Exhaustive Simulation and Test Generation Using fUML Activity Diagrams

Iqbal, Junaid; Ashraf, Adnan; Truscan, Dragos; Porres Paltor, Ivan

Published in:
Advanced Information Systems Engineering

DOI:
10.1007/978-3-030-21290-2_7

Published: 01/01/2019

Exhaustive Simulation and Test Generation Using fUML Activity Diagrams

Junaid Iqbal, Adnan Ashraf, Dragos Truscan, and Ivan Porres

Faculty of Science and Engineering
Åbo Akademi University, Turku, Finland
{jiqbal, aashraf, dtruscan, iporres}@abo.fi

Abstract. The quality of the specifications used for test generation plays an important role in the quality of the generated tests. One approach to improve the quality of the UML specification is the use of executable models specified using the Foundational Subset for Executable UML Models (fUML) and the Action language for fUML (Alf). Due to their precise semantics, fUML and Alf models can be simulated or executed using an fUML execution engine. However, in order to execute the models exhaustively, one must provide input data required to reach and cover all essential elements not only in the graphical fUML models, but also in the textual Alf code associated with the graphical models. In this paper, we present an approach for exhaustive simulation and test generation from fUML activity diagrams containing Alf code. The proposed approach translates fUML activity diagrams and associated Alf code into equivalent Java code and then automatically generates: (1) input data needed to cover or execute all paths in the executable fUML and Alf models and (2) test cases and test oracle (expected output) for testing the actual implementation of the system under development. We also present a tool chain and demonstrate our proposed approach with the help of an example.

Keywords: fUML · activity diagram · Alf · simulation · model-based testing · test data generation · Eclipse · Papyrus · Moka

1 Introduction

The Unified Modeling Language (UML) is the de facto standard for modeling software systems. It allows to model the structure and the behavior of the software at a high level of abstraction. UML models can be used for Model-Driven Development (MDD) and Model-Based Testing (MBT). However, UML lacks precise semantics, which hinders the creation of high quality models. To address this problem, the Object Management Group (OMG) has published the Foundational Subset for Executable UML Models (fUML)[1] and Action Language for fUML (Alf)[2] standards. fUML provides precise semantics and allows to create

[1] https://www.omg.org/spec/FUML
[2] https://www.omg.org/spec/ALF

models that are not only executable, but also provide the basis to generate fully functional code.

fUML includes many basic modeling constructs of UML. To implement the precise behavior of the specified system, fUML Activity Diagram (AD) plays an import role. fUML ADs are similar to UML ADs, but they allow to combine and complement the graphical modeling elements with textual syntax specified using the Alf programming language, which is particularly useful for specifying detailed behaviors in complex activities.

There are several fUML implementations, including the open source fUML Reference Implementation[3] and the Moka[4] simulation engine for Papyrus[5], which is an open source Eclipse-based[6] UML editing tool. fUML ADs containing Alf code can be executed and tested in Moka. Model execution and testing allows to examine and improve the functional correctness and the overall quality of models. However, one must provide input data required to reach and execute all important elements in the graphical fUML and textual Alf models. Manual generation of input data might be suitable for small and simple models, but it is often not the case for real-life complex models. Similarly, test generation for executable models is a difficult and tedious task. The work presented in this paper addresses two research questions:

1. *How to automatically generate input data needed to simulate all execution paths in fUML ADs containing Alf code?*
2. *How to generate test cases with oracle from fUML ADs containing Alf code?*

To address these research questions, we present an approach for exhaustive simulation and test generation from fUML ADs containing Alf code. The proposed approach, called *MATERA2-Alf Tester* (M2-AT), translates fUML ADs and associated Alf code into equivalent Java code and then automatically generates: (1) input data needed to cover or execute all paths in the executable fUML and Alf models and (2) a test suite comprising test cases and test oracle (expected output) for testing the actual implementation of the system under development. The generated test cases in M2-AT satisfy 100% code coverage of the Java code. The generated input data is used for executing the original fUML and Alf models in the Moka simulation engine. The interactive execution in Moka allows to determine model coverage of the executable models. In addition, the generated Java code can be reused later on as a starting point for the actual implementation of the system. We also present our tool chain integrated with Papyrus and demonstrate our proposed approach with the help of an example.

The rest of the paper is organized as follows. Section 2 presents relevant background concepts including UML ADs and fUML. In Section 3, we present our proposed M2-AT approach. Section 4 and 5 present an example and an experimental evaluation, respectively. In Section 6, we review important related works. Finally, Section 7 presents our conclusions.

