Coverage-driven Automatic Test Generation for UML Activity Diagrams

Mingsong Chen, Prabhat Mishra
Computer & Information Science & Engineering
University of Florida, Gainesville, FL 32611
{mchen, prabhat}@cise.ufl.edu

Dhrubajyoti Kalita
Intel Corporation
1900 Prairie City Road, Folsom, CA 95630
dhrubajyoti.kalita@intel.com

ABSTRACT

Due to the increasing complexity of today's embedded systems, the analysis and validation of such systems is becoming a major challenge. UML is gradually adopted in the embedded system design as a system level specification. One of the major bottlenecks in the validation of UML activity diagrams is the lack of automated techniques for directed test generation. This paper proposes an automated test generation approach for the UML activity diagrams. The contribution of this paper is the use of specification coverage to generate properties as well as design models to enable directed test generation using model checking. Our experimental results demonstrate that our approach can drastically reduce the validation effort in both specification and implementation levels.

Categories and Subject Descriptors: I.6.4 Simulation and ModelingModel Validation and Analysis

General Terms: Verification

Keywords: Test Generation, UML Activity Diagrams

1. INTRODUCTION

There is a noticeable trend in the embedded system design that the design flow starts from high level specifications in both hardware and software designs. As a general purpose modeling language, Unified Modeling Language (UML) [1] is becoming a promising specification for both software and hardware designs [2, 3]. UML activity diagram is a semi-formal specification that adopts the Petri net-like [4] semantics to describe the workflow of the system as well as the internal logic of a complex operation. Because of the ability to describe the global ordering of atomic pieces of behavior(i.e. activities), UML activity diagram can be used to model the dynamic concurrent scenarios of a group of objects, or the control flow of an operation. Therefore, the UML activity diagram is well suited for system level design of embedded systems.

Automatic test generation from high level specifications can have double impacts: i) the generated tests can be used to verify the specification to ensure its correctness, ii) the same tests can be reused to verify the implementation. As an abstract level executable specification, UML activity diagrams capture the key system behaviors. The tests generated from the UML activity diagram can not only be used to guarantee the consistency between abstraction levels, but also it can be reused to reduce the overall validation effort. However, there is a lack of automated techniques for directed test generation from UML activity diagrams. This paper presents an automated approach for directed test generation using model checking. It makes three important contributions: i) define specification coverage of UML activity diagrams to generate the required properties, ii) propose a set of rules to transform the UML activity diagrams to a formal model, and iii) apply model checking to perform test generation using the generated properties and the formal model.

The remainder of the paper is organized as follows. Section 2 describes the related work addressing validation of UML activity diagrams. Section 3 briefly describes the modeling of UML activity diagrams. Section 4 presents our test generation methodology followed by a case study in Section 5. Finally, Section 6 concludes the paper.

GLSVLSI'08, May 4–6, 2008, Orlando, Florida, USA.

2. RELATED WORK

There are several research works that use model checking based techniques for verifying UML activity diagrams. Eshuis [5] presents a translation procedure from UML activity diagrams to the input language of NUSMV [6]. However, the translation is used to verify the consistency of a UML activity diagram and a set of class diagrams. Guelfi et al. [7] provide a formal definition of syntax and semantics for UML activity diagrams endowed with time aspects. They outline the translation from the semantics into the PROMELA - an input language of the SPIN model checker. Das et al. [8] propose a method to deal with timing verification of UML activity diagram models. These activities primarily focus on checking the consistency or correctness of the model itself instead of generating directed tests.

Model driven testing [9] inspires the idea of automatic test generation from the UML activity diagrams. Wang et al. [10] presented an approach to generate test case from UML activity diagrams based on Gray-Box method. However, they did not provide any procedure to automatically generate the tests from the extracted scenario. Chen et al. [11] proposed a method that selects the tests from the random tests based on the coverage criteria of the UML activity diagrams for Java programs. But this method can not guarantee the selected tests can give a good coverage result because of the randomness. Furthermore, generating the program execution traces is very time-consuming and can be infeasible in many scenarios. To the best of our knowledge, there are no existing approaches that define fault model at the specification level and use it to generate directed tests from UML activity diagrams.

