TRACEM - Towards a Standard Metamodel for Execution Traces in Model-Driven Reverse Engineering

Claudia Pereira[1], Liliana Martinez[1], Liliana Favre[1,2]

[1] Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina
[2] Comisión de Investigaciones Científicas de la Provincia de Buenos Aires, Argentina
{cpereira, lmartine, lfavre}@exa.unicen.edu.ar

Abstract. Reverse engineering is a crucial stage in the software modernization process. The current techniques available in existing CASE tools provide forward engineering and limited facilities for reverse engineering, dynamic analysis in particular. The Architecture-Driven Modernization initiative has defined standards to support the modernization process in the model-driven engineering (MDE) context. Standardization increases interoperability between different tools enabling a new generation of solutions to benefit the whole industry and encourage collaboration among complementary vendors. In this paper, we present TRACEM, a metamodel to represent trace information under a standard representation. This metamodel complements a MDE framework for software modernization that aims to integrate static and dynamic analysis techniques during the reverse engineering process. This paper includes a case study that exemplifies how dynamic information combined with static information allows improving the whole reverse engineering process.

Keywords: Architecture-Driven Modernization, Metamodeling, Transformation, Static analysis, Dynamic analysis, Legacy System, Reverse Engineering

1 Introduction

Reverse engineering techniques allow supporting an integral part of software modernization, specifically, the process of analyzing available software artifacts in order to extract information and provide high-level views on the underlying system. Nowadays, many companies are facing the problem of having to modernize or replace their legacy software systems which have involved the investment of money, time and other resources through the ages. Many of them are still business-critical and there is a high risk in replacing them. The growing demand for modernization of software is due to the great advance in mobile technologies and the emergence of the paradigms of Cloud Computing, Pervasive Computing and the Internet of Things. Regarding the systematic modernization process, novel technical frameworks for information integration, tool interoperability and reuse have emerged. Specifically, Model-Driven Engineering (MDE) is a software engineering discipline which emphasizes the use of models and model transformations to raise the abstraction level and the automation degree in software development. Productivity and some aspects of software quality such as maintainability or interoperability are goals of MDE [1].

In the MDE context, the most recent OMG contributions to modernization are in line with the Architecture-Driven Modernization (ADM) proposal. It is defined as "the process of understanding and evolving existing software assets for the purpose of software improvement, modifications, interoperability, refactoring, restructuring, reuse, porting, migration, translation, integration, service-oriented architecture deployment" [2]. The OMG ADM Task Force is developing a set of standards (metamodels) to facilitate interoperability between modernization tools, such as KDM (Knowledge Discovery Metamodel) [3] and ASTM (Abstract Syntax Tree Metamodel) [4]. ADM has emerged complementing OMG standards such as MDA [5], which manages the software evolution from abstract models to implementations. The essence of MDA is the Meta Object Facility Metamodel (MOF) [6] which allows different kinds of artifacts from multiple technologies to be used together in an interoperable way. Metamodeling is an essential technique in MDA and its benefits are well known. The precise standard language definition that is processable by machines may be used to check if models are valid instances. On the other hand, a metamodel defined with the core of UML [7] class diagrams is an accessible language, easy to understand and maintain, therefore it contributes to an easy adaptation allowing language evolution. Based on the level of the meta-metamodel, tools that allow exchanging formats may be developed to manipulate models, regardless of the modeling language used [1].

OMG standards related to ADM allow obtaining models from code that represent static information. Despite the increasing attention to dynamic analysis techniques in reverse engineering, there is no standard for representing information at runtime. A standard for this purpose could be used by tools for visualization and analysis of execution traces, which would facilitate interoperability and data exchange. In previous works, we have shown how to reverse engineering models from code through static analysis, including class, use cases, behavioral and state diagrams [8][9][10]. In this paper, we present TRACEM, a trace metamodel that is the foundation for dynamic analysis in the ADM context. This metamodel allows representing the trace information under a standard representation that supports extensibility, interoperability, abstraction and expressiveness. Moreover, the proposed metamodel along with the specific ASTM aim at automatic instrumentation of the source code. An execution trace model is obtained each time the program runs. Then, by running the program with a significant set of test cases, we obtain a set of trace models that will be analyzed to obtain relevant dynamic information. The ultimate goal is to integrate dynamic and static analysis techniques combining the strengths of both approaches in the reverse engineering process within an MDE framework.

