Towards Automated Test Sequence Generation

U. Farooq, C. P. Lam and H. Li
School of Computer and Information Science
Edith Cowan University, Perth, Australia
ufarooq@student.ecu.edu.au, c.lam@ecu.edu.au, h.li@ecu.edu.au

Abstract

The article presents a novel control-flow based test sequence generation technique using UML 2.0 Activity Diagram, which is a behavioral type of UML diagram. Like other model-based techniques, this technique can be used in the earlier phases of the development process owing to the availability of the design models of the system. The Activity Diagram model is seamlessly converted into a Colored Petri Net. We proposed a technique that enables the automatic generation of test sequences according to a given coverage criteria from the execution of the Colored Petri Nets model. Two types of structural coverage criteria for AD based models, namely sequential and concurrent coverage are described. The proposed technique was applied to an example to demonstrate its feasibility and the generated test sequences were evaluated against selected coverage criteria. This technique can potentially be adapted to service oriented applications, workflows, and concurrent applications.

1. Introduction

UML is the de facto industry standard for software modeling and a key component in model driven development. Various UML diagrams, categorized as structural or behavioral, can be used to specify a particular view of the system. Structural diagrams such as the class, component and deployment diagrams are used to visualize the static view of the system. The behavioral diagrams include activity, state machine and interaction diagrams. The sequence diagram (one of the interaction diagrams) is provided for expressing time-oriented inter-object message sequencing. The state diagram is used for specifying the dynamic view of the system in terms of the sequence of states that a system can pass through during its lifecycle. The Activity Diagram (AD) is devised to visualize the flow-oriented aspects of the system that may encompass simple sequential, branching, looping and concurrency. The advantages of UML are its simple and intuitive syntax as well as its expressiveness to model large complex systems visually and efficiently. However, the lack of a formal semantics has hindered the direct applications of automated techniques on UML models for test case generation.

Model based testing (MBT) is an agile and systematic method which aims to automate the testing process through automated test suite generation and execution techniques and tools. A model is an intuitive approach for describing the structure and behavior of the system and is specific in representing particular aspects of the system according to defined objectives, assumptions and structures. The benefit of MBT is that it facilitates the construction of behavioral models early in a development lifecycle, thus exposing ambiguities in the specification and design [1]. More importantly, it supports re-use in future testing as these models captures the behavior of a software system and in contrast to a test suite, they are much easier to update if the specification changes [1]. In addition, MBT potentially supports earlier fault detection and a higher level of coordination between design and testing activities. One approach to generate a test suite from models is based on model checking [2], [3]. Model checking is a static analysis technique used to determine whether a specific property of interest is verifiable or if the system exhibits a particular functional behavior that violates this property. As it involves an exhaustive analysis of a model, involving the creation and exploration of the state space, it is prone to the state explosion problem [4], thus making it an infeasible approach for any analysis of non-trivial systems. In comparison, simulation, involving the execution of a model, is a dynamic and exploratory analysis of the state space and is relatively inexpensive. Analogous to software testing, it requires test vectors to execute the model and these are referred to as test cases.

In this paper we propose a test sequence generation (TSG) technique involving model execution of Colored Petri nets (CPN) that are derived from UML 2.0 AD. The resulting test sequences are then evaluated using two

1530-0803/08 $25.00 © 2008 IEEE
DOI 10.1109/ASWEC.2008.64

types of AD based coverage criteria that are introduced in this paper, namely sequential and concurrent coverage. The proposed technique allows software developers to continue to design in their familiar modeling environment and the resulting UML 2.0 ADs are seamlessly transformed into CPN for subsequent automatic test sequence generation. The developed technique has the advantages of both UML and CPNs whilst overcoming some of their shortcomings.

2. Motivation

A set of AD models can be seen as the visual "blueprints" of the functionality of the system and Kosters *et al.* [5] have stated that ADs can specify an entire set of use case scenarios in a single diagram. Lieberman [6] has suggested that specific scenarios can be generated by tracing a thread of execution from entry to exit through each AD. Stakeholders can easily understand the system behavior as well as monitor the development of system functionality by tracing execution paths through the ADs. A point to note is that deriving all possible usage scenarios manually from a set of realistic ADs can be a very time consuming task.

The potential scope and application of AD models in testing has been recognized by many researchers, for example, Briand and Labiche [7] described the derivation of usage scenarios as test guideline from AD models, Lieberman [6] suggested their use in monitoring the testing process and Bai *et al.* [8] explored the application of AD in scenario-based software testing. However, most of this work involved UML 1.x and UML 2.0 AD has significant changes from AD in UML 1.x. While there is some emerging work involving UML 2.0 ADs, most of the approaches have involved other types of UML diagrams (e.g. class diagrams, statecharts, etc).

While UML 2.0 AD has Petri Nets (PN)-like semantics, its informally defined semantics still have many ambiguities and inconsistencies, thus making it difficult to automatically derive test cases directly from ADs. The proposed technique resolves existing ambiguities and inconsistencies in a given AD and supports its seamless transformation into a CPN executable model, thus enabling the automatic generation of test sequences from an AD model for system behavioral testing. It brings the advantages of both domains, AD and PN, while overcoming their shortfalls. The rich syntax of AD is quite intuitive to program logic and is expressive enough to suit a wide range of application domains. It complements the daunting complexity of using CPN in designing complex and /or object-oriented systems. The well founded theory, strong analytical techniques and tool support of CPN and more importantly the informal foundation of the new UML 2.0 AD also make CPN an ideal platform for further analysis and evaluation of the AD models. At the same time, the proposed technique frees a tester from learning a new language or redesigning his already built models in order to execute them. The introduction of model based concurrent coverage criteria is timely as UML is now the dominant and standard modeling language, model based testing is also getting popular and no such criteria are available for concurrent system testing.