3. MODELING OF ACTIVITY DIAGRAMS

This section briefly describes UML activity diagrams. First, we formally define several aspects of activity diagrams which will be used in the test generation. Next, we define the coverage criteria of the UML activity diagrams.

3.1 Definitions

Figure 1 shows an activity diagram which uses most of the elements of the UML activity diagram. It describes the functionality of withdrawing money from the ATM. The user needs to enter the access code first, and in case of failure, he can input the access code again. The operation will abort if access code is wrong in both cases. If the input access code is right, the user will enter the amount of the money he wants to withdraw. At the same time, in order to print the receipt, the printer will be warmed. Once the ATM decides whether there is enough money the user can withdraw, it provides cash and generates the information of this withdraw operation. Finally, the printer will print the receipt.

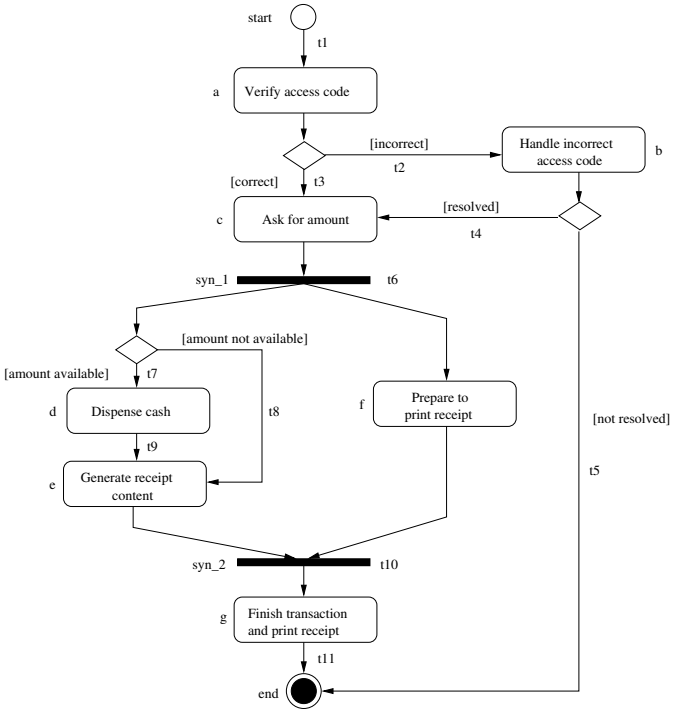

Figure 1: An example of UML activity diagram

In this paper, we mainly consider the control and data flow of activity diagrams that are relevant to test generation.

Definition 1. An activity diagram is a six-tuple $D = (A, T, F, C, a_I, a_F)$ where

- $A = \{a_1, a_2, \ldots, a_m\}$ is a finite set of activity states.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of completion transitions.
- $C = \{c_1, c_2, \ldots, c_n\}$ is a finite set of guard conditions, and c_i is in correspondence with t_i, $Cond$ is a mapping from t_i to c_i so that $Cond(t_i) = c_i$.
- $F \subseteq \{A \times T\} \cup \{T \times A\}$ is the flow relation between the activities and transitions.
- $a_I \in A$ is the *initial state*, and $a_F \in A$ is the *final state*. There is only one transition $t \in T$ such that $(a_I, t) \in F$, and for any $t' \in T$, $(t', a_I) \notin F$ and $(a_F, t') \notin F$. ■

Definition 1 describes the relation between the activities and transitions. It is a static structure of activity diagram. At any time, when analyzing an activity diagram, the current state (denoted by CS) of an activity diagram is represented by a set of activity states.

Definition 2. Let $D = (A, T, F, C, a_I, a_F)$ be an activity diagram. The *current state* CS of D is a subset of A. For any transition $t \in T$,

- ${}^\bullet t$, $t^\bullet$ denote the preset and postset of τ respectively, then ${}^\bullet t = \{\ a \mid (a,t) \in\ F\ and\ a \in A\}$ and $t^\bullet = \{a \mid (t,a) \in F\ and\ a \in A\}$.
- $enabled(CS)$ denotes the set of transitions that start from CS, then $enabled(CS)$ = $\{\ t \mid {}^\bullet t\ \subseteq\ CS\ \}$.
- *firable(CS)* denotes the set of transitions that can be fired from CS , then *firable(CS)*=$\{\ t \mid t \in enabled(CS)$ and ${}^\bullet t$ are all completed and $Cond(t)$ is satisfied and $(CS - {}^\bullet t) \cap t^\bullet = \emptyset\}$, and after some t is fired, the new current state $CS' = fire(CS, t) = (CS - {}^\bullet t) \cup t^\bullet$. ■

For example, when $\{d, f\}$ is the current state of the activity diagram, only the transition t_9 is firable at this time. If t_9 is fired, then the next state is $\{e, f\}$.

