Des Autom Embed Syst (2010) 14: 105–130
DOI 10.1007/s10617-010-9052-4

Efficient test case generation for validation of UML activity diagrams

Mingsong Chen · Prabhat Mishra · Dhrubajyoti Kalita

Received: 20 August 2009 / Accepted: 30 April 2010 / Published online: 18 June 2010

Abstract Unified Modeling Language (UML) is widely used as a system level specification language in embedded system design. Due to the increasing complexity of embedded systems, the analysis and validation of UML specifications is becoming a challenge. UML activity diagram is promising to modeling the overall system behavior. However, lack of techniques for automated test case generation is one major bottleneck in the UML activity diagram validation. This article presents a methodology for automatically generating test cases based on various model checking techniques. It makes three primary contributions: First, we propose coverage-driven mapping rules that can automatically translate activity diagram to formal models. Next, we present a procedure for automatic property generation according to error models. Finally, we apply various model checking based test case generation techniques to enable efficient test case generation. Our experimental results demonstrate that our approach can reduce the validation effort drastically by reducing both test case generation time and required number of test cases to achieve a functional coverage goal.

Keywords UML activity diagram · Testing · Model checking · Property decomposition

1 Introduction

Functional validation is becoming a major bottleneck in embedded system design due to the increasing complexity of both hardware and software components. As the most widely used

M. Chen (✉) · P. Mishra
Department of Computer & Information Science & Engineering, University of Florida, Gainesville, FL 32611, USA
e-mail: mchen@cise.ufl.edu

P. Mishra
e-mail: prabhat@cise.ufl.edu

D. Kalita
Intel Corporation, 1900 Prairie City Road, Folsom, CA 95630, USA
e-mail: dhrubajyoti.kalita@intel.com

validation form, simulation adopts three types of test case generation techniques: random, constrained-random and directed. Compared to the other two random methods, directed test cases can save the overall validation effort since fewer test cases can obtain the same coverage goal. However, lack of automated techniques is a common problem for directed test case generation.

Unified Modeling Language (UML) [1, 2] is becoming a promising specification language for both software and hardware designs [3–5]. It provides a set of diagrams that can capture different system views. Also it supports profiles that can be used for customizing UML models for a particular domain and purpose. As a kind of behavior specification, UML activity diagrams adopt semantics similar to the formally defined Petri-net [6]. However, unlike Petri-nets, the UML activity diagram is a semi-formal specification which is more intuitive and flexible to describe the concurrent behaviors of the system as well as the internal logic of complex operations. Therefore it is widely used as a front-end tool for system level design of software/hardware systems.

In a top-down design flow, validation of UML specifications is necessary to ensure the correctness of both the specified design and the subsequent implementation. Validation of UML activity diagrams using directed test cases is very promising. However, most directed test case generation work is performed by human intervention. Hand-written test cases entail laborious and time-consuming effort of verification engineers who have deep knowledge of the design under verification. Due to the manual development, it is difficult to generate all directed test cases to achieve a coverage goal. The problem is further aggravated due to the lack of comprehensive functional coverage metrics. Automatic directed test case generation based on a comprehensive functional coverage metric is the alternative to address this problem.

Model checking techniques have been proposed in the past for automated test case generation to validate software designs [8]. However, they have not been studied before in the context of automated generation of directed test cases for UML activity diagrams. In this article, we propose a directed test case generation methodology for UML activity diagrams. Our framework can translate UML activity diagram specifications to the model checker input. The properties can be automatically generated based on the coverage (e.g. node coverage, path coverage, etc.) of the specification. Each property asserts a valid behavior of the UML activity diagram to be checked. When checking the negation form of this property, one counterexample will be reported. The counterexample activates the required behavior using a sequence of variable assignment, so it can be treated as a test case. The generated test cases can not only be used to guarantee the consistency between different abstraction levels, but also they can be reused to reduce the overall validation effort. In this article, we explore two alternatives for directed test case generation: test case generation using Binary Decision Diagram (BDD) [9] based model checking, and Boolean SATisfiability (SAT) [10] based Bounded Model Checking (BMC) [11]. We compare these two methods in terms of their applicability and limitations and propose initial ideas to address some of the practical challenges in applying them on industrial designs.

The remainder of this article is organized as follows. Section 2 describes the related work addressing validation of UML activity diagrams. Section 3 gives the preliminary knowledge for the activity diagram and model checking. Section 4 presents the framework of our test case generation methodology in details. Several case studies demonstrate the efficiency of our proposed framework in Sect. 5. Finally, Sect. 6 concludes the article.

2 Related work

As a system level specification, UML diagrams are becoming popular in embedded system design [4, 12–14]. To guarantee the quality of embedded systems, model-based test process [15] is proposed. There are significant research efforts on model driven testing of UML diagrams [18]. The generated high level test cases are useful because they can be used to validate both specifications and implementations. Based on UML diagrams, Briand and Labiche [19] proposed a methodology to support the system testing. Their work supports the derivation of the system test requirement which can be transformed to test cases. For UML activity diagrams, some tools and methods have already supported test specifications and test case generation. For example, dSPACE developed the tool AutomationDesk which uses activity diagrams for test descriptions and test script generation [16, 17]. Chen et al. [20] presented a framework that can construct test cases from specifications by identifying a set of input categories for the activity diagrams as test cases. However, their method requires preliminary information provided by testing experts. Wang et al. [21] presented an approach based on Gray-Box method which defines test cases according to the dynamic scenarios of a system. However, the proposed algorithm can not capture all the concurrent scenarios. So the testing adequacy can not be guaranteed. Kim et al. [22] tried to convert an activity diagram to a directed graph. Then test cases can be generated from the directed graph. But this method lacks concrete rules for the automatic transformation. Chen et al. [23, 24] proposed a framework that can sift random test cases based on the coverage criteria of the UML activity diagrams. Because of the randomness, this approach can not guarantee that the selected test cases can achieve a required coverage in a reasonable timeframe.

Formal verification can be used to verify the correctness of specifications, so it can be used to guarantee the quality of UML models [25]. UML activity diagram adopts Petri-net semantics which is promising to describe the concurrent behavior [21, 23, 24]. In [26], Bell and Haverkort presented sequential as well as distributed algorithms for model checking Petri-nets. Their approach allows checking large systems with hundreds of millions states in an efficient way. Jensen et al. [27] introduced the Colored Petri-net (CPN) which enables discrete-event modelling of concurrent system. The CPN supports both simulation and model checking to conduct performance analysis as well as property validation. As an alternative, there are several approaches based on state machine that use model checking techniques to verify UML activity diagrams. Eshuis [28] presented a translation procedure from UML activity diagrams to the input language of NUSMV [29]. However, the translation is used to verify the consistency between UML activity diagrams and class diagrams. It focuses on checking the consistency between two different models. Guelfi and Mammar [30] provided a formal definition for timed activity diagrams. They outlined the translation from the semantic specifications into PROMELA—an input language of the SPIN model checker. Das et al. [31] proposed a method to deal with timing verification of UML activity diagram models. All these verification work primarily focus on checking the consistency or correctness of the model itself instead of generating directed test cases.

Reusing the validation effort in the top-down design context can reduce the overall validation time. By our observation [32], the high level test cases are very promising for the validation of low level implementations. Various efficient model checking techniques are proposed for automated test case generation from high-level specifications [8, 36–38]. Fraser and Wotawa [33] discussed several aspects of model checkers when used for testing and addressed ten challenges in which model checkers could be enhanced to be better suited for the task of test case generation. Mishra and Dutt [34] presented a functional coverage based test case generation technique for pipelined architectures. Their experimental results

demonstrate that the number of test cases generated by their approach to obtain a error coverage is an order of magnitude less than those generated by traditional techniques. Koo and Mishra [35] proposed an efficient test case generation technique using decompositional model checking. They developed design and property decomposition procedures to generate the functional test cases for pipeline processors. However, these works focus on the validation of the pipelined processor architecture instead of general specifications. Adequacy is one of the most important factors in testing. Coverage ratio is widely used to determine the quality of test cases. In [39], Rayadurgam and Heimdahl presented a method for automatically generating test cases to meet structural coverage criteria. However, most of these methods are not directly applicable for UML activity diagrams.

To the best of our knowledge, there are no existing frameworks that can automatically derive test cases from UML activity diagrams.