[3] `https://github.com/ModelDriven/fUML-Reference-Implementation`
[4] `http://git.eclipse.org/c/papyrus/org.eclipse.papyrus-moka.git`
[5] `http://www.eclipse.org/papyrus`
[6] `http://www.eclipse.org`

2 Preliminaries

The UML Activity Diagram (AD) is an important diagram for modeling the dynamic aspects of a system[7]. Following the Petri nets semantics, the UML ADs use Petri nets concepts such as places, tokens, and control flows [6,20]. However, the UML AD specification is semi-formal.

UML ADs can depict activities (sequential and concurrent), the data objects consumed or produced by them, and the execution order of different actions. An action specifies a single step within an AD. Edges are used to control the execution flow of the nodes in an activity. A node does not begin its execution until it receives the control or input on each of its input flows. As a node completes its computation, the execution control transits to the nodes existing on its output flows. The execution of an AD is completed if it reaches a final node and/or returns a data object as a result of the internal computations. Passing parameters to an AD as data objects is possible and used for the exchange of information between two actions.

Executable modeling languages allow one to model the specification of the static and dynamic aspects, that is, the executable behavior of the system [4]. The main advantage of executable modeling languages is to specify a software system based on a limited subset of UML comprising class diagrams, state charts, and ADs. The class diagram outlines conceptual entities in the domain while the state chart for each class models the object life cycle. The AD is used to model the behavior of a state in the state chart by exhibiting the sequence of actions to be performed in a particular state [13]. An executable model executes dynamic actions such as creating class instances, establishing associations, and performing operations on attributes and call state events. Meanwhile, in executable UML, the aforementioned dynamic actions are executed via Alf action language which conforms to the UML Action Semantics.

The fUML standard defines the semantics of the class diagrams and ADs for a dedicated virtual machine (called fUML VM) that can interpret both class and activity diagrams [19]. fUML provides concepts similar to object-oriented programming languages, including implementation of operations either by graphical activities or via the Alf action language. Hence, fUML allows one to capture the detailed executable system behavior at the model level. Modeling system behavior in an executable form enables dynamic analysis to be carried out directly at the model level and paves ways for generating fully-functional code from models.

3 MATERA2-Alf Tester (M2-AT)

Figure 1 presents a high-level overview of our proposed *MATERA2-Alf Tester* (M2-AT) approach. The input to M2-AT consist of executable fUML ADs and their associated Alf code. These executable models can be created in the modeling phase of the software development process by refining software requirements

[7] `https://www.omg.org/spec/UML/2.5/`

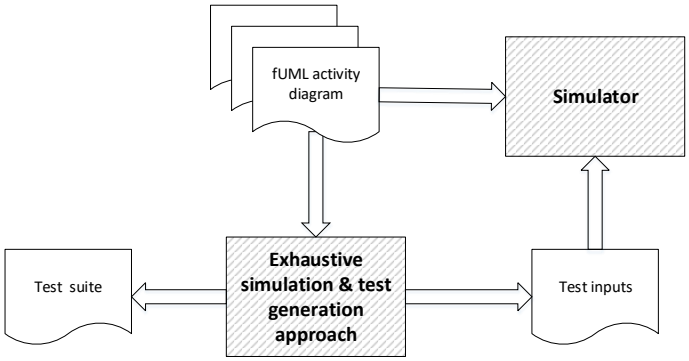


Fig. 1. A high-level overview of the proposed M2-AT approach

and use cases, which define the desired system functionality. M2-AT produces: (1) input data needed for the exhaustive simulation of the fUML ADs and associated Alf code and (2) a test suite comprising test cases and test oracle for testing the actual implementation of the system under development. The generated input data is transformed into an Alf script which allows to use these data in an automated manner.

Internally, the approach is composed of several steps. First, the fUML ADs and their associated Alf code are converted into Java code. Then, we obtain all the inputs of the Java program to achieve 100% coverage of the code. These inputs are used to simulate the AD. Since the Java code and the ADs are behaviorally equivalent, the input will also satisfy 100% coverage of the AD. During the simulation, one can detect and fix problems in the specifications. In the next step, the Java code is used to generate input data and a test suite.

The proposed approach allows to left-shift testing activities in the software development process. In M2-AT, exhaustive simulation of fUML models helps in validating software specifications and improving their quality at an early stage. Moreover, test cases and test oracle are generated before the actual implementation of the system is developed. In the following text, we present the two main phases of the approach namely, translation and input data and test suite generation phase.

3.1 Translation Phase

In order to translate fUML ADs and their associated Alf code into equivalent Java code, M2-AT performs the following steps: (1) separating structural and behavioral elements, (2) generating a dependency graph, (3) topologically sorting the dependency graph to solve node dependencies in the graph, and (4) generating Java code from the sorted dependency graph. Figure 2 presents the translation process.