Because of the inherent concurrency, several transitions can be fired at the same time. For an activity diagram, all the firable transitions in a state form a $concurrent\ transition$.

Definition 3. Let $D = (A, T, F, C, a_I, a_F)$ be an activity diagram. For a state CS of D, a concurrent transition τ is a set of transitions $t_1, t_2, ..., t_n \in firable(CS)$ where

1. $\forall\, i, j (1 \le i < j \le n)$, ${}^\bullet t_i \cap {}^\bullet t_j = \emptyset$;
2. $\forall\ t \in (enabled(CS) - \{t_1, t_2, ..., t_n\})$, there exists i $(1 \le i \le n)$ such that ${}^\bullet t \cap {}^\bullet t_i \neq \emptyset$.

After firing τ from state CS, the current state $CS' = fire(CS, \tau) = \bigcup_{i=1}^{n}(fire(CS, t_i)) = \bigcup_{i=1}^{n}((CS - {}^\bullet t_i) \bigcup t_i^\bullet)$. ■

An instance of dynamic behavior of an activity diagram can be represented by a sequence of states and concurrent transitions. We call it a *path* of the activity diagram.

Definition 4. Let $D = (A, T, F, C, a_I, a_F)$ be an activity diagram. A path ρ of the activity diagram is a sequence of states and transitions, let

$$\rho = cs_0 \xrightarrow{\tau_0} cs_1 \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_{n-1}} cs_n$$

where $cs_0 = \{a_I\}$, $cs_n = \{a_F\}$, and $cs_{i+1} = fire(cs_i, \tau_i)$ for any i $(0 \le i < n)$. ρ is a $keypath$ if there is no repetition in ρ, i.e. $\forall i, j\ (0 < i < j \le n)$, $cs_i \bigcap cs_j = \emptyset$. ■

There are five key paths in Figure 1. For the key path, when firing the transitions, we need to consider the condition guard expressions.

3.2 Test Adequacy Criteria

Objective measurement of the test quality is one of the key issues in the effort of testing. Test adequacy criterion specifies the requirement of a particular testing. In software testing, the definition of test adequacy is given in [12] as a measurement function. The situation of UML activity diagram is different because it is in the form of model instead of code. Especially the path coverage of activity diagram is complicated because of the concurrency. The definition of coverage of UML activity diagram is as follows:

- *Activity Coverage* requires that all the activity states in the activity diagram be covered. The value of $activity\ coverage$ is the ratio between the checked activities and all the activities in the activity diagram.

- *Transition Coverage* requires that all the transitions in the activity diagram be covered. The value of $transition\ coverage$ is the ratio between the checked transitions and all the transitions in the activity diagram.
- *Key Path Coverage* requires that all the key paths in the activity diagram be covered. The value of $key\ path\ coverage$ is the ratio between the traversed key paths and all the key paths in the activity diagram. ■

4. TEST GENERATION METHODOLOGY

This section describes our automated approach for coverage directed test generation of UML activity diagrams. Figure 2 shows our test generation methodology. The UML activity diagram is translated to a formal model (NUSMV input). Next, the properties in the form of CTL or LTL formulas can be generated from the coverage criteria. Finally, the properties (negated version) are applied on the formal model using model checking to generate required tests (counterexamples).