This paper is organized as follows. Section 2 describes a framework for reverse engineering in the MDE context. Section 3 details the TRACEM metamodel. In Section 4, we analyze the impact of dynamic analysis through an example. Section 5 discusses related work. Finally, Section 6 presents conclusions and future work.

2 Reverse Engineering into the MDE Framework

The combination of static and dynamic analysis can enrich the reverse engineering process. Ernst [11] provides a comparison of static and dynamic analysis from the point

of view of their synergy and duality. He argues that static analysis is conservative and sound. Conservatism means reporting weak properties that are guaranteed to be true, preserving soundness, but not strong enough to be useful. Soundness guarantees that static analysis provides an accurate description of behavior, no matter on what input or in what execution environment the program is run. Dynamic analysis is precise given that it examines the actual runtime behavior of the program, however the results of executions may not generalize to other executions. Also, Ernst argues that whereas the main challenge of static analysis is choosing a good abstract interpretation, the main challenge of performing good dynamic analysis is selecting a representative set of test cases. Static or dynamic analyses can enhance one another by providing information that would otherwise be unavailable.

We propose a framework to reverse engineering models that blends the strengths of static and dynamic analysis (Fig. 1). This framework is based on the MDE principles: all artifacts involved can be viewed as models and the process can be viewed as a sequence of model-to-model transformations where the extracted information is represented in a standard way. Each model can be reused, refactored, modified or extended for reverse engineering purposes or for other purposes. Metamodels are defined via MOF and the transformations are specified between source and target metamodels. Then, MOF metamodels "control" the consistency of these transformations.