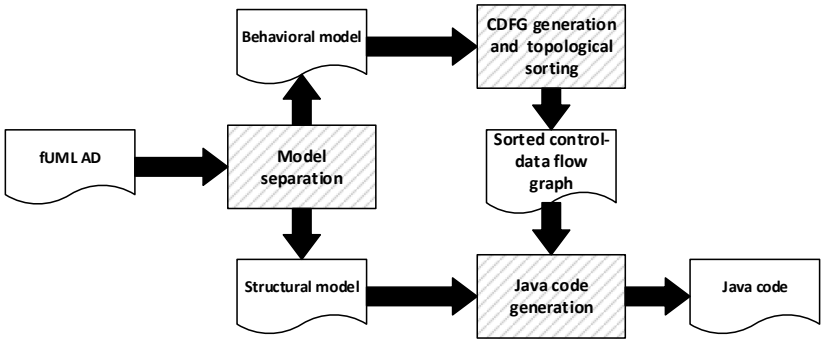


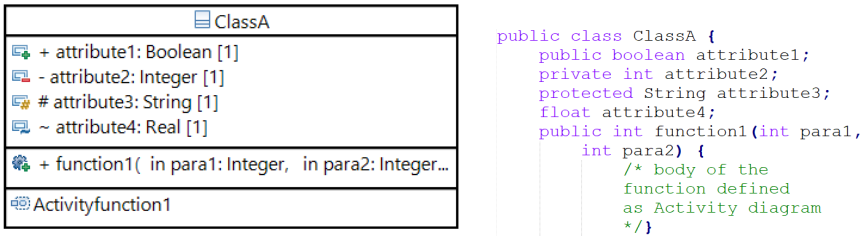
Fig. 2. Translation phase steps

Separating Structural and Behavioral Elements In the first step, the structural and behavioral elements in the executable fUML and Alf models are separated. The structural elements include static features of the systems, while the behavioral elements have a dynamic nature and they represent different interactions among the structural elements. The structural elements can be directly translated into equivalent Java code. However, for behavioral elements a dependency graph is first constructed.

Dependency Graph In order to identify data and control flow dependencies in the behavioral elements, M2-AT constructs a *Control-Data flow graph* (CDFG) [1]. A CDFG is a directed acyclic graph, in which a node can either be an operation node or a control node and an edge represents transfer of a value or control from one node to another.

Topological Sorting of Dependency Graph To solve node dependencies in a CDFG and to decide a starting point for code generation, M2-AT applies topological sorting [11] on CDFGs. The topological sorting algorithm takes a directed acyclic graph G as input and produces a linear ordering of all nodes or vertices in G such that for each edge (v, w), vertex v precedes vertex w in the sorted graph.

Java Code Generation After resolving node dependencies and the order of activity nodes in CDFGs, M2-AT translates structural model elements and sorted CDFGs into equivalent Java code. The structural elements such as packages, classes, interfaces, and associations are used to generator static structure of the Java code. An example of the structural mapping is shown in Figure 3. The fUML class object in Figure 3(a) is directly mapped to a Java class having fUML class properties as Java class attributes and fUML class operations as Java methods [10].



(a) (b)

Fig. 3. Structural mapping between fUML class and Java code:
(a) fUML Class with attributes and operations, and (b) Java code of fUML Class

The fUML ADs are translated into Java code using Java representation for UML activity as presented in the fUML standard. We support a subset of fUML diagram, which excludes asynchronous communication behaviors e.g, *Signal, Messages, and Reception*. Similarly, M2-AT currently does not support parallel execution of fUML nodes. Table 1 shows examples of some fUML AD model elements and their equivalent Java representation. To translate Alf code associated with fUML ADs, we devised a similar mapping that translates each element from Alf code to its equivalent code in Java.

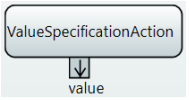
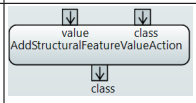
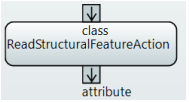
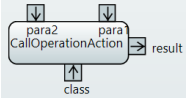
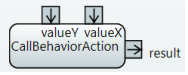
fUML element	Java representation	fUML element	Java representation
	value = ValueSpecifcationAction.value;		class.attribute = value;
	return = class.attribute;		Object object = new Object();
	result = callOperationAction(*para1, para2*);		result = CallBehaviorAction(*valueX, valueY*);

Table 1. Java representation of fUML activity digram nodes

3.2 Input Data and Test Suite Generation Phase

After translating the executable fUML and Alf models to their equivalent Java code, M2-AT uses the generated Java code to produce input data and test suite

Fig. 4. Input data and test suite generation phase

for exhaustive simulation and testing activities. Figure 4 presents the input data and test suite generation steps. The input data and test suite are generated by using EvoSuite [7]. EvoSuite generates and optimizes a test suite for different coverage criteria such as branch coverage, line coverage, method coverage, and exception coverage [8]. It also suggests possible test oracles by adding small and effective sets of assertions that concisely summarize the current behavior. Additionally, one can also use assertions to detect deviations from the expected behavior. In M2-AT, we use line and branch coverage criteria because achieving 100% line and branch coverage in the generated Java code ensures 100% node and edge coverage in fUML ADs.