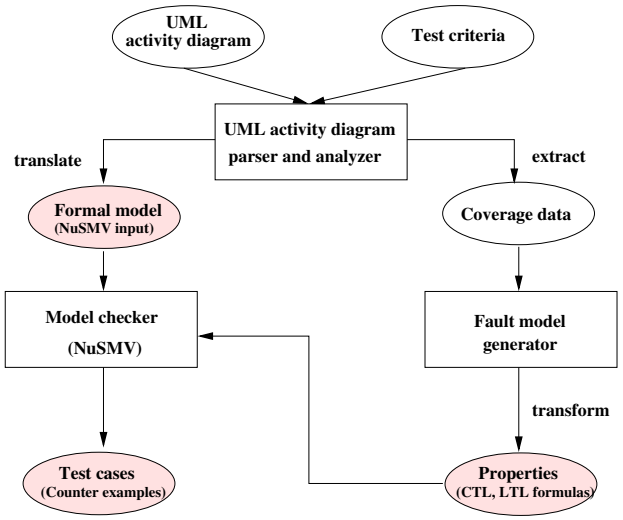

Figure 2: The test generation framework

4.1 Translation

Our technique can get both the control and data flow by parsing the UML activity diagram. The translation is a process of the mapping from control and data flow to the NUSMV input. The translation consists of two parts: structure extraction and behavior extraction.

Structure extraction analyzes the structure of the activity diagram and then generates the skeleton of the NUSMV input. It first collects all the data which is manipulated in the activities and the data which is referred at the condition of the transitions. For example in the Figure 1, there are five data members: $access\_code$, $access\_code\_input$, $access\_code\_resolve$, $amount\_input$, and $amount\_available$. Because the value range of each data member is continuous or a large set, in the model checking it may cause the state space explosion. So we can use the partition method to generate proper combination of values of input and output parameters to satisfy the condition constraints.

The behavior extraction analyzes the behavior of the system, such as the state change of each activities, data manipulation and the condition of the transitions. In the semantics of activity diagram, a $keypath$ is an dynamic execution scenario from the initial node to the final node. To get all the $keypaths$ of an activity diagram, we can traverse the diagram by using DFS (depth first search) which is similar to the method in [11]. But in our framework, we need to extract both the data manipulations and the conditions of transitions, because they will determine the executions of $keypaths$.

4.2 Fault Model and Property Generation

When generating directed tests, it is required to use some notion of coverage to indicate the sufficiency. Section 3.2 presents the coverage of the activity diagram. Based on this, we can create the fault model which is the negation of the coverage requirement. The fault model in this paper is as follows.

Definition 5. Let AD be an activity diagram, there are three fault models:

- **Activity fault model**. For each activity of AD, the model assumes that such activity is not reachable.
- **Transition fault model**. For each transition of AD, the model assumes that such transition can not be fired.
- **Key path fault model**. For every key path of AD, there is no corresponding executable $path$. ■

From these three different models, we can generate the tests to validate the various properties of the system. The activity fault model can be used to check the reachability of each activity. So it can be used to check whether there exists infinite loops in the system. The transition fault model can be used to check the execution order of the activities. It can also be used to check whether the condition guard of the transition can be satisfied. Sometimes, we need to check all the dynamic behaviors of the system, so key path fault model is preferable in this case. The transformation from the fault model to the properties (of the formal model) in the form of NUSMV is a one-to-one mapping.

5. CASE STUDY

We compare our approach with the random test based method [11], which is the best known result in this category. The experimental results demonstrate that our method can drastically reduce both the test generation time and overall validation effort by producing high quality tests for the implementation.

5.1 A Control System

The first case is a small control system. The activity diagram consists of 17 activities, 23 transitions and 6 key paths. Table 1 shows the comparison between our approach and the random test based method [11]. For generating the tests with highest coverage, the random method requires 8.83 seconds to run the 150 random tests, however our method needs 0.91 seconds. In this case study, our approach improves the test generation time by an order of magnitude.

Table 1: Comparison of two methods

Method	Coverage (%)			Time
	activity	transition	path	(second)
random 30	90	85	50	1.33
random 50	95	93	67	2.35
random 100	100	100	83	5.13
random 150	100	100	100	8.83
Our method	100	100	100	0.91

By running the tests derived from the fault models, we can get a 100% activity coverage and transition coverage on the activity diagram. However, at the beginning of the experiment, the key path coverage was 83.3% because the generated tests could not enable one of the six key paths. Further analysis revealed that this particular path was a false path – it can never be activated by any tests due to conflicting conditions used in that path. So we corrected this error and achieved 100% key path coverage. The random test based method [11] can only be used for implementation validation. However, the tests generated by our approach can be used for validation of specification as well as the implementation.