Section 3 describes related work and the proposed TSG technique is described in Section 4. Section 5 describes the test objective in terms of the test suite evaluation criteria. The case study, corresponding results and discussion are presented in Section 6 and the summary and future work in Section 7.

3. Related Work

Generation of test cases in MBT can be carried out via techniques such as model checking, graph-based approaches (e.g. random walk, or the Chinese Postman Walk), symbolic execution, and deductive theorem proving. Sivaraj and Gopalakrishnan [9] proposed a random walk based approach for model checking in parallel and distributed environment together with breadth first search. They defined four heuristic-based algorithms with configurable coupling between random walk and breadth first search for state space exploration. Lee *et al.* [10] presented an idea of using random walk for generating test sequences from Communicating Finite State Machine (CFSM) in conformance testing. According to the method, an adaptable random walk is guided by classified transitions in a directed graph and visited states are sampled for test traces.

Investigations involving the application of PN in software testing can be categorized into four groups: test suite generated (1) using typical state-space analysis techniques [11], (2) using invariant analysis [12], (3) deriving test scenarios by simulating or executing PN models and (4) directly deriving test data from a PN model using formal specification-based test generation techniques [13]. Ramaswamy & Neelakantan [14] showed the application of a PN based invariant analysis scheme for software design and testing. The proposed approach generates unique paths dubbed as sub-flows using the T-invariants obtained from a PN model. While the approach avoids the state explosion problem associated with model checking, it requires a high level of mathematical skills, thus inhibiting its applications at an industry level. Watanabe and Kudoh [11] proposed two CPN based algorithms for the automatic test suite generation in conformance testing involving concurrent systems. Their CP-tree method requires the generation of a reachability tree from a CPN model and test sequences are then produced by traversing through arcs and nodes from the root to the leaf nodes of the CP-tree.

Zhu and He [15] proposed four types of structural coverage criteria for testing PN model. The Transition based and State (place) based criteria are associated with the structural aspects of the Predicate (Prt) Net and the data (token) flow and specification oriented criteria are linked to the behavioral aspects of the Prt Net. However, a later study [16] identified some limitations with the proposed criteria (e.g. the 'state transition path' coverage and the 'K-concurrency length-L trace'). While the criteria we propose here are in some ways similar to those in [15] they are, however, specific for addressing structural coverage in ADs. For example, they specifically address the issue of branch and edge coverage in an AD which is not addressed in the criteria defined in [15].

Mingsong, Xiaokang and Xuandong [17] reported a test generation technique that used UML 2.0 AD as the design specifications. In order to obtain the execution traces, the approach involved program instrumentation where probes are inserted into the code of the software under test. Three types of test adequacy criteria for an AD were addressed, namely activity, transition and simple path coverage. A simple path was defined by the authors to be a path that has no loops or concurrency and the set of simple paths is generated using a modified depth first search (DFS) algorithm. The proposed technique however is prone to generating many invalid test cases.

Andrews, France and Craig [18] introduced a technique for dynamic analysis of the software design model comprising on class, activity and interaction diagrams. Their technique involved UML 1.4 and testing an executable model. An interesting aspect of this approach is that AD is used as a secondary artifact (to generate an executable model that captured the behaviour of a class and to obtain the interactions between objects from a set of ADs) and no coverage criteria for AD was considered here. Two sets of coverage criteria were subsequently used in [19] where UML design models were converted into an executable form for testing them. The approach used information from class and interaction diagrams for generating the required test cases. Dinh-Trong, Ghosh, & France [20] also used symbolic execution and a Variable Assignment Graph that incorporated information from UML class diagrams and sequence diagrams for generating test data which can then subsequently be used for testing design models.

Due to the lack of formal and executable semantics, UML models are not suitable for automation or formal behavioral analysis [21], [22]. Thus, many researchers have tried to integrate UML with various well-defined formal languages [13], [23-25]. In [13], Buchs *et al.* proposed a formal specification based test suite generation approach that transforms the UML models (i.e. class, collaboration and state diagrams) into a high level PN (CO-OPN) and then generates test data for the specified input domain. Pettit and Gomaa [25] used described the behavioral analysis of the system by transforming UML Collaboration diagram into CPN. Eichner *et al.* [23] introduced the PN based semantic for Sequence diagram for taking advantage of formal analysis tools and techniques. With the exception of the work in [13], most existing work involving UML have converted sequence diagrams to CPN for animation and for V & V of requirements and design models.

4. Generating Test Cases

Test case generation has always been fundamental to the testing process. Bertolino [26] articulated that test case generation is a most challenging and an extensively researched activity. In software testing, the definition of a test case is contextual and relates to the corresponding test case generation technique. Thus, it is important to clearly define the basic terms and concepts used throughout this paper. The test data is a set of inputs, expected outputs and execution conditions derived from a low level platform specific model using other techniques (e.g. equivalence partitioning, boundary values etc.) for a particular test case. A test sequence is a high level test where a sequence of tasks or operations is directly generated from a high level behavioral model according to a particular test objective. As the focus of the proposed technique is the behavioral correctness of the system, the generated test sequences enforce the functional correctness of tasks/operations, order of execution and the dependencies among the various tasks or operations. The term test suite implies the collection of test sequences.