In order to allow automated use of the generated input data for simulation purposes, M2-AT transforms these data into an Alf script that can be run in the Moka simulation engine. At the end of the simulation, Moka produces a model coverage report.

3.3 M2-AT Tool Chain

Figure 5 presents the M2-AT tool chain comprising M2-AT components along with Papyrus[8] (an open source Eclipse-based UML editing tool), Moka[9] (a simulation engine for Papyrus), and EvoSuite [7] (a test generation tool for Java classes). Papyrus provides a graphical user interface for creating and editing UML and fUML models. In our proposed approach, Papyrus is used for creating fUML models including class diagrams and ADs containing Alf code. M2-AT translates these models into equivalent Java code.

EvoSuite is a search-based tool for automatic generation of test suites from Java classes. Given the name of a target class, EvoSuite produces a set of JUnit[10] test cases aimed at maximizing code coverage. M2-AT uses EvoSuite to generate input data and a test suite from Java code and then transforms the input data into an Alf script that can be run in the Moka simulation engine. Moka is an Eclipse plug-in for Papyrus [21]. It provides support for model execution or simulation, debugging, and logging facilities for fUML models. Moka also allows to measure model coverage and produces a model coverage report at the end of the simulation. When the Alf script is run in Moka, the fUML ADs along with their associated Alf code are executed and a coverage report is produced.

[8] `http://www.eclipse.org/papyrus`
[9] `http://git.eclipse.org/c/papyrus/org.eclipse.papyrus-moka.git`
[10] `https://junit.org/`

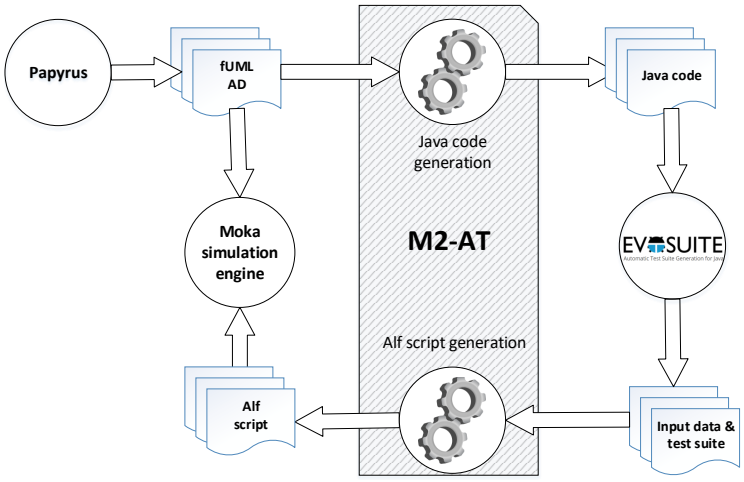


Fig. 5. M2-AT tool chain

3.4 Scalability of the Proposed Approach

The performance and scalability of the proposed approach is based on the time complexities of the M2-AT translation phase (Section 3.1) and the input data and test suite generation phase (Section 3.2). The M2-AT translation phase uses a linear-time topological sorting algorithm [11] that sorts a CDFG with $\mathcal{O}(|V| + |E|)$ complexity, where V and E represent CDFG vertices and edges, respectively. The overall time complexity of the M2-AT translation phase is also linear. Therefore, M2-AT provides highly scalable code generation. The scalability of the M2-AT input data and test suite generation phase is mainly based on the time complexity of EvoSuite, which uses several search-based test generation strategies to optimize test generation time and code coverage. The time complexity of the tool varies from one testing strategy to another and can not be generalized [17].

4 Example

In order to demonstrate the feasibility of our proposed approach, we use an automatic teller machine (ATM) system example originally presented in [15]. The structure of the ATM system is shown in Figure 6. The ATM system can be used to perform withdrawal and deposit transactions in a bank account. The withdrawal operation is realized with the *withdraw* and *makeWithdrawal* methods in the *ATM* and *Account* classes, respectively. Similarly, the *ATM.deposit* and *Account.makeDeposit* methods implement the deposit operation. These operations can be modeled with fUML ADs and Alf code.