3 Background

This section briefly describes model checking techniques and UML activity diagrams.

3.1 Model checking preliminaries

Model checking is a formal method that can enumerate all the state space to check whether a finite state system M satisfies a temporal property p, i.e., $M \models p$. When the property fails at some state, it will report a counterexample to falsify the specified property.

In this article, we focus on test case generation using *safety properties* (property asserts that the specified scenario never happens). Such properties are described using Linear Temporal Logic (LTL [40]). When we want to derive a test case to activate a desired functional scenario p, we need to generate its negation[1] $\sim F(p)$. Here $F(p)$ means *finally* the p will hold somewhere on the subsequent path. So $\sim F(p)$ asserts that property p can not be true in any computation paths of the model. If the design is correct, when applying the property $\sim F(p)$, a counterexample will be reported. The test case generated from the counterexample can be used to validate the specified functional scenario p.

Algorithm 1 outlines the general test case generation approach using SMV model checker. The inputs of this algorithm are a SMV model M and a set of assertions A. The output is a test suite extracted from the counterexamples. For each assertion A_i, one test case is generated. The algorithm iterates until all the assertions are checked. In each iteration, each assertion A_i is transformed to a temporal logic property P_i. Then, model checking is applied using the model M and negated property $\overline{P_i}$ to produce a counterexample (test case).

For complex designs and properties, BDDs based methods usually cause the state space explosion problem. As a promising alternative, Boolean satisfiability (SAT) based approaches have emerged, especially for the BMC. For SAT-based BMC, it translates the test generation problem into a Boolean formula. The test generation process is to figure out if the given Boolean formula has a satisfiable assignment. Usually this formula will be transformed into Conjunctive Normal Form (CNF) and checked by a SAT solver (e.g. GRASP [41] or Chaff [42]) to determine the result.

[1] In this article, both "$\sim$" and "$\neg$" denote the negation. "$\sim$" is used by SMV model checker, and "$\neg$" is used for general purpose Boolean formula.

Algorithm 1: *Test Case Generation using SMV*
Inputs: i) SMV Model M, ii) Set of assertions A
Outputs: Test suite
Begin
  TestSuite = ϕ;
  for each assertion A_i in the set A
    P_i = CreateProperty(A_i) ;
    $\overline{P_i}$ = Negate(P_i) ;
    $test_i$ = ModelChecking(M, $\overline{P_i}$) ;
    TestSuite = TestSuite $\cup$ $test_i$;
  endfor
  return TestSuite ;
End

3.2 UML activity diagram modeling

In this article, we adopt UML 2.1.2 [2] as our specification. To reduce the complexity of the testing work, we restrict our testing target and just investigate a subset of activity diagrams. The subset mainly contains action nodes, control nodes, object nodes and control and data flow. Especially for the object node, we just assume that it can hold at most one object at a time and it does not support *competition* and *data store*. This section first gives the notations used in our method. Then it presents the formal definitions of the UML activity diagrams. Finally, it proposes the coverage criteria that guide the test case generation.

3.2.1 Notations

UML activity diagram is used to coordinate the execution of actions. An action takes a set of inputs and converts them into corresponding outputs. As a kind of behaviors, an activity consists of a set of actions and flow edges. The actions are connected by object flow edges to show how object tokens flow through and connected by control flow edges to indicate the execution order.

UML activity diagrams adopt the semantics like Petri-net [6]. It uses a kind of directed graphs as its graphical representation. Tokens which indicate control or data values flow along the edges from the source node to the sink nodes driven by the actions and conditions. An activity diagram has two kinds of modeling elements: activity nodes and activity edges. More specially, there are three kinds of nodes in activity diagrams:

- *Action node*: Action nodes consume all input data/control tokens when they are ready, generate new tokens and send them to output activity edges.
- *Object node*: Object nodes provide and accept data tokens, and may act as buffers, collecting data tokens as they wait to move downstream.
- *Control node*: Control nodes route tokens through the graph. The control nodes include constructs to choose between alternative flows (decision/merge), to split or merge the flow for concurrent processing (fork/join).

Figure 1 shows the basic constructs of activity nodes. An action node is denoted with round cornered boxes. It represents an execution of operations on input tokens, and generated new tokens will be delivered to an outgoing edges. An object node denoted using

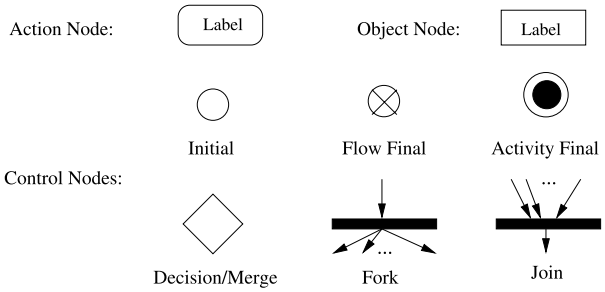

Fig. 1 UML activity nodes

rectangle boxes is used to temporarily hold the data tokens waiting to be processed or delivered. For simplicity, we assume object nodes do not support *competition* and *data store* for test case generation. A flow in an activity starts from the initial node. When a token arrives at a flow final node, it will be destroyed. The flow final node has no outgoing edges, so there is no downstream effect. When none of tokens exist in an activity diagram, the activity will be terminated. The activity final nodes are similar to flow final nodes, except that when a token reaches one activity final node, the entire flow will be terminated. Decision nodes and merge nodes use the same shape of diamond. Decision nodes choose one of the outgoing flows according to the value of Boolean expressions labeled on the outgoing edge. Merge nodes select only one of incoming flows to deliver to the next activity node. Forks or joins are shown by multiple arrows leaving or entering the synchronization bar respectively to describe the concurrent behavior of a system. When a token arrives at a fork node, it will be duplicated across the outgoing edges. Join nodes synchronize multiple flows. The tokens must be available on every incoming edge in order to be passed to outgoing edges.

Activity nodes are connected by activity edges along which tokens may flow under some condition. Activity edges include control and data flow edges as follows:

- *Control flow edge*: Control flow edges indicate the execution sequence of actions.
- *Object flow edge*: Object flow edges indicate the relation of data token transmission. It provides the inputs to actions.

In our method, we simplify the syntax and semantics of UML activity diagrams. We combine the control and data token together as a new kind of token which contains both control and data information. And such token can flow through activity edges. In other words, we do not distinguish control flow edges and object flow edges in our framework.

Figure 2 shows an example which uses most of the elements shown in Fig. 1. It describes the functionality of withdrawing money from an Automated Teller Machine (ATM) [43]. A user needs to enter the access code first. In case of failure, the user 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 can enter the amount of money he wants to withdraw. At the same time, the printer will be warmed to print a receipt. Once the ATM decides whether there is enough money the user can withdraw, it provides the cash and generates the information for this transaction. Finally, the printer prints the receipt and the transaction is complete.

The token for this example contains the ATM transaction information such as the input access code and input cash amount, the context information such as the available cash amount and correct access code. In general, a token reflects all the data information required for this activity. Table 1 shows the composition of a token of the ATM activity diagram. It consists of 5 variables which will be used to make the decisions illustrated in Table 2.