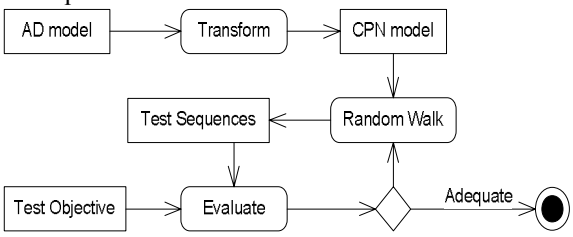

Figure 1: Automated TSG Process

Similar to execution paths in code based testing, it is the execution sequences of model artifacts that interest testers more than the execution of an individual artifact. For complete testing, it would need to test every possible combination of artifacts. Unfortunately, this approach is not scalable and is prone to combinatorial explosion. A manual exploration of the model or test suite generation is not viable and automatic TSG is most desirable as the test suite is automatically derived from the model according to the given test objective. The test objective specifies the required adequacy criteria as a guideline.

The overall process of the proposed TSG technique is illustrated in the figure-1.

As shown in figure 1, we propose a three-stage TSG process. In UML 2.0, after major revision, AD has a new PN-like token flow semantic which lacks precision and consistency in the description of flow rules as reported in [22], [24]. The imprecision and ambiguity in the semantic could yield differing interpretations and even unexpected runtime behavior of a syntactically valid AD. Thus, deriving a test suite directly from an AD could result in invalid test sequences. In fact, it is implicitly required to formulate the exact runtime behavior by transforming the AD into a precise semantic for its execution and analysis. Hence the inspiration for transforming AD into CPN stems from the fact that the CPN is a formal modeling language that provides an unambiguous and executable specification which has been quite successful in modeling both concurrent and sequential systems.

In the first stage, an AD model is transformed into a CPN model using the approach discussed in [27]. Then using the algorithm proposed in the following section, test sequences are generated and finally evaluated against a given test objective. If the test suite does not satisfy the required criteria then more test sequences are generated via another iteration of the random walk. This process continues until the required criterion specified in test objective is satisfied. In the next section, we introduce a random-walk based technique for automatic TSG.

```
if (Model is not initialized)
  initialize the model;
Endif
While (Action-Final place is not marked)
  do
     For all enable transitions;
       if (all postconditions are satisfied)
          Fire an enabled transition randomly;
       endif
     // end of For
     /* firing a transition will remove the input
      token and pass on to the output places. */
     if (Fired transition is not dummy)
        Record the fired transition for token trace;
     endif
     if (New  marked place is not dummy)
        Record newly marked place for token trace;
     endif
     For all transitions
       if (all inputs and preconditions are satisfied)
            Enable transition;
       endif
     // end of For
  enddo
Output the token trace;
```

Figure 2: Pseudo Code for Random Walk Algorithm for TSG

Random-walk based TSG

The concept of a random-walk is based on the theory of probability and referred to a movement where the path is initiated from a specific point and each successive step is then made randomly. In a connected graph, the trajectory of a random walk includes all visited nodes. In general, a random-walk is considered suitable for discrete problems and needs adaptation for a particular application. This apparently simple technique has received a fair amount of attention and has been applied in areas such as wireless networking, World Wide Web, model checking etc. In fact, Robinson [1] has discussed its use for MBT and has stated that the very random nature of this approach produces useful test cases which overcome the "pesticide paradox" problem in testing.

The graph based representation and formal executable semantic of CPN makes it ideal for using the random-walk algorithm for executing the model and recording the execution traces. In CPN, a token can abstractly represent a control or stimuli in the model and therefore the walk will simulate the token-flow during the model execution. Although the walk progresses randomly, we have adapted the random selection process to the predefined semantic for CPN to avoid any invalid paths. Moreover, as the technique is based on pseudorandom exploration of the model, the model inscriptions such as conditions and data information are not used during the random walk. Therefore all the conditions and constraints associated with an AD model have not been mapped across to the CPN model. This is specific only to the proposed random-walk algorithm as it aims to generate all possible control flow paths from the model. Moreover, as the alternative paths at any branching node are selected randomly, the description and evaluation of condition statements becomes superfluous. However this condition and constraint information can be used subsequently in conjunction with the previously generated paths at the test data generation stage involving various black box testing techniques such as equivalence partitioning. Furthermore, as one of the many enabled transitions in CPN will eventually occur/fire, we postulate that the walk randomly selects a transition and in visiting it makes one step of the walk. Similarly, the traversal of the walk through a place node is also marked as a step of in its path. A test sequence is any path in the CPN model from its initial node to its final node. Considering the concurrency support in CPN, the interleaving paths between fork/joins nodes intuitively constitute as potential test paths for concurrent testing i.e. synchronized sequence (SYN-sequence is defined as a sequence of synchronization events e.g. read/write, P/V, lock/unlock operations).

Table 1: CPN nodes with corresponding AD nodes and observing token-game semantic

AD2 Node	CPN Node	Semantic
Action	Transition	▪ An action can only start execution when all inputs have tokens. ▪ When an action starts execution it consumes tokens on all inputs. ▪ On completion, tokens are

		offered on all outputs.
Initial	Place	▪ Initialize with a token whenever the enclosing activity is invoked. ▪ An outgoing token can follow only one edge.
Activity Final	Place	▪ When a token reached it, the enclosing activity will be terminated; particularly, all executing actions are stopped, all other tokens are destroyed and all flows are terminated.
Flow Final	Place	▪ All tokens arriving on it are destroyed.
Fork	Transition	▪ Incoming tokens are duplicated to all outputs.
Join	Transition	▪ All incoming tokens are joined according to the rules given in [28].
Decision	Place	▪ Each incoming token can traverse only one outflow.
Merge	Place	▪ All incoming tokens are forwarded to a single outflow without synchronizing and joining them.