Figure 7 and 8 present the fUML ADs for the *ATM.withdraw* and *Account.makeWithdrawal* methods, respectively. To perform a withdrawal trans-

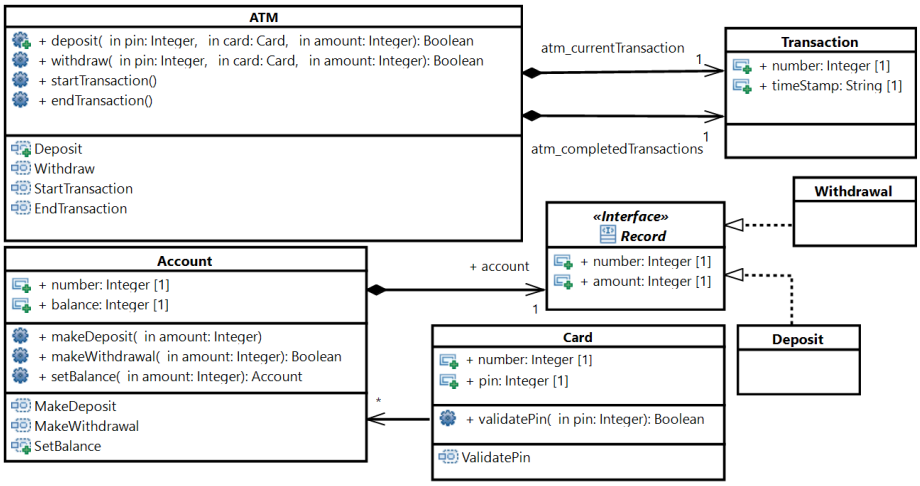


Fig. 6. Class diagram of the ATM system

action, the user inserts an ATM card and enters the associated pin and the amount of money to be withdrawn from the associated bank account. It invokes the *withdraw* method in the *ATM* class, which creates a new transaction and sets it as the current transaction (*startTransaction* method in *ATM* class). Next, it validates the entered pin (*validatePin* method in *Card* class). If *validatePin* returns true, the withdrawal transaction is successfully performed and the account balance is updated (*makeWithdrawal* method in *Account* class). Finally, the completed withdrawal transaction is recorded in the system (*endTransaction* method in *ATM* class).

Please note that the actions *startTransaction*, *validatePin*, *makeWithdrawal*, and *endTransaction* are call actions calling the declared operations. The explained functionality of these operations are implemented by dedicated activities. Additionally, the primitive behaviors such as addition and subtraction are encoded in Alf code. In the remainder of this paper, we use fUML ADs of the *ATM.withdraw* and *Account.makeWithdrawal* methods to demonstrate our proposed approach. ADs of all other operations in the ATM system are omitted due to space limitations.

5 Experimental Evaluation

As presented in Section 3.1, to translate fUML ADs and their associated Alf code into equivalent Java code, M2-AT first separates the structural and behavioral model elements and then generates a CDFG to identify and resolve data and control flow dependencies in the behavioral elements. Figure 9(a) presents the CDFG of the *ATM.withdraw* AD presented in Figure 7. It shows that the *readAccount* method must be invoked before *makeWithdrawal*. Similarly, the *readAc-*