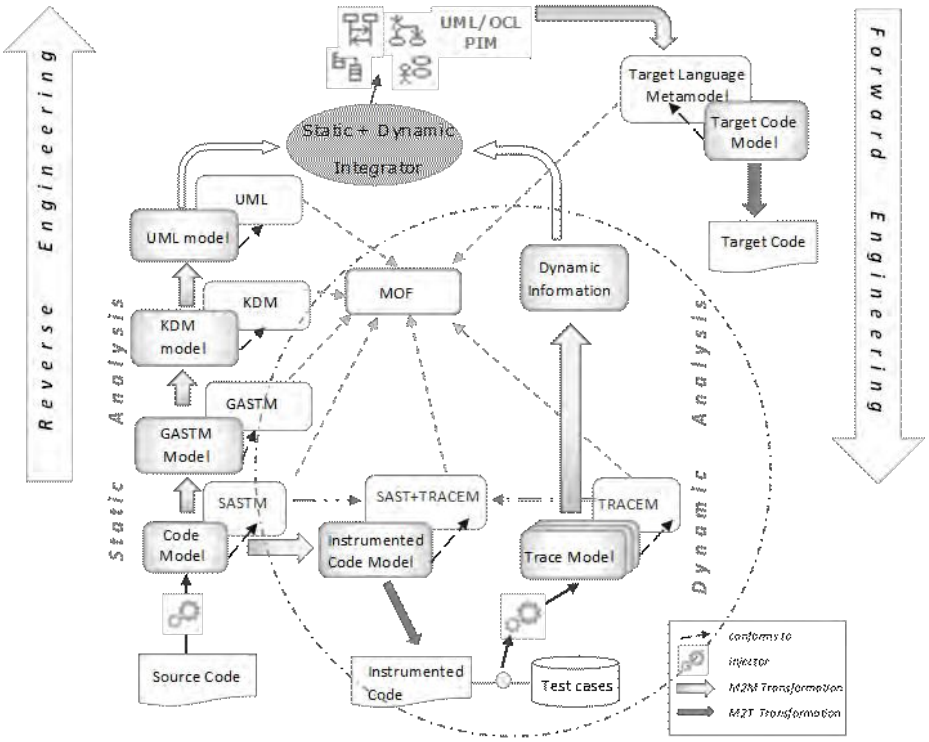

Fig. 1. MDE Modernization Process

In previous works, we present a process to reverse engineering models from code through static analysis, including class diagrams, use cases diagrams, behavioral diagrams and state diagrams [8][9][10]. In the framework, as shown in Figure 1, the first step of the static analysis is to obtain the code model, an abstract syntax tree model instance of the SASTM (Specific ASTM) by using a model injector. Next, an instance of the GASTM (Generic ASTM) is generated from the previous model by a model-to-model transformation. Finally, high-level UML models are obtained by means of a chain of model-to-model transformations, using a KDM model as an intermediate representation of the software system. In the first step of the process, an injector and transformations to obtain the GASTM model must be implemented for each programming language, whereas the sequence of transformations involved in the following steps is independent of the legacy code language .

In this paper we present TRACEM, a trace metamodel that is the foundation for dynamic analysis (see dotted circle in Fig.1). This metamodel allows us to obtain and record trace information. Dynamic analysis provides information about the runtime behavior of software systems, thus, it is a valuable tool for reverse engineering. However, dynamic analysis requires the availability of a full, executable system, which is run with some predefined input data and, on the other hand, it requires the code instrumentation to detect and record relevant events during runtime for later off-line analysis. To reverse engineering models from code, the first stage is to record trace data such as a set of objects, a set of attributes for each object, a location and type for each object, a set of messages, and time stamp for each event. This dynamic information is obtained by instrumenting the source code, a process that inserts additional code fragments into the source code under analysis. An execution trace model, instance of TRACEM, is obtained each time the program runs. Then, by running the program with a significant set of test cases, we obtain a set of trace models. These models will be analyzed to obtain relevant dynamic information that combined with static information allows improving the reverse engineering process. Then, the resulting models are the starting point for the forward engineering process.

3 TRACEM Metamodel

TRACEM allows specifying the trace information under a standard representation supporting extensibility and interoperability. TRACEM was implemented in the Eclipse Modeling Framework [12] that is the core technology in Eclipse for MDE. Figures 2 and 3 partially show this metamodel. The abstract syntax of TRACEM is described by UML class diagram (Fig. 2) augmented with OCL restrictions [13] (Fig. 3). Although the figures show a part of the metamodel, specifically the part focused on the representation of interactions between objects in terms of method calls, it can be extended from the abstract metaclass *Trace* to represent other types of relationships such as inter-process and system-level relationships. The main metaclasses are:

- *ExecutionTrace,* subclass of Trace metaclass, represents a particular execution of a program on a specific test case. Each instance has a name, start and end time, and owns objects and a sequence of statements discovered during the program execution.