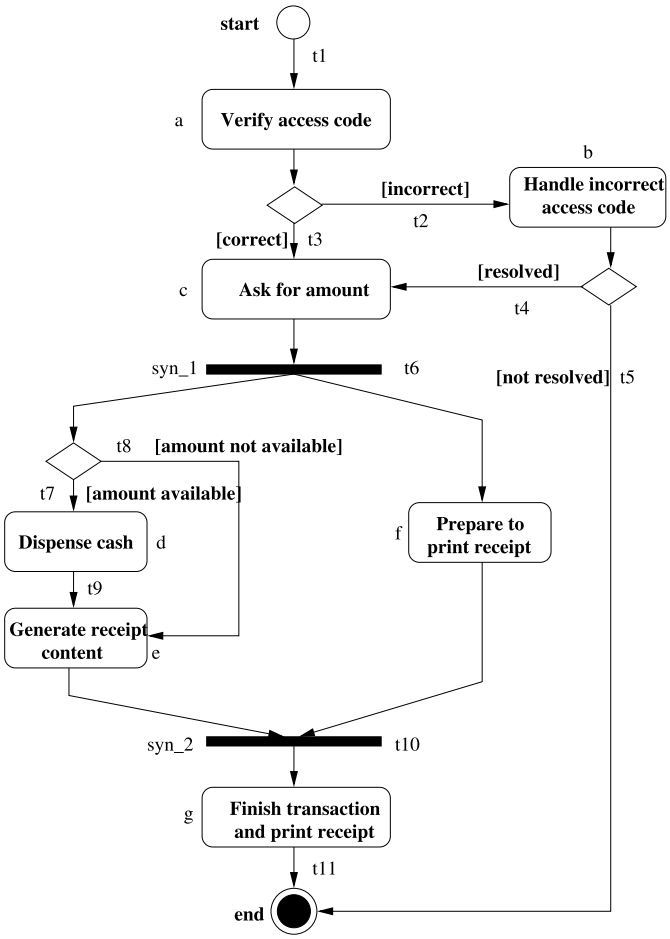

Fig. 2 The UML activity diagram of an ATM

Table 1 Break down of a token in Fig. 2

Variable	Type	Description
access_code	string	user's access code
access_code_input	string	user access code input
access_code_resolve	string	user access code input correction
amount_input	integer	user cash amount input
amount_available	integer	cash amount available

3.2.2 Formal definitions

Without formalism, it is hard to describe and model the activity diagrams accurately. UML activity diagram itself is a semi-formal specification that can not be directly mapped to a model checker input (e.g. SMV models). We use Petri-net as an intermediate formal model

Table 2 Condition on the flow edges in Fig. 2

Activity edge	Condition	Description
t2	incorrect	$access\_code! = access\_code\_input$
t3	correct	$access\_code = access\_code\_input$
t4	resolved	$access\_code = access\_code\_resolve$
t5	not resolved	$access\_code! = access\_code\_resolve$
t7	amount available	$amount\_input <= amount\_available$
t8	amount not available	$amount\_input > amount\_available$

between the translation from activity diagrams to SMV models, because the Petri-net formalism can capture the major functional scenarios as well as guide the translation.

Definition 1 describes the relation between the activity nodes and flow edges with a Petri-net semantics. It does not model the full features of activity diagrams and just formally depicts the static abstracted structure of activity diagrams which can be used to describe the scenarios that need to be tested.

Definition 1 An activity diagram is a directed graph described using eight-tuple $(A, T, F, C, V, A, a_I, a_F)$ where

- $A = \{a_1, a_2, \ldots, a_m\}$ is a set of action nodes.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a set of completion transitions.
- $F \subseteq \{A \times T\} \cup \{T \times A\}$ is a set of flow edges between activity nodes and completion transitions.
- $C = \{c_1, c_2, \ldots, c_n\}$ is a finite set of guard conditions. Here, c_i $(1 \leq i \leq n)$ is a predicate (expression) based on the input variables. There is a mapping from $f_i \in F$ to c_i, referred as $Cond(f_i) = c_i$.
- Let V be the set of all possible assignments for input variables $V_1, V_2, \ldots, V_k$ where k is a positive integer.
- $M : A \times V \rightarrow V$ is a mapping that describes the value change of the input variables inside an activity node.
- $a_I \in A$ is the *initial node*, and $a_F \in A$ is the *final node*. There is only one completion transition $t \in T$ and $c \in C$ such that $(a_I, t) \in F$, and for any $t' \in T$, $(t', a_I) \notin F$ and $(a_F, t') \notin F$.

In our formalization, a node can be an action node, an initial node or a final node. We use the *completion transition* and *flow edge* to model the behavior of the control nodes. In the graph, the nodes are connected by flow edges associated with a completion transition. Because activity diagrams allow tokens to exist in the flows concurrently, the completion transition can be used to synchronize the token flows. If a completion transition has multiple incoming flow edges, it will do the join operation. If there are multiple outgoing flow edges, then it will do the fork operation. For each flow edge, there may be a condition which can guide the token traverse. The graph just has one initial node that indicates the start of control and data flows. Activity diagrams have two kinds of final nodes: flow final nodes and activity final nodes. We combine them together and use a join operation to get a new activity final node. So in the definition there is only one final node.

When analyzing dynamic behaviors of an activity diagram, we need to use the *states* (a set of actions executing concurrently) to model the status of a system. Current state (denoted by *CS*) of an activity diagram indicates the actions which are being activated.

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

- ${}^\bullet t$ denotes the preset of t, then ${}^\bullet t = \{ a \mid (a, t) \in F\}$.
- $t^\bullet$ denotes the postset of t, then $t^\bullet = \{a \mid (t, a) \in F\}$.
- $enabled(CS)$ denotes the set of completion transitions that are associated with the outgoing flow edges of CS, then $enabled(CS) = \{t \mid {}^\bullet t \subseteq CS\}$.
- $firable(CS)$ denotes the set of transitions that can be fired from CS , then ($firable(CS) = \{t \mid t \in enabled(CS) \bigwedge {}^\bullet t$ *are all completed* $\bigwedge \exists n \in A.\ Cond((t, n))$ is satisfied $\bigwedge (CS - {}^\bullet t) \cap t^\bullet = \emptyset\}$. After some t is fired, the new current state $CS' = fire(CS, t) = (CS - {}^\bullet t) \cup t^\bullet$.

The current state of an activity diagram indicates which activity nodes are holding the tokens. For example, when $\{d, f\}$ is the current state of the activity diagram in Fig. 2, two tokens are in the activity nodes d and f individually. At this time, only the transition associated with t_9 is fireable. If it 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 fireable transitions in a state form a *concurrent transition*.

Definition 3 Let D be an activity diagram. For a state CS of D, a concurrent transition τ is a set of completion transitions $t_1, t_2, \ldots, t_n \in firable(CS)$ where

1. $\forall\, i, j (1 \le i < j \le n), {}^\bullet t_i \cap {}^\bullet t_j = \emptyset$;
2. $\forall\, t \in (enabled(CS) - \{t_1, t_2, \ldots, t_n\})$, 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. Because a path may have cycles, during the model checking, it is hard to determine the cycle numbers, so we neglect the cycles on a path. We call such path a *key path*.

Definition 4 A path ρ of the activity diagram D is a sequence of states and concurrent transitions, let

$$\rho = s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{n-1}} s_n$$

where $s_0 = \{a_I\}$, $s_n = \{a_F\}$, and $s_{i+1} = fire(s_i, \tau_i)$ for any i $(0 \le i < n)$. ρ is a *key path* if there is no state repetition in ρ, i.e. $\forall i, j$ $(0 < i < j \le n)$, $s_i \bigcap s_j = \emptyset$.

There are five key paths in Fig. 2:

- $\rho_1 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t5\}} \{end\}$
- $\rho_2 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$,
- $\rho_3 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t8\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$,
- $\rho_4 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$,
- $\rho_5 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t8\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$.

We insert a *dummy* node here because we assume that outgoing edges of the fork node must connect to an activity rather than a selection node. For a key path, when firing transitions, we need to consider guard conditions. For clarity, in Fig. 2, we did not label the condition guards for each transition.

In order to detect whether a concurrent state of an activity diagram is reachable or can be activated, we use the term *interaction*[2] to describe the scenario that a set of actions can be activated simultaneously. For example, in Fig. 2, $\{d, f\}$ is an example of "2-interaction" in the ATM.

Definition 5 Let D be an activity diagram. An interaction of the activity diagram is a set of activity nodes (actions) that can be activated simultaneously. A "k-interaction" is a set that contains k activity nodes.

3.3 Testing adequacy criteria for activity diagrams

Testing adequacy criterion specifies the requirement of a particular testing, and can be used as an objective measurement of the test case. In traditional software code testing, the definition of testing adequacy is given in [44] as a measurement function. The case of UML activity diagrams is different because it is in the form of model instead of code. Especially the coverage of activity diagram is more complex because of the concurrency. In this paper, we propose four types of coverage metrics as follows:

- *Activity coverage* requires that all the activity nodes in an 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 completion transitions in an 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 an 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.
- *Interaction coverage* requires that all the interactions in an activity diagram be covered. The value of *interaction coverage* is the ratio between the checked interactions and all the interactions in the activity diagram.