The pseudo code for the adaptive random walk algorithm is described in figure 2. Unlike a conventional random-walk algorithm, where all subsequent transitions should be enabled after a step in a random-walk, in this proposed algorithm only those transitions that have satisfied CPN semantic (i.e. all input tokens are available, all pre and post conditions for a transition are satisfied) are to be enabled.

Following the defined semantic in Table-1, the random walk begins from an initial node dubbed as 'Init' place in a CPN model. The walk then randomly selects one of the outgoing arcs according to the corresponding semantic of an initial node and the given token moves along the selected arc. After an occurrence of a transition a token is passed to each output place. The walk continues as long as it is visiting nodes with non-zero outgoing arcs. Currently the algorithm deals with an AD where there is only one Activity-Final node which is labeled as 'Final' to distinguish it from the flow final node. Once the walk reaches a node without any outgoing arc and explicitly labeled as 'Final', the walk terminates for the current iteration according to the semantic of an Activity-Final node. Using the random walk approach, test sequences are automatically generated in each iteration by recording the trace of the random walk starting from the initial node to the final node. At the end of each iteration the attained coverage of the test suite is evaluated against the specified test adequacy criteria and a decision is made whether to continue with another iteration of the random walk.

5. How is the quality of the generated TS measured?

Zhu *et* a*l*. [29] described the use of coverage based test adequacy criteria for test quality measurement. Generally, coverage criteria are used to determine the adequacy of the test suite and therefore are considered as an essential part of a testing method. As a general rule of thumb, the test suite with a higher coverage is considered to be better in quality. This is based on the fact that a higher coverage could potentially reveal more defects [30], [31]; hence eventually improves the software quality. Test coverage is measured in terms of the percentage of specific constructs that have been executed at least once during execution according to the defined coverage criterion.

Similar to code based testing techniques, in MBT, the test suite is also generated with the aim of providing maximum testing coverage of the system under test. However, during the test data generation process the coverage based adequacy metric usually relates to the nature of the source (e.g. code, model or fault-set) of the tests and therefore measures the artifacts of the source involved in TSG process. In this context, the coverage metric establishes the link between the test suite and source and further, imparts significant information about the contents and characteristics of the test suite.

Coverage Criteria

In the literature, a large number of coverage criteria have been suggested. However, we find only a few criteria that are appropriate for observing the coverage information in an AD model which has specific requirements such as an ordered execution of tasks/operations in isolated control paths or threads and coordinated execution of tasks/operations in synchronous/asynchronous parallel control paths or threads. In the following section, we present sequential and concurrent criteria adapted from [29] and [32] respectively.

Sequential Coverage Criteria:

Control flow based testing has already been extensively researched and a number of control flow based coverage criteria have been proposed. Although at the basic level, control based criteria are defined on a graph structure, and hence needs some adaptation to be associated with a particular technique and application. Testing in isolated control paths or threads is analogous to the control flow based testing. Therefore, sequential control flow based coverage criteria are considered appropriate here and are adapted for AD models. It will allow measuring and determining the scenarios that need to be covered by a behavioral test suite in order to be considered adequate.

- **CPN Transition Coverage**: A control flow based criterion that measures the number of transitions executed during testing. In CPN, a transition node is a set of outputs (a_1, a_2,.., a_n) that all are traversed after the transition's execution. On mapping back to the AD, transition coverage would seek the execution of each action, fork and join node at least once, thus, the test suite at a minimum includes execution of all actions, forks and joins nodes at least once. It is analogous to node-coverage in state-based testing and could be considered as the elementary and minimal required testing criterion.
- **AD Branch Coverage**: A control flow based criterion that measures the number of branches executed during testing at least once. In CPN, a place node with multiple outputs corresponds to a decision node in an AD with a set of branches (a_1, a_2,.., a_n) such that only one branch is traversed for an execution. For full branch coverage, a test suite needs to have at least one test case for each branch which also includes the execution of all transitions. Therefore, branch coverage subsumes the transition coverage.
- **AD Edge Coverage**: According to this criterion, the number of edges traversed during testing is determined. In an AD model, a number of activities/actions are linked in a particular order which represents a sequential flow. This aspect of flow is visual and superficial. Thus we need to remember that AD has particular semantic as well which indicates the dependency between linked actions/activities. It means an action can only execute after the earlier action has completed and all input tokens are available. Note that visually, an edge is a link between two nodes or a link between two operations in a scenario and it only shows the sequence of actions. The edge coverage criterion ensures that all arcs between all types of nodes must be evaluated during testing.
- **All Path Coverage**: It is a given to ensure that all possible ordering and sequencing of flow occurred at least once during testing. Path based coverage criterion is the strongest but is difficult to achieve as there could potentially be an infinite numbers of execution paths. As an execution path can be represented by the combination of states, transition or both in CPN; the number of executions involving all combinations becomes so large, that in practice, exhaustive testing at this level is deemed to be impractical.

Concurrent Coverage criteria:

Studies have shown that conventional test coverage criteria are inadequate for concurrent program testing [33], [34]. Consequently, we seek further analysis of the generated test suite in relation with the concurrent nature of the AD, namely the concurrency and synchronization constructs (i.e. fork and join), to mimic the interactions between sub-processes. The fork node in an AD depicts the case where all of the actions following a specific action may execute concurrently. It splits the control into 'n' sub-processes and allows all of them to execute simultaneously. On the other hand, the join node mimics the convergence point for all of the concurrent sub-processes. Consequently a model based test suite could reveal concurrency faults such as failure to realize a certain execution order of concurrent operations and failure to identify the blocking/deadlocking. Definitions of concurrent coverage criteria modified from [32] are discussed below.