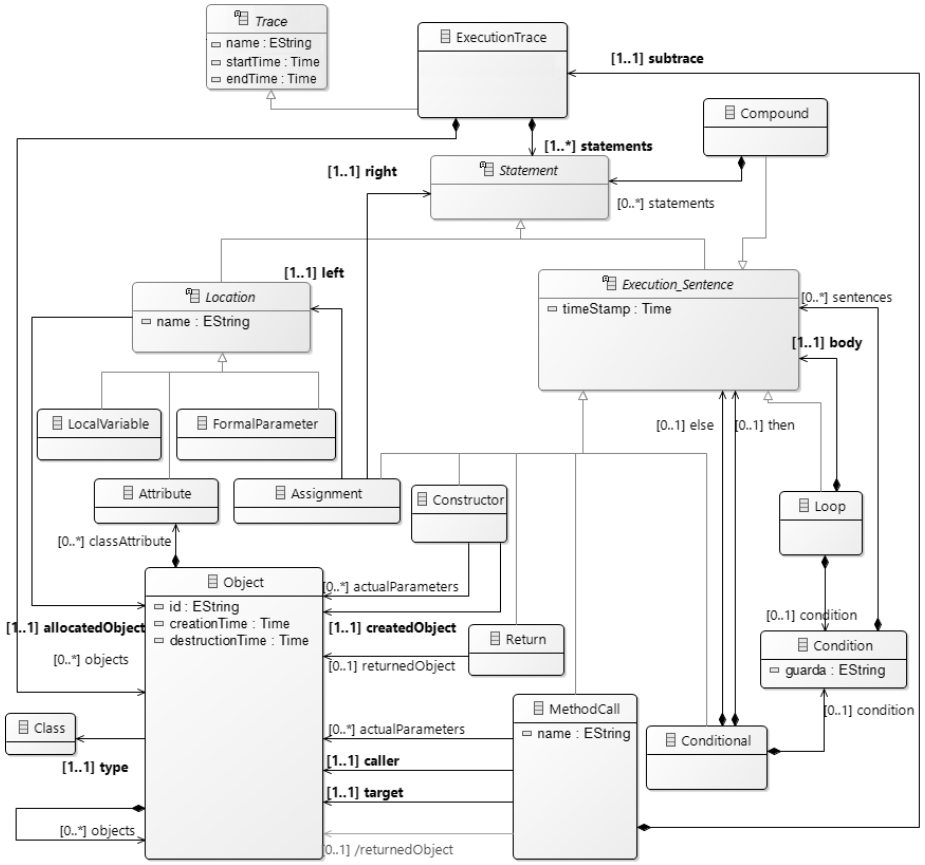

Fig. 2. TRACEM metamodel: Abstract syntax

-- the returned object of a method call corresponds to the object returned by its return sentence
context MethodCall::returnedObject:Object
derived: returnedObject = subtrace.statements->collect (s| s.oclIsTypeOf(return).returnedObject)

context Assignment -- *restrictions on the right and left parts of an assignment*
inv: right.OclIsKindOf(location) **or** right.OclIsKindOf(MethodCall) **or**
right.OclIsKindOf(Constructor) **and** left.allocatedObject =
if right.oclIsTypeOf(Location) **then** right.allocatedObject
else if right.oclIsTypeOf(MethodCall) **then** right.returnedObject
else if right.oclIsTypeOf(Constructor) **then** right.createdObject **endif endif endif**

context Compound -- *compound only has local variables as locations*
inv: statements->select(s | s.oclIsKindOf(Location))-> forAll (l| l.oclIsTypeOf (localVariable))

context MethodCall -- *relationship between formal and actual parameter*
inv: actualParameters->forAll (ap| self.subtrace.statements->
collect(oclIsTypeOf(FormalParameter)) ->exists(fp| fp.allocatedObjet = ap)

Fig. 3. TRACEM metamodel: OCL restrictions

- *Location* is an abstract metaclass that represents a storage that holds an object. Program locations are either local variables, class attributes or method parameters. A *Location* instance has a name and an allocated object which may be changed during program execution.
- *ExecutionSentence* is an abstraction which specifies instructions carried out during the program execution such as *MethodCall*, *Assignment* and *Constructor*.
- *Object* represents objects created during the program execution. An instance has an identifier, a creation and destruction time and owns attributes and objects. It can be stored in different locations throughout the program execution.

4 Recovering Execution Traces from Code: an Example

Dynamic analysis is exemplified in terms of the same case study used in Tonella and Potrich [14], the Java program *eLib* that supports the main library functions (Fig. 4). It contains an archive of documents of different kinds, books, journals, and technical reports. Each of them has specific functionality. Each document can be uniquely identified and library users can request documents for loan. To borrow a document, both user and document must be verified by the Library. As regards the loan management, users can borrow documents up to a maximum number; while books are available for loan to any user, journals can be borrowed only by internal users, and technical reports can be consulted but not borrowed.

```
class Library {
  Map documents = new HashMap();
  Map users = new HashMap();
  Collection loans = new LinkedList();
  ...
  private boolean verifyData (User u, Document d)
  {  if (u == null || d == null) return false;
     if (u.numberOfLoans() <
        MAX_NUMBER_OF_LOANS &&
        d.isAvailable() &&  d.authorizedLoan(u))
                  return true;
     return false;
  }
  public boolean borrowDocument
              (User user, Document doc) {
  if (verifyData (user, doc) {
        Loan loan = new Loan(user, doc);
        addLoan(loan);
        return true;        }
  return false;
  }
  public int numberOfLoans() {
    return loans.size();      }

  private void addLoan(Loan loan) {
    if (loan == null) return;
    User user = loan.getUser();
    Document doc = loan.getDocument();
    loans.add(loan);
    user.addLoan(loan);
    doc.addLoan(loan);   }
...} // end class Library

class Document {...
  public boolean isAvailable() {return loan==null;}
  public boolean authorizedLoan(User user) {
      return true;        }
} // end class Document

class Book extends Document {...}
class InternalReport extends Document {...}
class User {... }
class InternalUser extends User {}
class Loan {
   User user;  Document document;
   public Loan(User usr, Document doc) {
       user = usr;     document = doc;
} // end class Loan
```