Generally, the above *specification level coverage criteria* do not directly relate to the *implementation level coverages* (classical code coverage) because of the abstraction. However, our experimental results (in Sect. 5) show that using test cases generated by specification level coverage can produce good implementation level coverage. The activity and transition coverage are used to check whether all the specified activities and transitions can be activated. Because an activity may be implemented by a module or a code segment, and a transition may be represented by a function call, their coverage can guarantee the function coverage and statement coverage in the corresponding implementation. The key path coverage enumerates all the possible token flows. Thus it can indicate the decision coverage and path coverage in its implementation. In activity diagrams, multiple tokens can stay in different activities at the same time, interaction coverage is proposed to investigate such

[2] Unlike the interaction in UML Interaction overview diagram, the interaction here means that several actions are active at the same time.

concurrent behavior of the system which can not be reflected in traditional sequential code coverage. However, interaction coverage is a superset of activity coverage and transition coverage. Therefore it can guarantee the statement coverage as well as function coverage in the implementation.

4 Test case generation for UML activity diagrams

There are many research work on Petri-net validation. However, very few of them can automatically generate test cases. UML activity diagrams adopts Petri-net like semantics. To automatically generate test cases, we resort to the model checker SMV which is based on Kripke structures. Therefore we need to use the Kripke constructs to mimic the behavior of activity diagrams (described in Sect. 4.1). However, direct translation from UML activity diagrams to SMV models is not feasible because activity diagrams are a semi-formal specification. In Sect. 3.2.2, we formalized both the syntax and semantics of UML activity diagrams in order to accurately capture the modeling information. Such information can be used as an intermediate formalism to guide the translation proposed in Sect. 4.1. Also in Sect. 3.3, we presented the test coverage criteria for UML activity diagram for testing adequacy. This information will be helpful for property generation described in Sect. 4.2.

Figure 3 shows our automated approach for coverage directed test case generation of UML activity diagrams. First, a UML activity diagram is translated to a formal model (SMV model) using the rules presented in Sect. 4.1. Next, the properties in the form of logic formulas can be generated using error models proposed in Sect. 4.2. Finally, the properties are checked using efficient model checking techniques (proposed in Sect. 4.3) to generate required test cases (counterexamples). The generate test cases can be applied for validation of both specifications and implementations. This methodology has three important tasks: formal model generation, property generation, and directed test case generation. The remainder of this section describes these tasks in detail.

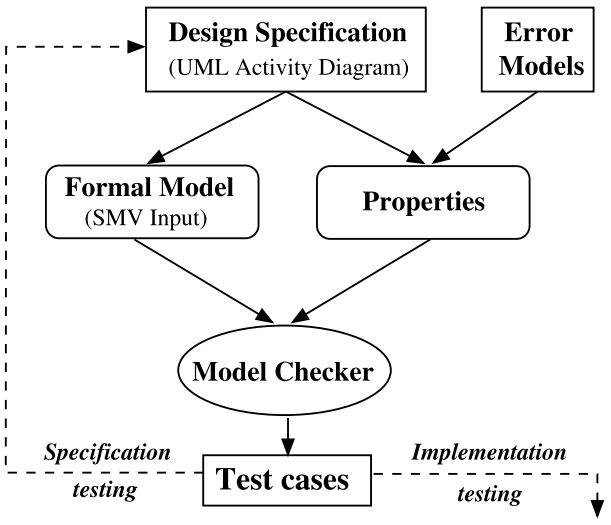

Fig. 3 The test case generation flow

4.1 Formal model generation

Our technique can extract both the control and data flows by parsing a UML activity diagram. The translation is a process of mapping from control and data flows to the input format of Cadence SMV model checker [45]. The translation consists of two parts: static information extraction and dynamic information extraction. Static information extraction analyzes the structure of an activity diagram and then generates a skeleton of the SMV input. The dynamic information extraction analyzes the dynamic behavior of the system by focusing on control and data flow analysis (i.e. the state change of activities, data manipulation in activities and the condition of the transitions).

4.1.1 Static information extraction

This step collects both the input data manipulated by the activities and the predicates used as guard conditions of the transitions. For example in Fig. 2, there are five input data variables that determine the data and control flows: *access_code*, *access_code_input*, *access_code_resolve*, *amount_input*, and *amount_available*. Because there may be a number of possible values for a variable, during model checking it will cause the state space explosion. In our approach, we adopt the model checker SMV which does not support complex data types (e.g. float, double and etc.). And for each variable, it is required that the value range should be specified explicitly. To avoid state space explosion, we use the following methods to reduce the complexity of data types:

– *Scaling*: Scaling is used to proportionally reduce the value range of a variable.
– *Reduction*: Reduction is used to reduce the cardinality of possible values for a variable.

Since it is hard to implement the above techniques automatically, before the SMV translation, the variable type information is tuned manually on activity diagrams.

In our translation, we assign each activity with a *state variable* which has three possible state values: *unvisited* (0), *visiting* (1) and *visited* (2). *Unvisited* indicates there is no token passed this activity node. *Visiting* indicates currently the activity is holding some tokens. And *visited* indicates some token passed this activity node and currently there is no token in this activity node. The extraction procedure instantiates the activity state variables and assigns suitable values to them. During initialization, the initial activity node is assigned *visiting* that means there is a token ready at the initial state. Other nodes are initialized to *unvisited*. Also we assign each flow edge a state variable which has two possible values: *fired* (1) and *unfired* (0). *Fired* means some tokens flowed from the incoming activity nodes to its outgoing activity nodes. *Unfired* means no token passed this activity edge. Initially we set them with value 0.

Figure 4 shows the generated skeleton of Fig. 2 in SMV format [29, 45]. There are 3 modules in this skeleton. The module *state* defines the token information (described in Table 1) as well as the state variable for activity nodes and flow edges. For example, *verify_access_code* is a state variable for an action with three states. Initially it is assigned the state *unvisited* (0). Module *ATM* gives a static skeleton without dynamic information (state transition for state variables of activity nodes and flow edges, and the value changes for date variables in tokens). In this phase, we just collect variables without any processing. The missing state transition details will be described in Sect. 4.1.2. The module *main* creates the module instances and elaborates them together. For example, *st* is an instance of state module and *atm* is an instance of ATM module. We bind the *st* and *atm* together, because *atm* will handle the state changes of variables in *st*.

Fig. 4 The generated skeleton after structure extraction

```
MODULE state
 VAR
   access_code: { A1, B1, C1 };
   access_code_input: { A1, B1, C1, D1 };
   start: 0..2;
   syn_1: 0..2;
   verify_access_code: 0..2;
   t2_cond: 0..1;
   t3_cond: 0..1;
   ......
 ASSIGN
   init(start):=1;
   init(syn_1):=0;
   init(verify_access_code):=0;
   ......
MODULE ATM(st)
 ASSIGN
   next(st.start):=
   next(st.t2_cond):=
   ......
   next(st.prepare_print_receipt):=
   ......
   next(st.dispense_cash):=
   next(st.t7_cond):=
   ......
MODULE main() {
   st: state;  atm: ATM(st);
   p_print:  prepare_print(st);
   check:  check_amount(st);
}
```

4.1.2 Dynamic information extraction

After the static information extraction, we need to extract both the data manipulations and the transitions of state variables, because they will determine the data and control flows.

In our method, we define a set of rules that specify the state transition for each activity node and the value changes of the each data. Figure 5 shows the details of the rules. In these rules, we use the preset and postset notations. In these rules, the assignment and constraint to a set means the assignment and constraint to each element in the set. For example, if ${}^\bullet t = \{a_1, a_2, \ldots, a_k\}$, then ${}^\bullet t = 1$ means $a_1 = 1 \,\&a_2 = 1 \,\& \ldots \,\& a_k = 1$ and $cond(t)$ means $cond((a_1, t)) \,\& cond((a_2, t)) \,\& \ldots \,\& cond((a_k, t))$.