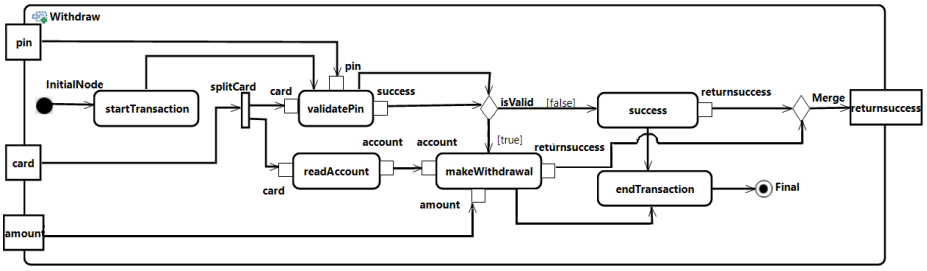


Fig. 7. fUML AD for *ATM.withdraw*

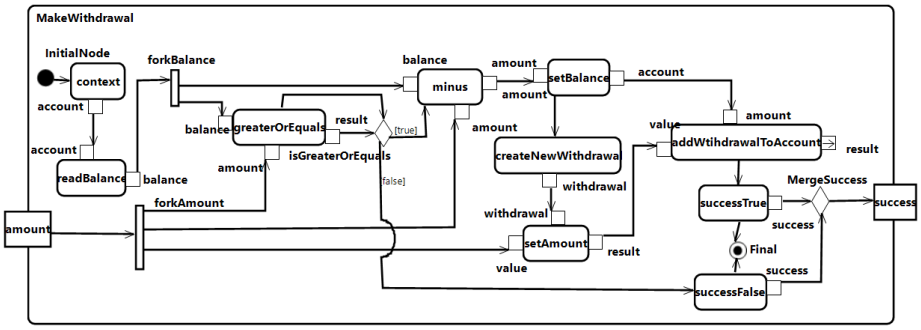


Fig. 8. fUML AD for *Account.makeWithdrawal*

count requires a *Card* object to perform its execution. This data-dependency path is independent of the main control-flow path in the AD, which consist of *InitialNode* → *startTransaction* → *validatePin* → *isValid* → *makeWithdrawal*. In such scenarios, manually deciding a starting point for code generation can be challenging and tedious. Figure 9(b) shows that by using topological sorting of the CDFGs, one can easily resolve all node dependencies in CDFGs and determine the starting point for code generation. Finally, Figure 9(c) shows the Java code generated by traversing the topologically sorted CDFG in Figure 9(b).

In the next step, M2-AT used the Java code in Figure 9(c) to generate input data for model simulation and a test suite for testing the system under development. The initial test suite contained 8 test cases. However, 6 of them were not usable in the Moka simulation engine because they contained invalid *null* values. The invalid cases were also redundant for simulation purposes because they did not have any effect on the node and edge coverage of the *ATM.withdraw* AD. The remaining 2 valid test cases provided 100% node and edge coverage. We parsed the valid test cases to extract input data for model simulation and then transformed the extracted data into an Alf script. Listing 1 presents a fragment of the generated Alf script used for simulating the *ATM.withdraw* AD. Moreover, Figure 10 shows the model coverage results, in which: (1) a solid line represents

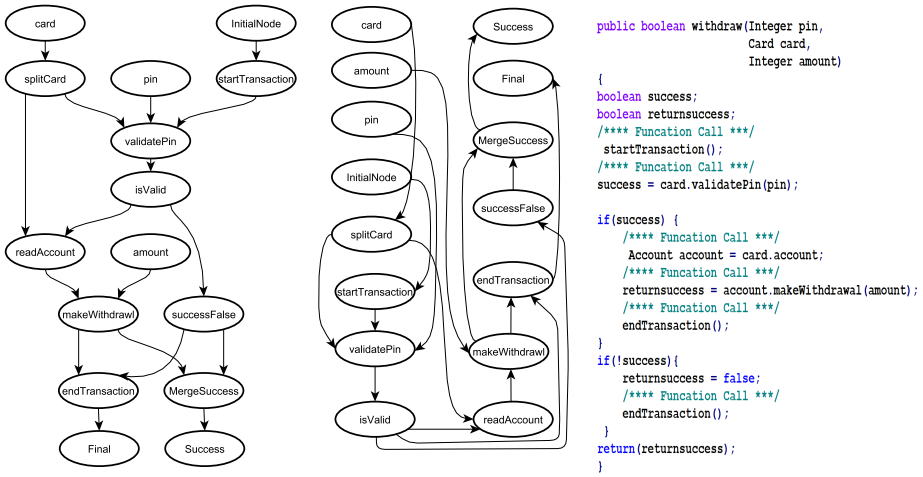


Fig. 9. Code generation from fUML AD: (a) CDFG for *ATM.withdraw* AD, (b) topologically sorted CDFG, and (c) generated Java code.

a covered edge, (2) a dashed line denotes an uncovered edge, and (3) a dotted line represents an unutilized object.

Listing 1. A fragment of the generated Alf script

```
namespace structure;
activity ActivityTester15() {
ATM aTM0 = new ATM();
Card card0 = new Card();
card0.pin = 234532;
card0.number=1;
Account account = new Account();
Boolean boolean0 = aTM0.withdraw(card0.number, card0, card0.number);
}
```

6 Related Work

In this section, we discuss the most important related works on verification of ADs and test and code generation from ADs.

6.1 Verification of ADs

Model verification aims at verifying certain properties in the models under consideration. Model checkers like UPPAAL[11], NuSMV[12], and SPIN[13] verify several

[11] http://www.uppaal.org/
[12] http://nusmv.fbk.eu/
[13] http://spinroot.com/

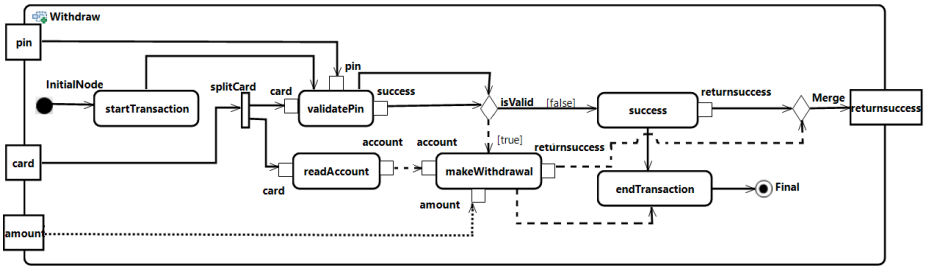


Fig. 10. Model coverage results

properties including deadlock-freeness, reachability, liveness, and safety. Using a model checker for fUML ADs requires that the original or extended ADs are first translated into a graph-based intermediate format and then the intermediate models are translated into the input language of the model checker. For example, for UPPAAL, the intermediate models are translated into UPPAAL Timed Automata (UTA). Daw and Cleaveland [6] translated extended UML ADs into flow graphs and then the flow graphs into the input language of several model checkers including UPPAAL, NuSMV, and SPIN.