Fig. 4. Source code of the *eLib* Program

Tonella and Potrich describe a reverse engineering approach at model level of object-oriented code based on classical compiler techniques and abstract interpretation to obtain UML diagrams from Java code, particularly class, object, interaction, state and

package diagrams. This case study was used in previous works to show the extensions proposed with respect to the approach of Tonella and Potrich [8][9][10]. To highlight the contributions of dynamic analysis, we use the same example.

Dynamic analysis produces a set of execution trace instances, one for each test case. Fig. 5 partially shows an instance of trace metamodel obtained from the execution of the method borrowDocument resulting in a successful loan of the book1 (instance of Book) to the internalUser1 (instance of InternalUser). Each time an object is created, it is identified by the class name concatenated with a numeric value.

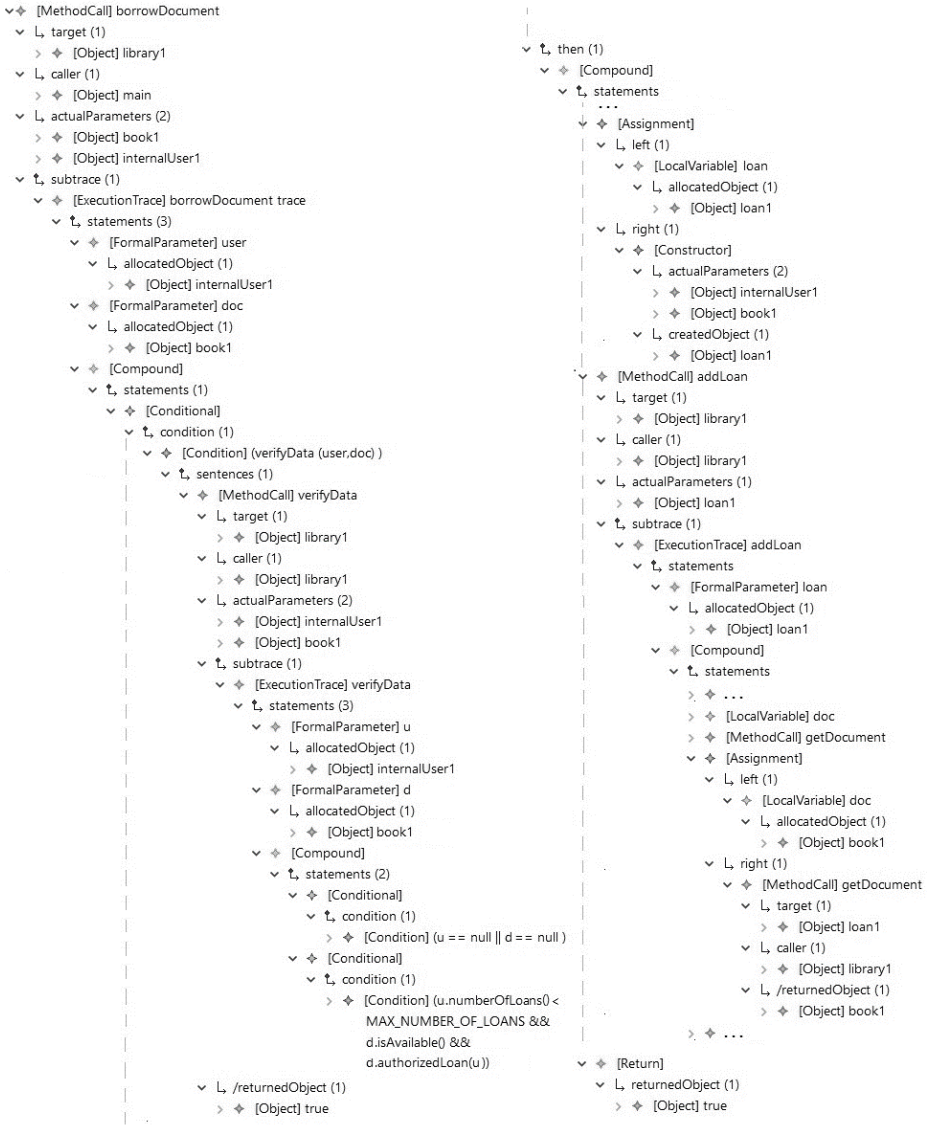

Fig. 5. Trace model: A successful loan

4.1. Dynamic Information Impact

The execution traces provide information that allows complementing the models obtained through static analysis in the aforementioned previous works.

As regards the UML behavioral diagram, it is possible to identify:

- the current object that invokes the method (*caller*) and the one that receives the message (*target*).
- the current parameter linked to each formal parameter, that is, which object is actually stored in each formal parameter for a particular trace. As an example, the objects allocated in the formal parameters *user* and *doc* of the borrowDocument method are the objects *internalUser1* and *book1* respectively (Fig. 5).
- the object flows, that is to say, how an object is passed from one location to another, starting from where it is created. As an example, it is possible to realize that the object *book1* allocated in the formal parameter "*doc*" of the borrowDocument method, is the same object as the one allocated in the formal parameter "*d"* of the verifyData method, the actual parameter "*doc"* of the constructor that creates a new Loan object called loan1, the class attribute "*doc"* of the loan1 object, the local variable "*doc"* in the addLoan method that receives the message addLoan (Fig. 5).
- the kind of dependence relationship between use cases, *include* or *extend*. The common traces reflect primary flow and allow detecting possible *include* relationships between use cases whereas other traces may correspond to *extend* relationships.

As regards the UML structural diagram, it is possible to identify:

