fUML. In all mentioned cases, more reverse engineering support is still required in order to automatically obtain relevant and valid behavioral models from already existing source code.

6.2 Model-based Reverse Engineering

Generally, the elaboration of mappings between programming and modeling languages such as Java and UML is not new in software engineering [6, 14, 18, 19, 23, 28]. For instance, round-trip engineering for UML and Java has been extensively studied in the context of the development of FUJABA [23]. However, only a few approaches [15, 17, 29] have been considering UML activity diagrams for the purpose of expressing application behavior at model-level. These approaches focus on forward engineering as they use Java as the output language and their mapping (from UML) is encoded by code generators. The base of such mappings may also be reusable in a reverse engineering context (such as ours), but they would have to be complemented to express concepts such as `ControlFlow` and `ObjectFlow` that are not explicitly represented in the application code. The difference between existing approaches that deal with reverse engineering of activity diagrams from application code [20] and our approach is that their proposed tooling is strongly language and visualization-oriented, while we follow a more generic approach targeting model execution. In our case, models are discovered solely by static analysis, whereas in the work of Martinez *et al.* [21] rather dynamic analysis techniques are employed. They produce an activity diagram capturing a certain execution path, while we obtain a representation of the overall behavior independently from any execution scenario. Carrying out execution at model-level to provide dynamic analysis, we thus aim for more completeness during the whole reverse engineering and analysis process.

6.3 Model-based Dynamic Analysis

[10] http://www.ossmeter.org

Finally, from a software analysis perspective, there is already a significant body of work [12] that covers different techniques and tools [11]. Existing approaches that support dynamic analysis typically gather runtime information directly at code-level, based on which the analysis is then carried out. Many of these approaches use UML(-like) representations to capture actual analysis results in terms of models. For instance, the UML sequence diagram is often used in the context of execution trace analysis. Our approach is different as we aim at performing the full dynamic analysis at model-level, in particular on top of previously discovered fUML models. Such models are more expressive compared to program code in several respects (e.g., different kinds of relationships, precise multiplicities, explicit control flow and data flow) which is beneficial for realizing more powerful dynamic analysis tools. Clearly, this additional information could be inferred from program code as the latter is also the basis for our reverse engineering step. However, once the model-level has been reached, analysis tools working at this level can directly benefit from these richer model-based representations as well as from the large ecosystem of model-based techniques and tools.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented an MDRE approach enabling the representation and dynamic analysis of existing software behaviors. The proposed framework mainly relies on the use of fUML as a pivot language for representing application behavior and on its associated VM for executability purposes. The overall idea is to fully perform the behavioral analysis at model-level, thus benefiting from interesting characteristics in terms of genericity, reusability, extensibility, and non-intrusiveness. Generally, we believe our approach is particularly useful when (i) analyzing dynamic aspects of existing software is required in a given reverse engineering process and (ii) input software employs a variety of languages/platforms, which makes it unfeasible (or too costly) to get specific analysis techniques for each of them. In this context, available models and/or model-based techniques can be reused extensively. The first obtained results are promising but several open challenges remain, e.g., as far as the scenarios proposed in Section 5 are concerned. Obviously, the scope can also be extended to other practical applications such as workload extraction in large-scale database systems, for instance. We plan to tackle these challenges progressively in the next steps of our work.

Notably, the mapping from Java to fUML revealed interesting findings from an fUML perspective. Several aspects of the Java language are currently challenging to be represented by fUML models such as dynamic dispatching, generics (for classes and interfaces), exceptions and assertions, external libraries, Java Native Interface (JNI) and corresponding reflection aspects. These concepts or equivalent ones are currently not directly supported by fUML. Thus, in the future, we plan to explore how fUML may be extended to provide a more complete set of concepts which can in turn be used to map more programming language concepts directly to fUML. In particular, we plan to investigate on how the mapping between annotations at code-level and model-level can be extended in order to incorporate also behavioral aspects. Our current idea is to exploit automatically discovered UML profiles providing corresponding annotation stereotypes [6], which include also behavioral aspects in a form that they are directly usable by the fUML VM for execution purposes.

Furthermore, the current version of the fREX framework only comes with single language support so far, i.e., for Java via our proposed Java-to-fUML mapping. On one hand, this mapping and implementing model transformation still need to be improved to support more and more (behavioral) aspects of the complete Java language (and not restricted to MiniJava). On the other hand, for validation purposes, it would be very interesting to enlarge the scope of the framework by covering another widely used object-oriented language, such as C# or C++ for instance. Eventually, for the sake of completeness, the study may also be extended to a few non object-oriented programming languages whenever relevant and possible. Thus, as mentioned in introduction of this paper, the capability to reverse engineer multiple programs written with different languages into a single fUML model (at a same abstraction level) is a relevant aspect to study deeper in the future. Moreover, a comparison with OMG's Knowledge Discovery Metamodel (KDM) [2] may be conducted to explore the pros and cons of using either UML or KDM to represent existing software at model-level.