Planas et al. [16] proposed a model verification tool called Alf-Verifier that checks the consistency of Alf operations with respect to integrity constraints specified in a class diagram using Object Constraint Language (OCL). For each inconsistency found in the Alf code, the tool returns corrective feedback comprising a set of actions and guards that should be added to the Alf operations to make their behavior consistent with the OCL constraints. Micskei et al. [14] presented a modeling and analysis approach for fUML and Alf. In their approach, the system behavior is first modeled as UML state machines, which are then translated to fUML ADs. In the next step, they manually enrich the fUML ADs with Alf code and then translate them to full Alf code. Finally, the Alf code is translated to UTA to perform model verification. In this approach, Alf is used as an intermediate modeling formalism.

6.2 Test Generation from ADs

Samuel and Mall [18] translated UML ADs to flow dependency graphs (FDGs) to generate dynamic slices for automated test case generation. In their approach, FDGs are created manually, but then an edge marking method is used to generate dynamic slices automatically from FDGs. To generate test data for a dynamic slice, a slice condition is formed by conjoining all conditional predicates on the slice and then function minimization is applied on the slice condition.

Mijatov et al. [15] presented a testing framework for fUML ADs, comprising a test specification language for defining assertions on fUML ADs and a test interpreter for evaluating the defined assertions to produce test verdicts. Tests

are run by executing fUML ADs in an extended fUML VM, which allows to capture execution traces.

Arnaud et al. [2] proposed a timed symbolic execution [5] and conformance testing framework for executable models. Their approach checks correctness of fUML ADs with respect to high-level system scenarios modeled as UML MARTE[14] sequence diagrams. The test data is generated from sequence diagrams by using symbolic execution and constraint solving techniques. The fUML ADs are tested in the standardized fUML VM in Moka. Yu et al. [22] presented a model simulation approach for UML ADs. It uses model-based concolic execution [12], which combines concrete and symbolic execution.

6.3 Code Generation from ADs

Gessenharter and Rauscher [9] presented a code generation approach for UML class diagrams and UML ADs. For the structural part, their approach generates Java code from class diagrams comprising classes, attributes, and associations. For the behavioral part, additional code corresponding to UML activities and actions is added into the Java classes. Their code generator is designed for activities with at most one control node in an activity flow and does not provide support for more realistic, complex flows. Backhauß [3] proposed a code generation approach that translates UML ADs for realtime avionic systems into ANSI-C code. The approach works for control flow edges, but requires further investigations for data flow edges.

In comparison to the aforementioned model verification, test generation, and code generation approaches, the main focus of the proposed M2-AT approach is not on formal verification of ADs. M2-AT provides a light-weight approach that generates input data from fUML ADs containing Alf code and then uses the generated data to exhaustively simulate the original fUML models with the aim of improving their quality. The proposed approach also generates a test suite, which can be used for testing the actual implementation of the system under development. Moreover, it generates and uses topologically sorted CDFGs and Java code as intermediate formalisms. The generated Java code can also be reused for the actual implementation of the system.

7 Conclusion

The Foundational Subset for Executable UML Models (fUML) and the Action language for fUML (Alf) allow to create executable models, which can be simulated using an fUML execution engine. However, to execute such models exhaustively, one must provide input data required to reach and cover all essential elements not only in the graphical fUML models, but also in textual Alf code associated with the graphical models. In this paper, we presented an approach for

[14] `https://www.omg.org/omgmarte/`

exhaustive simulation and test generation from fUML ADs containing Alf code. The proposed approach, called *MATERA2-Alf Tester* (M2-AT), translates fUML ADs and associated Alf code into equivalent Java code and then automatically generates: (1) input data needed to cover or execute all paths in the executable fUML and Alf models and (2) a test suite comprising test cases with oracle (expected output) for testing the actual implementation of the system under development. The generated test cases in M2-AT satisfy 100% code coverage of the Java code. The generated input data is used for executing the original fUML and Alf models in the Moka simulation engine. The interactive execution in Moka allows to measure model coverage of the executable models. In addition, the generated Java code can be reused as a starting point for the actual implementation of the system. We also presented our tool chain and demonstrated our proposed approach with the help of an example. Our proposed tool chain integrates M2-AT code generation and Alf script generation components with the state-of-the-art model simulation and test generation tools allowing researchers and practitioners to generate test suites and input data for exhaustive model simulation at early stages of the software development life cycle.