- the current objects stored in the generic collections. Containers with weak types (parameterized in abstract types or interfaces) complicate the reverse engineering process. Relationships between classes, such as associations and dependencies, are determined from the declared type for attributes, local variables, and parameters. When containers are involved, the relations to retrieve must connect the given class to the classes of the contained objects. If an attribute type is a generic container, the relationship connects the given class to the class of the contained object, however this information is not directly available in the source code, as a result, the relationship is not depicted in a UML class diagram. Identifying the type of objects that a collection actually stores allows obtaining more complete and accurate class diagrams. As an example, from the trace models, the generic collection *loans* will only contain objects of Loan type, thus, an association between Library and Loan will be inferred.
- composition relationships by analyzing the lifetime of the referenced objects since the metamodel allows recording the creation and destruction time of each object. Within composition, the lifetime of the part is managed by the whole, in other words, when the whole is destroyed, the part is destroyed along with it. As an example, by analyzing the creation and destruction times of the library1 object and the objects of type Loan added to the loans collection of Library, it is possible to infer that the association between Library and Loan is indeed a composition.

Moreover, the execution traces provide information that allows detecting functionality that may never be executed.

5 Related work

Many works have contributed to reverse engineering object-oriented code, dynamic analysis techniques in particular. [14] and [15] perform dynamic analysis to complement the static analysis from java code. Trace information obtained from the program execution is represented with UML models. In the MDE context, [16] presents the first steps towards extending MoDisco with capabilities for dynamic program analysis. MoDisco injects the program structure into a model [17], the authors propose to add execution trace information to the model during program execution. Unlike these works, we propose to represent traces as a new domain in software engineering, independent of any language and providing more expressiveness than those approaches that use UML models to represent the dynamic analysis results.

Following, some works that propose the creation of a standard to represent execution traces are presented. [18] presents a metamodel for representing trace information of routine calls with the aim to develop a standard format for exchanging traces among trace analysis tools. [19] and [20] describe model driven approaches in specific domains that involve dynamic analysis. The former focuses on reverse engineering of AUTOSAR-compliant models using dynamic analysis from trace recordings of a real-time system in the automotive domain. [20] proposes a common metamodel for representing High Performance Computing system traces. Unlike these related works, we propose a MOF-compliant trace metamodel to represent execution traces. This metamodel is the foundation for dynamic analysis within a framework in the MDE context, based on ADM standards in particular.