Rule 1 specifies the translation rule for the initial node. The token will be first put at the initial state and the node is marked as *visited* in the next step. Rule 2 specifies the translation rule for the final node. At first, the state is *unvisited*, when one of the incoming edges is activated, its state will become *visited*. Rule 3 defines the state changes of an activity. Initially, the state of an activity is *unvisited*. If the incoming edge is activated, the state will become *visiting* in the next step. If the current state is *visiting*, the state will change to *visited* in the next step. Rule 4 presents the state transition of the fork nodes. When the incoming edge is activated, the fork node will maintain the *visiting* status until all the outgoing edges are visiting or visited. Rule 5 gives the state transition of join nodes. The join node is used to synchronize the token flows. When all the incoming flows are ready, the transition corresponding to the join node can be fired. In this rule, if we want to fire the transition, we need to wait until all the activity nodes in the preset of the transition are visited. Rule 6 shows how to manipulate the state change of the transition when it is fired. Rule 7 presents the

Rule 1: If n is an initial node
$init(n) := 1;$
$next(n) := 2;$

Rule 2: If n is a final node, and there are k incoming transitions $t_1, t_2, \ldots, t_k$.
$init(n) := 0;$
$next(n) :=$
$case$
$((^\bullet t_1 = 1 \ \& \ cond(t_1)) \mid (^\bullet t_2 = 1 \ \& \ cond(t_2)) \mid$
$\ldots \mid (^\bullet t_k = 1 \ \& \ cond(t_k))) : 2;$
$1 : n;$
$esac;$

Rule 3: If n is an activity node (not join or fork), and there are k incoming transitions $t_1, t_2 \ldots, t_k$.
$init(n) := 0;$
$next(n) :=$
$case$
$n = 1 : 2;$
$(^\bullet t_1 = 1 \ \& \ cond(t_1)) \mid (^\bullet t_2 = 1 \ \& \ cond(t_2)) \mid$
$\ldots \mid (^\bullet t_k = 1 \ \& \ cond(t_k))) : 1;$
$1 : n;$
$esac;$

Rule 4: If n is a fork node, and the corresponding transition is t.
$init(n) := 0;$
$next(n) :=$
$case$
$n = 1 \ \& \ t^\bullet > 0 : 2;$
$^\bullet t = 1 : 1;$
$1 : n;$
$esac;$

Rule 5: If n is a join node corresponding to transition t, and $a_1, a_2, \ldots, a_k$ are k elements of $^\bullet t$.
$init(n) := 0;$
$next(n) :=$
$case$
$n = 1 : 2;$
$n = 0 \ \& \ (a_1 + a_2 + \cdots + a_k = 2 * k) : 1;$
$n = 2 \ \& \ (a_1 + a_2 + \cdots + a_k < 2 * k) : 0;$
$1 : n;$
$esac;$

Rule 6: If t is a transition which corresponds to the flow edges.
$init(t) := 0;$
$next(t) :=$
$case$
$! \ cond(t) \ \& \ ^\bullet t = 1 : 0;$
$cond(t) \ \& \ ^\bullet t = 1 : 1;$
$1 : t;$
$esac;$

Rule 7: If v is a variable whose new value is changed by expression exp_i in the activity act_i $(1 \le i \le n)$.
$next(v) :=$
$case$
$act_1 = 1 : exp_1;$
$act_2 = 1 : exp_2;$
$\ldots\ldots$
$act_n = 1 : exp_n;$
$1 : v;$
$esac;$

Fig. 5 Translation rules for state and data transitions

translation for value change of the variables. If an activity performs some operation on the variable, we can modify the value of the variable only when the activity state is *visiting*.

4.2 Property generation based on error models

When generating directed test cases, it is required to use coverage metrics to indicate the sufficiency. Based on the definition in Sect. 3.3, we create the *error models* which are the negation of the coverage requirements. An error model defines a set of errors for an arbitrary design. Each error described by the model represents a set of potential errors in a design. The validation of all errors can guarantee the detection of all errors of the type covered by the error model. The error models in this article are as follows.

Definition 6 Let *AD* be an activity diagram, there are four error models:

- *Activity error model*. For each activity of *AD*, the model assumes that such activity is not reachable.
- *Transition error model*. For each transition of *AD*, the model assumes that such transition can not be fired.
- *Key path error model*. For every key path of *AD*, there is no corresponding executable *path*.
- *Interaction error model*. For every interaction of *AD*, the activities associated with the interaction cannot be activated at the same time.

From these four different models, we can generate various properties to validate the activity diagram. The activity error 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 error 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. We also need to check all the dynamic behaviors of the system, so key path error model is preferable in this case. The interaction error model can be used to check whether several activities can be activated simultaneously. In general, if all the interactions have only one activity, the interaction error model is same as the activity error model.

The transformation from an error to a property (of the formal model) is a one-to-one mapping. In the activity error model, for each activity we can assert that it is not reachable. For transition error model, we assert that such transition can not be fired. In the key path error model, we assert that there is no single execution scenario that visited all the activities and transitions of the key path. For interaction error model, we assert that the concurrent state is not reachable. Figure 6 presents illustrative examples of the four error models. Because an error is described in a negated form of a desired system behavior, the generated properties are the final version properties which need to be checked.

4.3 Efficient test case generation techniques

This section presents two efficient test case generation techniques that can drastically reduce the overall test case generation time. The first technique uses unbounded model checking for test case generation. We have developed efficient design and property decomposition techniques to reduce the test generation time. The second technique employs SAT-based bounded model checking. We have developed a procedure for determining the bound of a SAT instance in the context of test case generation. This section is organized as follows.

```
Property 1:The activity dispense_cash is
  not reachable.
  LTL formula: ~ F (st.dispense_cash = 2)
Property 2:The transition with condition
  [amount available] can not be fired.
  LTL formula: ~ F( st.t7_cond = 1 )
Property 3:The key path 4 can not be covered.
  LTL formula:
   ~F ( st.start = 2
      & st.verify_access_code=2
      & st.handle_access_code = 2
      & st.ask_for_amount = 2
      & st.prepare_print_receipt = 2
      & st.dispense_cash = 2
      & st.generate_receipt_content=2
      & st.finish_transaction_print_receipt = 2
      & st.end = 2 & st.t2_cond=1
      & st.t4_cond=1 & st.t7_cond=1 )
Property 4:The activities dispense_cash and
  prepare_to_print_receipt can not be activated
  simultaneously.
  LTL formula: ~ F( st.dispense_cash = 1
                  & st.prepare_to_print_receipt =1)
```

Fig. 6 Error model examples

Sect. 4.3.1 describes our model checking based test case generation approach using decomposition techniques. Sect. 4.3.2 presents our bound determination technique and test case generation approach using SAT-based bounded model checking.

4.3.1 Test case generation using property decompositions

By using Algorithm 1 with the formal model translated from the UML activity diagram (described in Sect. 4.1) and properties derived from the error models (described in Sect. 4.2) as inputs. We can get the a set of test cases for validation both specification and implementation. Since the whole design is used during model checking, this approach is limited by the capacity restrictions of the model checking tool. As a result, this approach is not suitable for today's complex UML activity diagrams since the time and memory requirements can be prohibitively large in many test case generation scenarios. In fact, test case generation may not be possible in various instances due to state space explosion. We propose a test case generation approach using decomposition of the properties to make the automated test case generation applicable in practice.

Property decomposition technique has been successfully used in the processor verification domain [35]. The basic idea is to break one system level property into multiple component level properties and apply them to the respective components. Checking the system with a simple property usually need less time and space than using a complex property. So this technique is promising when dealing with a large system and complex properties. However, not all properties are decomposable and in some circumstances decompositions need the human intervention. So generally this technique can be used as a complement of traditional model checking and SAT-based bounded model checking.

It is important to note that when decomposing a cluster of similar properties, the properties textually and semantically share a large overlap. That means some decomposed part of a property can be the same as the decomposed part of other properties. So the verification

effort of this decomposed property can be reused by other properties. In general, the overall test case generation time of this cluster can be drastically reduced.

A complex property consists of various combinations of temporal operators, Boolean connectives and atomic predicates. If the complex property is decomposable, a set of simple properties will be derived with the same or similar semantics as the complex property. Since the simple properties just describe partial behavior of the system, for some decomposable complex property, it is necessary to merge the partial counterexamples of the derived properties. If the partial counterexamples generated by the decomposed properties can be refined to reason about the complex property, the property is *decomposable*.

Definition 7 Let P be a false property for the design, and P is *decomposable* in the form $p_1 \wedge p_2 \wedge \cdots \wedge p_n$ or in the form $p_1 \vee p_2 \vee \cdots \vee p_n$ if all the following conditions are satisfied.

- If the decomposed properties are in the form $p_1 \wedge p_2 \wedge \cdots \wedge p_n$, then at least one property p_i $(1 \leq i \leq n)$ has a counterexample.
- If the decomposed properties are in the form $p_1 \vee p_2 \vee \cdots \vee p_n$, then each properties p_i $(1 \leq i \leq n)$ has a counterexample.
- The counterexamples generated from properties p_i $(1 \leq i \leq n)$ can be refined to a counterexample of property P.