- **Interleaving node coverage:** The interleaving node coverage criterion is about the execution of n-wise permutated set of concurrent nodes in 'n' synchronized sub-processes. In AD, the combination of all types of nodes (i.e. action, activity, decision and merge) will make up the permutated set. The degree to which the permutated set has been exercised by a test suite implies the coverage attained according to this criterion.
- **Interleaving edge coverage:** Similarly, the interleaving edge coverage criterion requires that all the n-wise permutated set of edges in 'n' synchronized sub-processes are executed during testing. The percentage of the paired edges exercised during testing implies the degree in meeting this adequacy criterion.
- **Synchronized path coverage:** The execution of the set of all possible interaction between concurrent sub-processes is required in order to satisfy the synchronized path coverage criterion. Given the concurrent execution of sub-processes, the number of interleaving paths grows exponentially along with the growth of the number of sub-processes. In this way, attaining adequate coverage for this criterion is intractable and therefore it is mentioned here only for the purpose of completeness.

6. Illustrated Example

For evaluation and demonstration of the proposed technique, we use the model shown in figure 3. It describes an enterprise customer commerce system taken from [35] and contains an AD describing a system level process. It describes the process of online purchasing of products that is comprised of two sub-processes: authentication and shopping. The authentication process allows the user to login and in the case of a new user, it allows the new user to register first. Within the shopping process, a user can order the selected products and can configure his/her account if required. Following the technique proposed in section-4, this sample AD model is first converted into a CPN model. Using the transformation methodology proposed in [27], we

acquire the CPN version of this model as shown in Figure 4.

Using the proposed random-walk based TSG algorithm and the resulting CPN model, a test suite is then generated. The trace of the walk is recorded for each iteration of the random-walk and the generated test suite is presented in Tables 2 and 3. The generated test sequences are then evaluated according to the coverage criteria described in the previous section.

Results and Discussion

Tables 2 and 3 summarized the degree to which the generated test suite, obtained from the example model via the proposed approach, meets the various test coverage criteria. Table 2 presents the evaluation of generated TS against the sequential coverage criteria. Similarly Table 3 presents the interleaving node and edge coverage analysis of the generated TS. The column names in both tables are abbreviated as Cd, Ud, Cov, Uq and CC for covered, uncovered, coverage, unique and cumulative coverage respectively. The numbers of executed and missed model artifacts are indicated in columns 'Cd' and 'Ud' respectively for a particular test sequence. The degree of individual coverage of each test sequence is presented in column 'Cov'. In Table 3, the 'Cd' and 'Ud' columns show the number of paired interleaving artifacts in the concurrent processes that have been exercised by an individual test sequence. The 'Uq' column contains the number of unique artifacts that were previously uncovered. The 'CC' column is about the cumulative coverage gained by each additional test sequence. As an example, TS-1 in Table 2 tested 5 artifacts and missed 12 artifacts according to the branch coverage criterion. As the total number of artifacts for branch coverage criterion was 17, the coverage it attained was 29.4%. Moreover, as TS-1 was the first test from the suite, all tested artifacts were unique and therefore the cumulative coverage was also 29.4%. After that, TS-2 tested 12 artifacts and missed only 5. It attained 70.6% coverage and, as only 8 out of 12 tested artifacts were unique, the coverage accumulated by this test sequence was 76.4%. With the execution of two new artifacts by TS-3, the CC reaches to 88.2%. However, from TS-4 to TS-9, the CC value does not improve as no new artifact s has been executed. The CC is further improved when TS-10 executed a new artifact. After that, CC remains stagnant for TS-11 and TS-12. Finally, CC reaches to 100% when TS-13 executed the remaining artifact. For the edge coverage criterion, the test suite attained 100% cumulative coverage in a similar pattern. Despite this apparent similarity at test suite level, the individual test case coverage reveals the fact that both the branch and edge coverage criteria are not similar and the edge criterion overlaps branch criterion.

As seen from the given data, the generated test suite is adequate in meeting the sequential coverage criteria but is not sufficient in meeting the concurrent testing criteria, thus requiring more test cases to be generated. For a sound analysis we further analyze the generated test suite in terms of the following characteristics: redundancy, test priority and test suite size.

Redundancy: The basic idea behind a random-walk based algorithm is enumerating all the possible and unique control flow paths in an AD model. With arbitrary interactions in the concurrent processes, the number of permutated paths grows exponentially and manual or exhaustive test generation techniques (e.g. depth first algorithm) are therefore infeasible. Consequently, the random-walk based TSG algorithm is deemed more than adequate because of the exploratory nature of the algorithm. The algorithm incrementally generates more test sequences and stops once a specified coverage criterion is achieved. However, the proposed algorithm is not ideal, since we can see from the coverage data in Tables 2 & 3 that it has some tendency to produce a number of redundant test cases as a result of the stochastic nature of the proposed algorithm. However, a careful analysis of the data at the test level reveals that some degree of redundancy is indispensible if the associated test cases add to the cummulative coverage at the test suite level. At the same time there are test cases such as TS-4,5,6,7,8,9 that do not bring any extra coverage and thus should be discarded. A future improvement to the existing algorithm is to modify the algorithm to guide the random-walk heuristic to skip any redundant tests cases for optimum test suite generation.