6 Conclusions

This paper describes the basis for dynamic analysis in the reverse engineering process integrating static analysis, dynamic analysis, and metamodeling in the ADM context. The main contribution is the TRACEM metamodel, which describes concepts and relationships existing in the information obtained from program execution. It allows specifying the execution trace information under a standard representation. Thus, the traces are considered first-class entities, which provide relevant dynamic information that combined with static information allows improving the whole reverse engineering process.

TRACEM together with the metamodels of the different programming languages will allow the automatic instrumentation of code and from this, the injection of trace models that act as decoupling from source technologies. However, there are no available injectors or metamodels for different programming languages and it is necessary to implement them.

We foresee experimenting with different programming languages to implement injector prototypes. Furthermore, we will investigate analysis techniques of execution traces to understand and manipulate the models obtained from program executions.

References

1. Brambilla, M., Cabot, J., & Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, Second edition (2017)
2. ADM Architecture-Driven Modernization. http://www.omg.org/adm
3. KDM ADM: Knowledge Discovery Meta-Model Version 1.4 OMG Document Number: formal/2016-09-01. http://www.omg.org/spec/KDM/1.4 (2016)
4. ASTM Abstract Syntax Tree Metamodel Version 1.0 OMG Document Number: formal/2011-01-05. Standard document URL: http://www.omg.org/spec/ASTM (2011)
5. The Model-Driven Architecture (MDA). http://www.omg.org/mda/ UML OMG Unified Modeling Language. Version 2.5.1, OMG Document Number: formal/2017-12-05. http://www.omg.org/spec/UML/2.5.1/ (2017)
6. MOF OMG Meta Object Facility (MOF) Core Specification. Version 2.5.1, OMG Document Number: formal/2019-10-01. https://www.omg.org/spec/MOF/2.5.1 (2019)
7. UML OMG Unified Modeling Language. Version 2.5.1, OMG Document Number: formal/2017-12-05. http://www.omg.org/spec/UML/2.5.1/ (2017)
8. Favre, L., Martinez, L. & Pereira, C.: Reverse Engineering of Object-Oriented Code: An ADM Approach. In: Handbook of Research on Innovations in Systems and Software Engineering, pp. 386-410. IGI Global (2015)
9. Martinez, L., Pereira, C. & Favre, L.: Recovering Sequence Diagrams from Object-oriented Code - An ADM Approach. Proc. of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2014, pp. 188-195 (2014)
10. Pereira, C., Martinez, L., & Favre, L.: Recovering Use Case Diagrams from Object-Oriented Code: an MDA-based Approach. International Journal of Software Engineering (IJSE), vol. 5 (2) (2012)
11. Ernst, M.: Static and Dynamic Analysis: Synergy and duality. Proceedings of ICSE Workshop on Dynamic Analysis. (WODA 2003), pp. 24-27 (2003)
12. EMF EMF. Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/
13. OCL Object Constraint Language Version 2.4, OMG Document Number: formal/2014-02-03, Standard document URL: http://www.omg.org/spec/OCL/2.4 (2014)
14. Tonella, P., & Potrich, A.: Reverse Engineering of Object-Oriented Code. Monographs in Computer Science. Heidelberg: Springer-Verlag (2005)
15. Systa, T.: Static and Dynamic Reverse Engineering Techniques for Java Software Systems. Ph.D Thesis, University of Tampere, Report A-2000-4 (2000)
16. Béziers la Fosse, T., Tisi, M., & Mottu, JM.: Injecting Execution Traces into a Model-Driven Framework for Program Analysis. In: Software Technologies: Applications and Foundations. STAF 2017. LNCS, vol 10748. Springer, Cham (2018)
17. MoDisco Eclipse MoDisco project.https://www.eclipse.org/MoDisco/
18. Hamou-Lhadj, A., & Lethbridge, T.C.: A metamodel for the compact but lossless exchange of execution traces. Softw Syst Model 11, pp. 77-98 (2012)
19. Sailer, A.: Reverse Engineering of Real-Time System Models from Event Trace Recordings. University of Bamberg Press (2019)
20. Alawneh L., Hamou-Lhadj A. & Hassine J.: "Towards a common metamodel for traces of high performance computing systems to enable software analysis tasks," IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 111-120 (2015)