Since in this article we are only interested in the test case generation for different error models, the generated properties are negations of the required system behavior. So at first we need to negate the property. The negation of the properties can be done using the following rules:

$$\begin{aligned} \neg G(p) &\equiv F(\neg p) \\ \neg F(p) &\equiv G(\neg p), \end{aligned} \tag{1}$$

For example, the safety property generated by the UML error model in the form $\neg F(p_1 \wedge p_2 \wedge \cdots \wedge p_n)$, according to (1), is equivalent $G(\neg p_1 \vee \neg p_2 \vee \cdots \vee \neg p_n)$.

Decomposable properties

A LTL property consists of temporal operators (G, F, X, U) and Boolean connectives ($\wedge$, $\vee$, $\neg$ and $\rightarrow$). In (2), according to the semantics of the temporal logic, the left part and the right part of the following four equations are equivalent.

$$\begin{aligned} G(p \wedge q) &\equiv G(p) \wedge G(q) \\ F(p \vee q) &\equiv F(p) \vee F(q) \end{aligned} \tag{2}$$

In fact, these equations can be used to perform the property decomposition and counterexample generation. For example, the counterexamples generated by $G(p)$ and $G(q)$ can be used as counterexample of $G(p \wedge q)$. Intuitively if a path can not satisfy the property p or q, then it can never satisfy the property $p \wedge q$.

There are many scenarios where decompositions are not possible. For example, the following two decompositions are not allowed.

$$\begin{aligned} F(p \wedge q) &\neq F(p) \wedge F(q) \\ G(p \vee q) &\neq G(p) \vee G(q) \end{aligned} \tag{3}$$

Although they are not semantically equivalent, they are decomposable under certain restrictions. For $F(p \wedge q) \neq F(p) \wedge F(q)$, the counterexamples generated from the property $F(P)$ or $F(q)$ can be used as the counterexamples for property $F(p \wedge q)$. For the second equation, according to Definition 7, it is hard to guarantee if $G(p \vee q)$ has a counterexample, then each property of $G(p)$ and $G(q)$ has a counterexample. However, when introducing the notion of clock (or steps), it can be decomposed. The proof is as follows.

Theorem 1 *The counterexamples of properties $G(p)$ and $G(q)$ can be used to derive a counterexample for property $G(p \vee q)$ by introducing the notion of clock (or step) as a synchronization mechanism.*

Proof By using the negation in (1), $\neg G(p \vee q \vee clk \neq t) \equiv F(\neg p \wedge \neg q \wedge clk = t)$. When introducing the clock into the properties and the clock in the property is set to the value t, then $F(\neg p \wedge \neg q \wedge clk = t)$ means that at clock t, both p and q will be violated. That means $F(\neg p \wedge \neg q \wedge clk = t) \equiv F(\neg p \wedge clk = t) \wedge F(\neg q \wedge clk = t)$. So $\neg G(p \vee q \vee clk \neq t) \equiv F(\neg p \wedge clk = t) \wedge F(\neg q \wedge clk = t)$. By using the negation again on both sides, we can get $G(p \vee q \vee clk \neq t) \equiv G(p \vee clk \neq t) \vee G(q \vee clk \neq t)$. Let C_p be the set of all possible counterexamples generated by property p. We can get $C_{G(p \vee q \vee clk \neq t)} = C_{G(q \vee clk \neq t)} \cap C_{G(p \vee clk \neq t)}$. So property $G(p \vee q \vee clk \neq t)$ is decomposable. □

Theorem 1 shows that when dealing with a property with a clock predicate, if the property is in the form of $G(p \vee q \vee clk \neq t)$, then the property is decomposable. We can check the properties $G(p \vee clk \neq t)$ and $G(q \vee clk \neq t)$ respectively, and combine the counterexamples together to achieve the final test case.

4.3.2 SAT based bounded model checking

Algorithm 2 describes the general test case generation algorithm using BMC. This algorithm takes the model M generated from UML activity diagrams and properties derived form the errors F as inputs and generates a test suite extracted from the counterexamples. For each error E_i, one test case is generated. The iteration stops until all the specified errors are checked. In each iteration, each error E_i is described using a temporal logic property P_i.

Algorithm 2: *Test Case Generation using BMC*
Inputs: i) UML Activity Diagram Model, M
ii) Set of errors E (based on coverage criteria)
Outputs: Test suite
Begin
TestSuite = ϕ;
for each error E_i in the set E
P_i = CreateProperty(E_i) ;
$bound_i$ = DetermineBound(M, P_i) ;
$test_i$ = ModelChecking(P_i, M, $bound_i$) ;
TestSuite = TestSuite ∪ $test_i$;
endfor
return TestSuite ;
End

After the bound k_i of P_i is determined, SAT-based BMC will generate a counterexample (test case) to falsify P_i.

Generally BMC is more efficient than unbounded model checking since it restricts its search within a small range, called bound. SAT-based BMC translates the search problem into a propositional formula. The test case generation in fact is a process to figure out a satisfiable assignment for this formula which can be converted into a test case. During the test case generation, bound determination plays an important role. If it can be known a priori, SAT-based BMC can be more effective than BDD based model checking techniques [11]. However, any incorrect bound determination will increase test case generation time as well as memory requirement. Therefore, the techniques of deciding property bounds determine the efficiency of test case generation using SAT-based BMC. In this article, according to the structure information of UML activity diagrams, we propose a method that can efficiently estimate the bound for each property in practice.

Determination of bound

Biere et al. [11] described several ways to determine the bound. If $M \models_k \mathbf{E}f$ for all k within the bound, then $M \models \mathbf{E}f$. However there is no deterministic way of computing the bound of the system. In fact, determining the minimal bound for a property is as hard as the model checking itself. So bounded model checking is promising only when the bound can be predetermined and is shallow.

According to the definition of the diameter in [11], the bound for each node error instance is decided by the temporal distance between the root node and the node under verification. Generally, the bound for the key path error is determined by the activities and transitions along the path. For example, in Fig. 2, the length of the key path $\rho_4 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$ is 9. The property derived for this key path is shown in the Fig. 6. In our translation rules, an activity state transition needs one step delay. Fork node needs one step delay, and join node needs two steps delay. One step delay at the start node is also required. The bound size will be $9 + 1 + 2 + 1 = 13$. The bound of the activity error or transition error is determined by the delay of activities and transitions on a valid shortest path from the *start* node to the activity or transition which need to be verified in the UML activity diagram. For example, when we want to check the activity error model instance "*prepare_to_print_receipt* can not be activated", the system will generate the property $\sim F(st.prepare\_print\_receipt = 2)$. The shortest path from start to such an activity is $\rho = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\}$. In a similar way, the bound for this property is $4 + 1 + 1 = 6$. Sometimes in the system, there is a counter that acts like a clock which counts the execution steps. Such variable in a property will affect the bound of the property. For example, because of the introduction of a counter, the property $\sim F(clk = 10 \ \& \ st.prepare\_print\_receipt = 2)$ has a bound of 10 instead of 6.

Different properties based on different error models have different methods to achieve the bounds. Assume there is no counter variables, the determination of bound is according to the following rules:

- Activity or transition property. Extract all the paths without loops from the initial node to the activity or transition. Calculate the bound for each extracted path and choose the shortest one as the property bound.
- Key path property. Calculate the bounds for the key path based on the delay of activities and transitions on the key path.

– Interaction property. Calculate the bound for each element (activity or transition) in the interaction. Choose the largest bound as the property bound.

If a property contains a counter variable. Then bound of the property is the larger one between the counter value and the bound calculated using the above rules. Therefore, the complexity of bound determination is polynomial to the nodes in the UML activity diagram. In general, it is more efficient to use BMC for shallow counterexamples because the bound can be pre-determined.