Test priority: An analysis of the Tables 2 & 3 also indicates that some test cases have a higher coverage with respect to a specific criterion than others but at a test suite level, owing to the order of execution, it does not bring any additional coverage. For instance, in Table 2, TS-11 has almost double of the coverage of TS-1 but appears redundant due to the test execution order. Thus prioritizing the test cases according to their coverage criteria could further optimize the test suite. Future work in this project would also investigate approaches for prioritization of the generated test cases.

Test suite size: Another important aspect is the size of the generated test suite or, in other words, the number of tests needed with this approach to attain adequate coverage. As a guideline we assume that there should be at least one test case for each independent path. Given the fact that an AD model itself depicts the control flow graph of a particular module, the cyclomatic complexty metric [36] was used to analyse the example model and 12 distinct control flow paths were found. Thus, initially, we ran the algorithm for 12 iterations and generated 12 test cases. The test suite is then evaluated for both sequential and concurrent coverage.

Subsequently more test cases are added incrementally until the complete coverage is achieved. The results indicate that the algorithm produces 100% coverage for sequential criteria after 13 iterations but as indicated in Table 3, many more tests cases will be required to be generated in terms of meeting the concurrent criteria. A point to note is that every test case generated here is essentially feasible as the execution of the algorithm is also guided by the default AD semantic. Due to space limitations we have omitted the action, all-path and synchronized-path coverage analysis. It is essential to note that the test suite is generated automatically; the coverage is analyzed manually for now and will be easily automated in the near future, as it is not a complex task.

7. Summary and future work

We described a TSG technique and two types of coverage criteria for AD based models. The technique is characterized by an automatic transformation of an AD model into a CPN model and then automatically deriving test sequences by executing the CPN model. We illustrated the test suite generation process with an example and demonstrated the usability of the technique by evaluating the generated test suite against some specified coverage criteria. Moreover we demonstrated the key characteristic of the technique in that it does not generate any infeasible test sequence.

The technique proposed in this paper is defined along the concept of platform independent model (PIM) and platform specific model (PSM) in model driven architecture (MDA), where the PSM is developed in consistence with the PIM and then software is implemented according to the PSM. The contribution of this work is two fold: (1) it introduces a new model based TSG technique to automatically derive test sequences from UML 2.0 AD; (2) a framework is defined in section-4 for generating and measuring model based test suites and sets a base for developing exploratory testing techniques. The current work is limited to the intermediate level AD and covered only control flow related aspects of the model. Future work will expand to other aspects of AD such as data flow and high level design artifacts. Efficiency analysis of both the proposed TSG technique and coverage criteria will also be a part of our future work.

Bibliography

[1] H. Robinson, "Graph Theory Techniques in Model-Based Testing," presented at International Conference on Testing Computer Software, 1999.

[2] M. P. E. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao, "Auto-generating Test Sequences Using Model Checkers: A Case Study," presented at Formal Approaches to Software Testing, Montreal, Quebec, Canada, 2003.

[3] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized Symbolic Execution for Model Checking and Testing," presented at Tools and Algorithms for the Construction and Analysis of Systems, Warsaw, Poland, 2003.

[4] S. Merz, "Model Checking: a tutorial overview," presented at Modeling and verification of parallel processes 2000.

[5] G. Kosters, H. W. Six, and M. Winters, "Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specification," *Requirements Engineering*, vol. 6, 2001.

[6] B. Lieberman, "UML Activity Diagrams Versatile Roadmaps for Understanding System Behavior," *The Rational Edge*, pp. 12, 2001.

[7] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," Carleton University TR SCE-01-01-Version 4, June, 2002 2002.

[8] X. Bai, C. P. Lam, and H. Li, "An Approach to generate the Thin-threads from the UML Diagrams," presented at Computer Software and Applications Conference (COMPSAC), 2004.

[9] H. Sivaraj and G. Gopalakrishnan, "RandomWalk Based Heuristic Algorithms for Distributed Memory Model Checking," University of Utah, School of Computing, Salt Lake City 2003.

[10] D. Lee, K. K. Sabnani, D. M. Kristol, and S. Paul, "Conformance Testing of Protocols Specified as Communicating Finite State Machines - a Guided Random Walk Based Approach," *IEEE Trans. on Communications*, vol. 44, pp. 631-640, 1996.

[11] H. Watanabe and T. Kudoh, "Test Suite Generation Methods for Concurrent Systems Based on Colored Petri Nets " in *Asia-Pacific Software Engineering Conference*: IEEE Computer Society, 1995.

[12] S. Ramaswamy and R. Neelakantan, "Software Design and Testing Using Petri Nets: A Case Study Using a Distributed Simulation Software System," in *Performance Metrics for Intelligent Systems*. Gaithersburg, MD, 2002.

[13] D. Buchs, L. Pedro, and L. Lucio, "Formal Test Generation from UML Models," in *Research Results of the DICS Program*: Springer, 2006.

[14] S. Ramaswamy, " A Petri net based approach for establishing necessary softwaredesign and testing requirements," presented at Systems, Man, and Cybernetics, 2000 IEEE International Conference on, Nashville, TN, USA, 2000.

[15] Hong Zhu and X. He, "A theory of testing high level Petri nets," Oxford Brookes University, Technical Report CMS-TR-2000-02, January, 2000 2000.

[16] Junhua Ding, Peter J. Clarke, Gonzalo Argote-Garcia, and X. He, "Evaluating Test Adequacy Coverage of High Level Petri Nets Using Spin," Florida International University, Miami, Technical Report FIU_SCIS 2006-05-02, 02-05-2006 2006.