Acknowledgment

The presented work has been funded by the European Commission via grant no. 317859 (ARTIST project) and by the Christian Doppler Forschungsgesellschaft and the BMWFW, Austria

8. REFERENCES

[1] ARTIST EU project - Advanced software-based seRvice provisioning and migraTIon of legacy SofTware, 2016. http://www.artist-project.eu.

[2] OMG's Architecture Driven Modernization (ADM) Knowledge Discovery Metamodel (KDM), 2016.

[3] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt. State-based Model Slicing: A Survey. *ACM Comput. Surv.*, 45(4):53:1–53:36, 2013.

[4] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju. M^3: A general model for code analytics in rascal. In *Proc. of SWAN*, pages 25–28, 2015.

[5] A. Bergmayr, H. Bruneliere, J. Cánovas, J. Gorroñogoitia, G. Kousiouris, D. Kyriazis, P. Langer, A. Menychtas, L. Orue-Echevarria, C. Pezuela, and M. Wimmer. Migrating Legacy Software to the Cloud with ARTIST. In *Proc. of CSMR*, pages 465–468, 2013.

[6] A. Bergmayr, M. Grossniklaus, M. Wimmer, and G. Kappel. JUMP - From Java Annotations to UML Profiles. In *Proc. of MODELS*, pages 552–568, 2014.

[7] A. Bergmayr and M. Wimmer. Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques. In *Proc. of MDEBE*, pages 22–31, 2013.

[8] L. C. Briand, Y. Labiche, and J. Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Trans. Software Eng.*, 32(9):642–663, 2006.

[9] H. Bruneliere, J. Cabot, G. Dupe, and F. Madiot. MoDisco: a Model Driven Reverse Engineering Framework. *Information & Software Technology*, 56(8):1012–1032, 2014.

[10] H. Bruneliere, J. G. Perez, M. Wimmer, and J. Cabot. EMF Views: A View Mechanism for Integrating Heterogeneous Models. In *Proc. of ER*, pages 317–325, 2015.

[11] G. Canfora, M. Di Penta, and L. Cerulo. Achievements and Challenges in Software Reverse Engineering. *Commun. ACM*, 54(4):142–151, 2011.

[12] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Trans. Software Eng.*, 35(5):684–702, 2009.

[13] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal? UML shortcomings for coping with round-trip engineering. In *Proc. of UML'99*, pages 630–644, 1999.

[14] G. Engels, R. Hücking, S. Sauer, and A. Wagner. UML Collaboration Diagrams and their Transformation to Java. In *Proc. of UML*, pages 473–488, 1999.

[15] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Proc. of TAGT*, pages 296–309, 2000.

[16] fREX. fREX Project web page, 2016. https://github.com/atlanmod/fREX.

[17] D. Gessenharter and M. Rauscher. Code Generation for UML 2 Activity Diagrams: Towards a Comprehensive Model-driven Development Approach. In *Proc. of ECMFA*, pages 205–220, 2011.

[18] W. Harrison, C. Barton, and M. Raghavachari. Mapping UML Designs to Java. In *Proc. of OOPSLA*, pages 178–187, 2000.

[19] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Proc. of WCRE*, pages 22–32, 2002.

[20] E. Korshunova, M. Petkovic, M. van den Brand, and M. Mousavi. CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code. In *Proc. of WCRE'06*, pages 297–298, 2006.

[21] L. Martinez, C. Pereira, and L. Favre. Reverse Engineering Activity Diagrams from Object Oriented Code: An MDA-Based Approach. *Computer Technology & Application*, 2(11):969–978, 2011.

[22] T. Mayerhofer, P. Langer, and G. Kappel. A Runtime Model for fUML. In *Proc. of MRT*, pages 53–58, 2012.

[23] U. Nickel, J. Niere, and A. Zündorf. The FUJABA Environment. In *Proc. of ICSE*, pages 742–745, 2000.

[24] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.1, August 2013. http://www.omg.org/spec/FUML/1.1.

[25] Oracle. Java Language Specification 8, March 2015. http://docs.oracle.com/javase/specs.

[26] R. Pérez-Castillo, I. G.-R. De Guzman, and M. Piattini. Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532, 2011.

[27] B. Selic. The Less Well Known UML: A Short User Guide. In *Proc. of SFM*, pages 1–20, 2012.

[28] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code (Monographs in Computer Science)*. Springer, 2004.

[29] M. Usman and A. Nadeem. Automatic generation of Java code from UML diagrams using UJECTOR. *Software Engineering & Its Applications*, 3(2):21–37, 2009.