Acknowledgments

This work has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement number 737494. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Sweden, France, Spain, Italy, Finland, the Czech Republic.

References

1. Amellal, S., Kaminska, B.: Scheduling of a control data flow graph. In: 1993 IEEE International Symposium on Circuits and Systems. pp. 1666–1669 vol.3 (1993)
2. Arnaud, M., Bannour, B., Cuccuru, A., Gaston, C., Gerard, S., Lapitre, A.: Timed symbolic testing framework for executable models using high-level scenarios. In: Boulanger, F., Krob, D., Morel, G., Roussel, J.C. (eds.) Complex Systems Design & Management. pp. 269–282. Springer International Publishing (2015)
3. Backhauß, S.: Code Generation for UML Activity Diagrams in Real-Time Systems. Master's thesis, Institute for Software Systems , Hamburg University of Technology (2016)
4. Breton, E., Bézivin, J.: Towards an understanding of model executability. In: Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001. pp. 70–80. FOIS '01, ACM (2001)
5. Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: Preliminary assessment. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 1066–1071. ICSE '11, ACM (2011)
6. Daw, Z., Cleaveland, R.: Comparing model checkers for timed UML activity diagrams. Science of Computer Programming **111**, 277 – 299 (2015), special Issue on Automated Verification of Critical Systems (AVoCS 2013)

7. Fraser, G., Arcuri, A.: EvoSuite: Automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 416–419. ESEC/FSE '11, ACM (2011)
8. Gay, G.: Generating effective test suites by combining coverage criteria. In: Search Based Software Engineering. pp. 65–82 (2017)
9. Gessenharter, D., Rauscher, M.: Code generation for UML 2 activity diagrams. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) European Conference on Modelling Foundations and Applications (ECMFA). Lecture Notes in Computer Science, vol. 6698, pp. 205–220. Springer Berlin Heidelberg (2011)
10. Harrison, W., Barton, C., Raghavachari, M.: Mapping UML designs to Java. SIGPLAN Not. **35**(10), 178–187 (2000)
11. Kahn, A.B.: Topological sorting of large networks. Commun. ACM **5**(11), 558–562 (Nov 1962). https://doi.org/10.1145/368996.369025
12. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proceedings of the 29th International Conference on Software Engineering. pp. 416–426 (2007)
13. Mellor, S.J., Balcer, M.: Executable UML. A Foundation for Model-Driven Architecture. Addison-Wesleyy (2002)
14. Micskei, Z., Konnerth, R.A., Horváth, B., Semeráth, O., Vörös, A., Varró, D.: On open source tools for behavioral modeling and analysis with fUML and Alf. In: Bordelau, F., Dingel, J., Gerard, S., Voss, S. (eds.) 1st Workshop on Open Source Software for Model Driven Engineering (2014)
15. Mijatov, S., Mayerhofer, T., Langer, P., Kappel, G.: Testing functional requirements in UML activity diagrams. In: Blanchette, J.C., Kosmatov, N. (eds.) Tests and Proofs. pp. 173–190. Springer International Publishing, Cham (2015)
16. Planas, E., Cabot, J., Gómez, C.: Lightweight and static verification of UML executable models. Computer Languages, Systems & Structures **46**, 66 – 90 (2016)
17. Rojas, J.M., Vivanti, M., Arcuri, A., Fraser, G.: A detailed investigation of the effectiveness of whole test suite generation. Empirical Software Engineering **22**(2), 852–893 (Apr 2017). https://doi.org/10.1007/s10664-015-9424-2
18. Samuel, P., Mall, R.: Slicing-based test case generation from UML activity diagrams. ACM SIGSOFT Software Engineering Notes **34**(6), 1–14 (2009)
19. Selic, B.: The Less Well Known UML, pp. 1–20. Springer Berlin Heidelberg (2012)
20. Störrle, H.: Semantics and verification of data flow in UML 2.0 activities. Electronic Notes in Theoretical Computer Science **127**(4), 35–52 (2005)
21. Tatibouet, J., Cuccuru, A., Gérard, S., Terrier, F.: Principles for the realization of an open simulation framework based on fUML (WIP). In: Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium. pp. 4:1–4:6. DEVS 13 (2013)
22. Yu, L., Tang, X., Wang, L., Li, X.: Simulating software behavior based on UML activity diagram. In: Proceedings of the 5th Asia-Pacific Symposium on Internetware. pp. 31:1–31:4. Internetware '13, ACM (2013)