Table 2: Implementation level coverage of the control system

Package	Class	Method	Block	Line
100%	100%	90%	88%	93%

We applied the generated tests to the Java implementation of the control system. Table 2 shows the coverage of the Java code. The generated tests obtained 100% package as well as class coverage. However, the $method$, $block$ and $line$ coverage is around 90%. Our analysis showed that the Java implementation have many "try" and "catch" blocks to handle exceptions whereas the specification does not have any information on the exception scenarios. As a result, the generated tests did not activate any of the exception blocks which also resulted in low coverage of methods, blocks as well as lines. Clearly, this is an issue of incomplete specification. Based on this observation, we added exception information at the specification level and generated tests which led to required coverage in all the categories of the implementation.

5.2 A Stock Exchange System

The purpose of the on-line stock exchange system (OSES) [11] is to process the following scenarios: accept, check and execute the customer's orders (market order and limit order). The system uses the UML activity diagram as its behavior specification. It has 27 activities, 29 transitions and 18 key paths. The system is implemented in JAVA and consists of 7 packages, 39 classes, 372 methods and 2510 lines. This system is much larger than the first case study.

Table 3: Comparison of two methods

Method	Coverage (%)			Time (minute)
	activity	transition	path	
random 800	96	83	89	19.06
random 1000	96	86	94	24.26
random 1500	100	100	100	30.25
Our method	100	100	100	7.08

In Table 3, the first three rows depict the results by using 800, 1000, 1500 random tests respectively. The result by our method is shown in the last row. In the case of $random$ 800, two key paths are missing due to the randomness. So the coverage metrics are not 100%. If we raise the number of the random tests to 1000, one key path is still missing. Based on our observation, in the random method, it is hard to determine what is an appropriate upper bound for the random test number. And it is hard to discover whether the specification is correct by the random tests. Clearly, our approach reduced the validation effort by four times compared to the best known result in this category.

Table 4: Implementation level coverage of OSES

Package	Class	Method	Block	Line
100%	100%	58%	55%	51%

Table 4 presents the coverage in the implementation level by applying the generated tests using our method. The coverage of $method$, $block$ and $line$ are not sufficient because the activity diagram does not consider all the scenario of the system, such as the registration of the customers and so on. In this case, we needed to add the missing details in the specification to obtain the required coverage.

6. CONCLUSIONS

In this paper, we proposed an approach to automatically generate tests from UML activity diagrams. Our experimental results demonstrated that the generated tests can produce the required functional coverage and also can make a significant reduction in validation of specification as well as implementation. Model checking based test generation is promising but it can lead to state space explosion in the presence of complex designs and properties. Our future work will investigate various design and property decomposition techniques to reduce the test generation complexity.

7. ACKNOWLEDGMENTS

This work was partially supported by grants from Intel Corporation and NSF CAREER award 0746261.

8. REFERENCES

[1] OMG. *UML2.0 Superstructure Specification*. Available at http://www.omg.org/#UML2.0, October 2004.
[2] Grant Martin. UML for Embedded Systems Specification and Design: Motivation and Overview. *DATE*, 2002.
[3] W. Mueller, et al. UML for ESL Design - Basic Principles, Tools, and Applications. *ICCAD*, 2006.
[4] J. Peterson. *Petri Nets Theory and the Modeling of Systems*. Prentice-Hall, N.J., 1981.
[5] Rik Eshuis. Symbolic model checking of UML activity diagrams. *ACM Transactions on Software Engineering and Methodology* , 15(1), 2006.
[6] A. Cimatti, E. M. Clarke, F. Giunchiglia, M. Roveri. NUSMV: A New Symbolic Model Verifier. *CAV*, 1999.
[7] N. Guelfi and A. Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. *APSEC*, 2005.
[8] Dipankar Das, Rajeev Kumar and P. P. Chakrabarti. Timing Verification of UML Activity Diagram Based Code Block Level Models for Real Time Multiprocessor System-on-Chip Applications. *APSEC*, 2006.
[9] R. Heckel and M. Lohmann. Towards Model-Driven Testing. *TACoS*, 2003.
[10] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. *APSEC*, 2004.
[11] M. Chen et al. UML Activity Diagram Based Automatic Test Case Generation for Java Programs. *The Computer Journal*, doi:10.1093/comjnl/bxm057.
[12] H. Zhu, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4), 1997.