[17] Q. X. Chen Mingsong, Li Xuandong, "Automatic Test Case Generation for UML Activity Diagrams," presented at AST, Shanghai, China, 2006.

[18] A. A. Andrews, R. B. France, S. Ghosh, and G. Craig, "Test adequacy criteria for UML design models," *Softw. Test., Verif. Reliab.*, vol. 13, pp. 95-127, 2003.

[19] T. T. Dinh-Trong, N. Kawane, S. Ghosh, R. B. France, and A. A. Andrews, "A Tool-Supported Approach to Testing

UML Design Models," presented at International Conference on Engineering of Complex Computer Systems (ICECCS 2005), Shanghai, China, 2005.
[20] T. T. Dinh-Trong, S. Ghosh, and R. B. France, "A Systematic Approach to Generate Inputs to Test UML Design Models," presented at 17th International Symposium on Software Reliability Engineering, Raleigh, North Carolina, USA, 2006.
[21] J. B. Jørgensen, "Coloured Petri Nets in Development of a Pervasive Health Care System," presented at 24th International Conference Applications and Theory of Petri Nets (ICATPN 2003), Eindhoven, The Netherlands, 2003.
[22] H. Störrle, "Semantics of Control-Flow in UML 2.0 Activities," presented at VL/HCC Rome, Italy, 2004.
[23] C. Eichner, H. Fleischhack, R. Meyer, U. Schrimpf, and C. Stehno, "Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets," presented at 12th International SDL Forum, Grimstad, Norway, 2005.
[24] R. Eshuis and R. Wieringa, "A Formal Semantics for UML Activity Diagrams - Formalising Workflow Models," University of Twente, Department of Computer Science, Enschede, Netherlands, CTIT technical reports 04-2001 2001.
[25] R. G. Pettit and H. Gomaa, " Modeling State-Dependent Objects using Coloured Petri Nets " presented at Workshop on Modelling of Objects, Components, and Agents, Aarhus, Denmark, 2001.
[26] A. Bertolino, "Software Testing Research and Practice," presented at 10th International Workshop on Abstract State Machines (ASM'2003), Taormina, Italy, 2003.
[27] U. Farooq, C. P. Lam, and H. Li, "Transformation Methodology for UML 2.0 Activity Diagram into Colored Petri Nets," presented at 4th IASTED International Conference on Advances in Computer Science and Technology Phuket, Thailand, 2006.
[28] C. Bock, "UML 2 Activity and Action Models," *Journal of Object Technology* vol. 2, pp. 43-53, 2003.
[29] Z. Hong, A. V. H. Patrick, and H. R. M. John, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, pp. 366-427, 1997.
[30] P. G. Frankl and S. N. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," *IEEE Transactions on Software Engineering*, vol. 19, pp. 774-787, 1993.
[31] W. E. Wong, J. R. Horgan, L. London, and A. P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," presented at 17th International Conference on Software Engineering, 1995.
[32] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka, "Testing concurrent programs: a formal evaluation of coverage criteria," presented at Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE '96) Washington, DC, USA, 1996.
[33] R.-D. Yang and C.-G. Chung, "A path analysis approach to concurrent program testing," presented at Ninth Annual International Phoenix Conference on Computers and Communications, Scottsdale, AZ, USA, 1990.
[34] K. Tai, "Testing of concurrent software," presented at 13th Annual International Computer Software and Applications Conference, 1989.
[35] J. M. Küster, J. Koehler, and K. Ryndina, "Improving Business Process Models with Reference Models in Business-Driven Development," presented at Business Process Management Workshops, 2006.
[36] T. McCabe and C. Butler, "Design Complexity Measurement and Testing," *Communications of the ACM*, vol. 32, pp. 1415-1425, 1989.

Table 2: Evaluation of the generated TS with branch and edge coverage criterion

	Branch Coverage Analysis					Edge Coverage Analysis				
Tests	**Cd**	**Ud**	**Cov %**	**Uq**	**CC %**	**Cd**	**Ud**	**Cov %**	**Uq**	**CC %**
TS-1	5	12	29.4	5	29.4	19	14	57.6	19	57.57
TS-2	12	5	70.6	8	76.5	28	5	84.8	10	87.87
TS-3	11	6	64.7	2	88.2	26	7	78.8	2	93.93
TS-4	8	9	47.1	0	88.2	22	11	66.7	0	93.93
TS-5	7	10	41.2	0	88.2	23	10	69.7	0	93.93
TS-6	8	9	47.1	0	88.2	22	11	66.7	0	93.93
TS-7	6	11	35.3	0	88.2	20	13	60.6	0	93.93
TS-8	9	8	52.9	0	88.2	24	9	72.7	0	93.93
TS-9	7	10	41.2	0	88.2	21	12	63.6	0	93.93
TS-10	9	8	52.9	1	94.1	25	8	75.8	1	96.97
TS-11	10	7	58.8	0	94.1	26	7	78.8	0	96.97
TS-12	8	9	47.1	0	94.1	21	12	63.6	0	96.97
TS-13	10	7	58.8	1	100	25	8	75.8	1	100

Table 3: Evaluation of the generated TS with interleaving node and edge coverage criterion

	Interleaving Node Coverage Analysis					Interleaving Edge Coverage Analysis				
Tests	**Cd**	**Ud**	**Cov %**	**Uq**	**CC %**	**Cd**	**Ud**	**Cov %**	**Uq**	**CC %**
TS-1	2	12	14.3	2	14.3	2	26	7.1	2	7.14
TS-2	1	13	7.14	1	21.4	1	27	3.6	1	10.7