5 Experiments

Based on the framework proposed in Sect. 4, we developed a prototype tool which can automate the process of test case generation. The tool takes three inputs: type definition of the data which is used in the activity diagram, the context information which set the parameters for the execution of an activity diagram (e.g. when to trigger the initial node and so on), and UML activity diagrams. The UML activity diagrams are stored in the format of XML Metadata Interchange (XMI) files. The tool can parse the XMI files to get the static and dynamic information for formal model translation. Combined with the context information and data type information, a formal model can be achieved using the proposed mapping rules. From the extracted information, we can get the structure of the graph. Based on this graph we can generate the properties according to specific error models. When using BMC, the estimated bounds can be derived along with the properties. Test cases (a sequence of variable assignments) for activity diagrams can be obtained from the model checking counterexamples. These system level test cases represent assignments to the primary inputs of specifications. So they can be used for specification simulation. In addition, such test cases can be reused for low level implementation validation. Like the method proposed in [32], we developed a script which incorporates a set of transformation rules (e.g. variable mappings and value mappings) for the test case conversion.

In this section, we present three detailed case studies. We compare our approach with the random test case based method [24], which is the best known result in this category. The experimental results demonstrate that our method can drastically reduce both test case generation time and overall validation effort by producing high quality test cases for the implementation. We applied Cadence SMV model checker in our study. All the experiments were conducted using 2.0 GHz Intel Core2 Duo CPU with 1 GB RAM.

5.1 A control system

The first case study is a small control system. The UML activity diagram representation of the control system consists of 17 activities, 23 transitions and 6 key paths. Table 3 shows the comparison between our approach and the random test based method [24]. For generating test cases with highest coverage, the random method requires 8.83 seconds to run the 150 random test cases, however our approach using unbounded model checking method (UMC) just needs 0.91 seconds. In this case study, UMC approach improves the test case generation time by an order of magnitude.

We applied the generated test cases to a simulator of the control system (implemented using JAVA). Table 4 shows the coverage of the Java code. The generated test cases obtained 100% package as well as class coverage. However, the *method*, *block* and *line* coverage are 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 test cases did not activate any of the exception blocks which 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 test cases 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) [24] is to process three 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.

In Table 5, the first three rows depict the results by using 800, 1000, 1500 random test cases respectively. The result by our method is shown in the last two rows. In the case of *random* 800, two key paths are missing due to the randomness. So the coverage metrics are not 100%. If we increase the number of the random test cases 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 number of required random test cases. As a result, it is hard to discover whether the specification is correct by the random test cases. The result of the UMC shows that we can get an order of magnitude improvement compared to the random method. Because the bounds of the properties of OSES system are shallow and can be pre-determined, we applied SAT-based bounded model checking (BMC) in this situation. The result shows that BMC method can be an order of magnitude faster than UMC method. Clearly, BMC approach reduces the validation effort by two hundred times compared to the best known result [24] in this category.

Table 6 presents the coverage of the implementation by applying the generated test cases. The coverage of *method*, *block* and *line* are not sufficient because the activity diagram does not consider all the scenarios of the system, such as the registration of the customers and so on. In this case, we need to add the missing details in the specification to obtain the required coverage.

Table 3 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 approach (UMC)	100	100	100	0.91

Table 4 Implementation level coverage of the control system

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

Table 5 Comparison of three 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 approach (UMC)	100	100	100	3.47
Our approach (BMC)	100	100	100	0.15

Table 6 Implementation level coverage of OSES

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

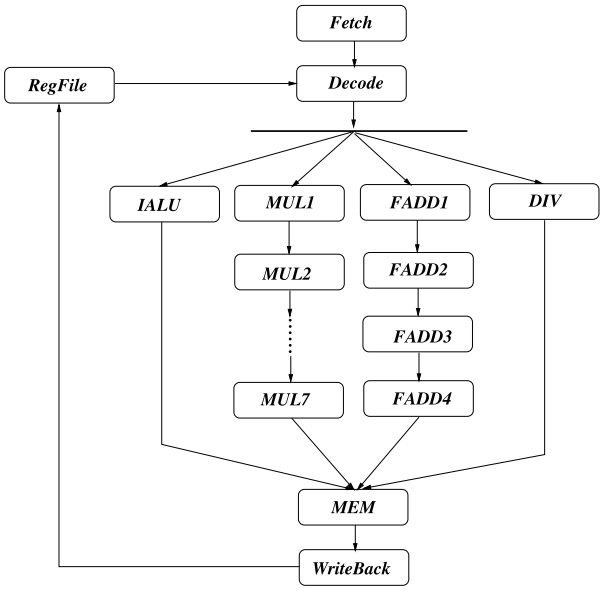

Fig. 7 The activity diagram for a MIPS processor

5.3 A MIPS processor

Petri-net is promising for modeling generic processors. Based on the method proposed in [46], we created a UML activity model for the single-issue MIPS design presented in [47]. Figure 7 shows the activity diagram for the MIPS processor design. It has five pipeline stages: fetch, decode, execute, memory (MEM), and writeback. The execute stage has four parallel execution paths: integer ALU, 7-stage multiplier (MUL1–MUL7), 4-stage floating-point adder (FADD1–FADD4), and multi-cycle divider (DIV). In total there are 13 execution units.

In order to compare the efficient test case generation methods proposed in Sect. 4.3, we applied the following four test case generation techniques on a MIPS process design:

Table 7 MIPS processor result using the four methods

Techniques	Bound	Number of properties	Average memory requirement	Time (seconds)
UMC	–	75	BDDs: 10121528	4929.75
DUMC	–	13	BDDs: 2343270	94.77
BMC	10	75	Clauses: 19203	8.25
DBMC	10	13	Clauses: 16340	1.17

– *UMC*: Unbounded model checking using BDDs.
– *BMC*: SAT-based bounded model checking.
– *DUMC*: Decompose the properties and then use UMC.
– *DBMC*: Decompose the properties and then use BMC.

In this case study, due to the parallelism in the execute stage, we applied the interaction error model to validate the concurrent behaviors of the MIPS processor model. A 2-interaction of the execution units indicates that the two specified execute units can be *busy* at the same clock cycle. For example, the following property asserts that at clock cycle 10, the ALU unit and the DIV unit can not be busy simultaneously. Thus the generated test case (a sequence of instructions) from this property can be used to activate both execution units at clock cycle 10.

```
assert ~F( alu_active = 1 & div_active = 1 & clock = 10);
```

Based on the 2-interaction error model, 78 properties generated (choose all pairs from 13 execution units). However, due to the single issue design, some interactions are invalid. For example, ALU unit and MUL1 unit can not be activated at the same time. Therefore finally we generated 75 properties.

Table 7 presents the validation result of the MIPS processor design applying the four test case generation techniques (first column). Due to the introduction of the clock, the bounds of the BMC and DBMC techniques are pre-determined and shown in the second column. The third column shows the number of properties that need to be checked. The fourth column describes the average memory requirement by the respective technique. When unbounded model checking is used (UMC or DUMC), the number of BDD nodes is reported. However, when SAT-based bounded model checking (BMC or DBMC) is used, we report the number of CNF clauses. The last column presents the total test case generation time. The result shows that our decomposition method can drastically reduce both the test case generation time and the memory requirement. It is important to note that in DUMC and DBMC, we reuse the validation effort of the decomposed sub-properties to improve test case generation time. For the MIPS processor design using DUMC and DBMC, generating 75 test cases just need to check 13 properties. Because all the 75 test cases can be composed by the counterexamples of these 13 properties.

5.4 Summary

This section presents three case studies. The first one shows that automatic test generation using BMC techniques for small systems can achieve the required specification coverage quicker than random simulation methods. The second example compares the random simulation method, UMC based and BMC based techniques. It shows that model checking based

methods are promising. Especially that BMC performs better than UMC. For these two case studies, we apply the test cased obtained at specification level to the corresponding implementations. The results shows that the high level validation efforts can be applied on the low level implementation, and the implementation coverage is satisfactory. In addition, the generated test cases can be used to check the consistency between different abstraction levels. The final case study compares four different kinds of model checking based techniques. It shows that decomposition and SAT bases techniques can drastically reduce the test generation time.

In practice, the four model checking based techniques have both merits and limits. For UMC, it can be implement fully automatically. However, due to state space explosion problem, it can not be applied to large scale models. So in general, BMC is a promising alternative for UMC. The decomposition based techniques has two advantages: First, it can trigger some reduction techniques such as Cone of Influence (COI) to reduce the validation complexity. Second, the validation effort reusing of the same decomposed properties can reduce the overall test generation time. By our observation, DUMC can benefit from both complexity reduction and validation reusing. But DBMC can just benefit from the validation reusing, because the validation complexity of the complex property and decomposed complexities may differ slightly. It is important to note that the decomposition sometimes can not be fully automated, because it is hard for a tool to parse the semantics of an activity diagram and figure out the dependence of components. So the human intervention is needed when necessary.