TS-3	2	12	14.3	1	28.6	2	26	7.1	2	17.9
TS-4	2	12	14.3	0	28.6	2	26	7.1	1	21.4
TS-5	1	13	7.14	0	28.6	1	27	3.6	0	21.4
TS-6	2	12	14.3	0	28.6	2	26	7.1	0	21.4
TS-7	1	13	7.14	0	28.6	1	27	3.6	0	21.4
TS-8	2	12	14.3	0	28.6	2	26	7.1	0	21.4
TS-9	1	13	7.14	0	28.6	1	27	3.6	0	21.4
TS-10	2	12	14.3	2	42.9	2	26	7.1	2	28.6
TS-11	1	13	7.14	0	42.9	1	27	3.6	0	28.6
TS-12	2	12	14.3	1	50	3	25	11	2	35.7
TS-13	2	12	14.3	0	50	2	26	7.1	0	35.7

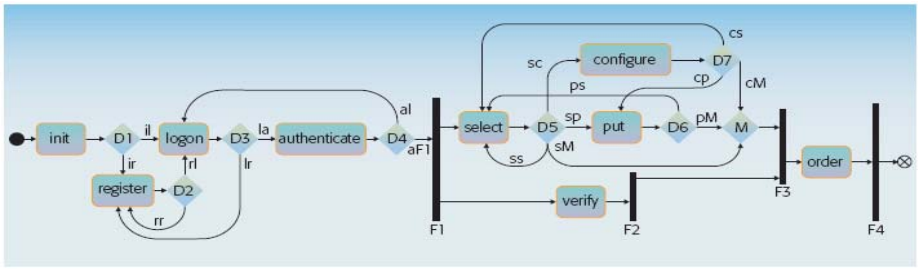

Figure 3: AD model of an Enterprise Customer Commerce System[35]

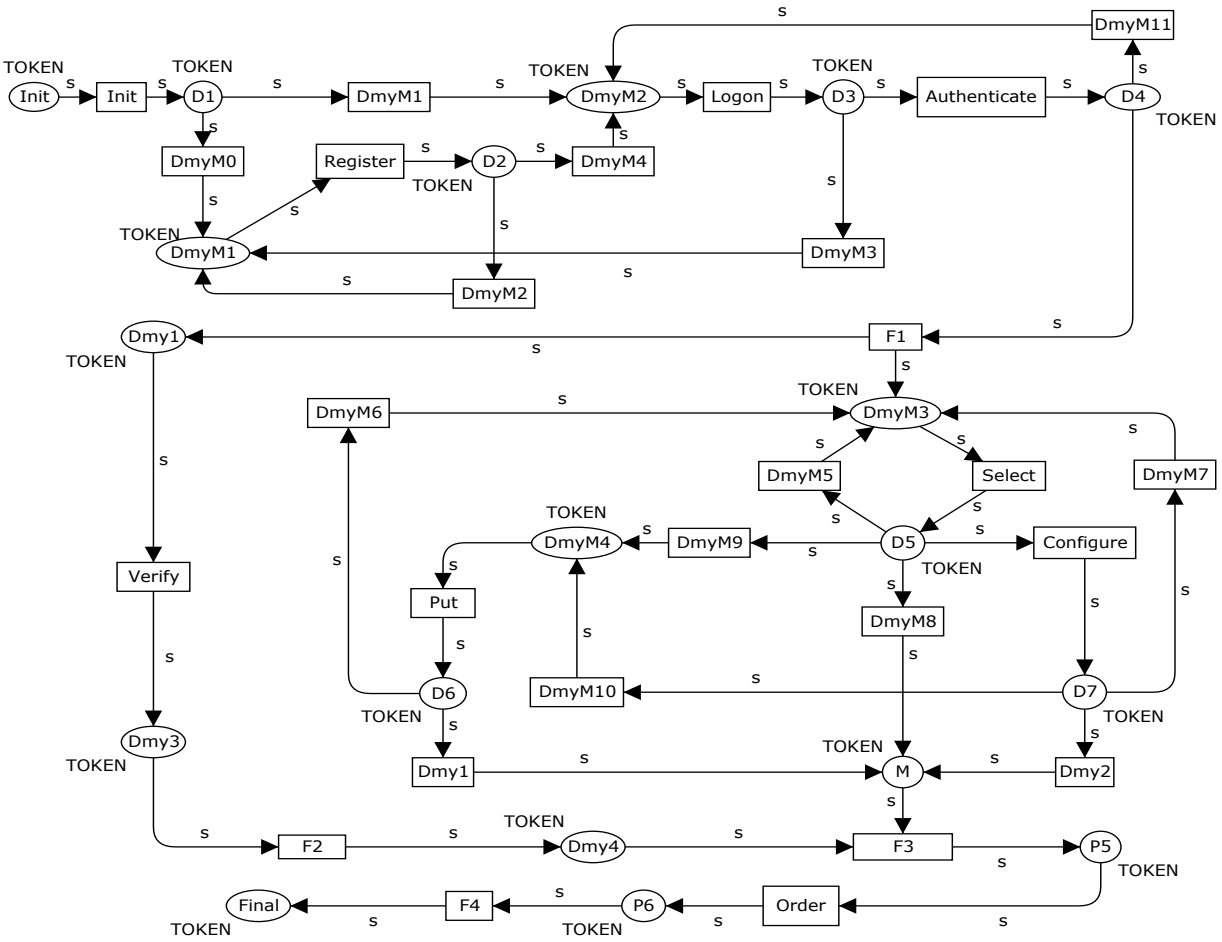

Figure 4: CPN version of Enterprise Customer Commerce System (without initial marking)