6 Conclusions

It is widely acknowledged that automatic test case generation from high level specifications can have double impact: i) the generated test cases can be used to verify both the specification and the implementation, and ii) it can drastically reduce the overall validation effort. However, due to lack of comprehensive error models and associated test case generation techniques, it is not possible to automatically generate directed test cases to activate all the interesting scenarios and corner cases in the UML specification. This paper proposed an approach to automatically generate test cases from UML activity diagrams. This paper made three important contributions. First, we proposed a set of testing-oriented translation rules that can automatically convert the UML activity diagram to the input specification of the model checker. Second, we presented a method to derive properties from specifications based on error models. Finally, we incorporate several existing promising model checking based techniques into our framework to improve the test generation effort for UML activity diagrams. Our experimental results demonstrate that our approach can reduce both test case generation time and overall validation effort by several orders-of-magnitude without sacrificing the functional coverage goal.

Acknowledgements This work was partially supported by grants from Intel Corporation and National Science Foundation Faculty Early Career Development (CAREER) Award 0746261. A preliminary version [7] of this paper appeared in the proceedings of ACM Great Lakes Symposium in VLSI (GLSVLSI) 2008.

References

1. Rumbaugh J, Jacobson I, Booch G (2001) The unified modeling language user guide. Addison-Wesley, Boston
2. OMG (2007) UML superstructure V2.1.2. Available at http://www.omg.org/docs/formal/07-11-02.pdf

3. Martin G (2002) UML for embedded systems specification and design: motivation and overview. In: Design automation and test in Europe 2005, pp 773–775
4. Martin G, Müller W (2005) UML for SOC design. Springer, Berlin
5. Müller W, Rosti A, Bocchio S, Riccobene E, Scandurra P, Dehaene W, Vanderperren Y (2006) UML for ESL design–basic principles, tools, and applications. In: International conference on computer-aided design 2006, pp 73–80
6. Peterson J (1981) Petri nets theory and the modeling of systems. Prentice-Hall, New York
7. Chen M, Mishra P, Kalita D (2008) Coverage-driven automatic test generation for UML activity diagrams. In: Proceedings of Great Lakes symposium of VLSI (GLSVLSI), pp 139–142
8. Ammann P, Black P, Majurski W (1998) Using model checking to generate tests from specifications. In: Proceedings of international conference on formal engineering methods (ICFEM), pp 46–54
9. Bryant RE (1986) Graph-based algorithms for Boolean function manipulations. IEEE Trans Comput 35(8):677–691
10. Prasad M, Biere A, Gupta A (2005) A survey of recent advances in SAT-based formal verification. Int J Softw Tools Technol Transf 7(2):156–173
11. Biere A, Cimatti A, Clarke E, Zhu Y (1999) Symbolic model checking without BDDs. In: International conference on tools and algorithms for the construction and analysis of systems, pp 193–207
12. Riccobene E, Scandurra P, Rosti A, Bocchio S (2005)A SoC design methodology involving a UML 2.0 profile for SystemC. In: Design automation and test in Europe, pp 704–709
13. Müller W, Rosti A, Bocchio S, Riccobene E, Scandurra P, Dehaene W, Vanderperren Y (2006) UML for ESL design: basic principles, tools, and applications. In: International conference on computer aided design, pp 73–80
14. Rosenberg D, Mancerella S (2010) Embedded systems development using SysML: an illustrated example using enterprise architect. Sparx Systems Pty Ltd and ICONIX
15. Lamberg K (2007) Trends and perspectives in automated ECU testing. In: Automotive electronic, June, 2007
16. dSPACE. http://www.dspace.com
17. Lavagno L, Müller W (2006) UML as a next-generation language for SoC design. In: Electronic design
18. Heckel R, Lohmann M (2003) Towards model-driven testing. In: International workshop on test and analysis of component based systems, pp 284–291
19. Briand LC, Labiche Y (2002) A UML-based approach to system testing. Softw System Model 1(1):10–42
20. Chen T, Poon P, Tang S, Tse T (2005) Identification of categories and choices in activity diagrams. In: International conference on software quality 2005, pp 55–63
21. Wang L, Yuan J, Yu X, Hu J, Li X, Zheng G (2004) Generating test cases from UML activity diagram based on gray-box method. In: Asia-pacific software engineering conference 2004, pp 284–291
22. Kim H, Kang S, Baik J, Ko I (2007) Test cases generation from UML activity diagrams. In: Software engineering, artificial intelligence, networking, and parallel/distributed computing 2007, pp 556–561
23. Chen M, Qiu X, Li X (2006) Automatic test case generation for UML activity diagrams. In: International workshop on automation on software test 2006, pp 2–8
24. Chen M, Qiu X, Xu W, Wang L, Zhao J, Li X (2009) UML activity diagram based automatic test case generation for Java programs. Comput J 52(5):545–556
25. Unhelkar B (2005) Verification and validation for quality of UML 2.0 models. Wiley, New York
26. Bell A, Haverkort BR (2005) Sequential and distributed model checking of Petri nets. Int J Softw Tools Technol Transf 7(1):43–60
27. Jensen K, Kristensen LM, Wells L (2007) Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. Int J Softw Tools Technol Transf 9(3–4):213–254
28. Eshuis R (2006) Symbolic model checking of UML activity diagrams. ACM Trans Softw Eng Methodol 15(1):1–38
29. Cimatti A, Clarke EM, Giunchiglia F, Roveri M (1999) NUSMV: a new symbolic model verifier. In: International conference on computer aided verification 1999, pp 495–499
30. Guelfi N, Mammar A (2005) A formal semantics of timed activity diagrams and its PROMELA translation. In: Asia-Pacific software engineering conference 2005, pp 283–290
31. Das D, Kumar R, Chakrabarti PP (2006) Timing verification of UML activity diagram based code block level models for real time multiprocessor system-on-chip applications. In: Asia-Pacific software engineering conference 2006, pp 199–208
32. Chen M, Mishra P, Kalita D (2007) RTL towards test generation from systemc TLM specifications. In: High level design validation and test workshop 2007, pp 91–96
33. Fraser G, Wotawa F (2007) Improving model-checkers for software testing. In: International conference on software quality 2007, pp 25–31

34. Mishra P, Dutt N (2005) Functional coverage driven test generation for validation of pipelined processors. In: Design automation and test in Europe 2005, pp 678–683
35. Koo H, Mishra P (2006) Functional test generation using property decompositions for validation of pipelined processors. In: Design automation and test in Europe, pp 1240–1245
36. Mishra P, Chen M (2009) Efficient techniques for directed test generation using incremental satisfiability. In: International conference on VLSI design, pp 65–70
37. Chen M, Qin X, Mishra P (2010) Efficient decision ordering techniques for SAT-based test generation. In: Design, automation and test in Europe, pp 490–495
38. Chen M, Mishra P (2010) Functional test generation using efficient property clustering and learning techniques. IEEE Tran Comput-Aided Des Integr Circuits Syst 29(3):396–404
39. Rayadurgam S, Heimdahl MPE (2001) Coverage based test-case generation using model checkers. In: International conference and workshop on the engineering of computer based systems 2001, pp 83–91
40. Clarke EM, Grumberg O, Peled DA (2000) Model checking. MIT Press, Cambridge
41. Marques-Silva J, Sakallah K (1999) GRASP: A search algorithm for propositional satisfiability. IEEE Trans Comput 48(5):506–521
42. Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an efficient SAT solver. In: Design automation conference, pp 530–535
43. Ericsson M (2004) Activity diagrams: What they are and how to use them. The Rational Edge
44. Zhu H, Hall P, May J (1997) Software unit test coverage and adequacy. ACM Comput Surv 29(4):366–427
45. McMillan KL Cadence SMV. Available at http://www.kenmcmil.com/
46. Reshadi M, Gorjiara B, Dutt N (2006) Generic processor modeling for automatically generating very fast cycle-accurate simulators. IEEE Trans CAD Integr Circuits Syst 25(12):2904–2918
47. Hennessy J, Patterson D (2003) Computer architecture: a quantitative approach. Morgan Kaufmann, San Mateo