MISTA in a Nutshell

January 2012

1. What Is MISTA?

MISTA (Model-based Integration and System Testing Automation) is a tool for model-based test generation. Given a Model-Implementation Description (MID), MISTA automatically generates executable test code in the target language. The test code can then be executed against the system under test (SUT). As shown in Figure 1, a MID consists of a test model, a model-implementation mapping (MIM), and user-provided helper code. Function nets[1] are the primary modeling notation, while UML protocol state machines and contracts (preconditions and postconditions) are also supported. A MIM maps individual elements in a test model into target code.

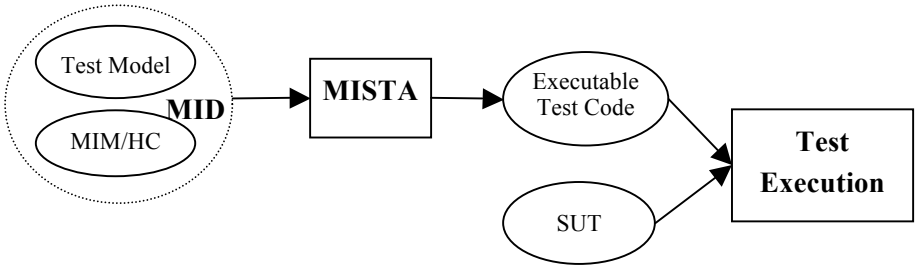

Figure 1. Test generation in MISTA

As a model-based testing tool, MISTA has the following features:

- Function nets as test models can represent both control- and data-oriented test requirements. They can be built at different levels of abstraction and independent of the implementation. For example, entities in a test model are not necessarily identical to those in the SUT. Building a MID does not require availability of the source code of the SUT.
- Test models can be edited in either graphical or spreadsheet format and analyzed via verification and animation.
- Tests can be generated automatically to meet various coverage criteria of test models, such as reachability coverage (round-trip paths), sneak path, state coverage, transition coverage, depth coverage, and random generation. Partial ordering and pairwise techniques can be used to reduce tests.
- Executable test code or test scripts can be generated automatically in various languages and test frameworks, including Java (JUnit and WindowTester), C# (NUnit), C++, PHP (PHPUnit), VB, C, HTML (Selenium IDE), and Robot Framework.
- On-the-fly testing (simultaneous generation and execution of tests) and online execution of existing tests are based on JSON-RPC and XML-RPC. They are HTTP-based RPC protocols.

[1] Function nets are a simplified version of high-level Petri nets (colored Petri nets and Predicate/Transition nets).

MISTA supports various testing activities, including, but not limited to, the following:

- Function testing: MISTA can be used to generate function tests for exercising interactions among system components.
- Acceptance testing and GUI testing: MISTA can be used to generate various sequences of use scenarios and GUI actions.
- Security testing: MISTA can be used to test whether or not a SUT is subject to security attacks by using threat models and whether or not a SUT has enforced security policies by using access control models[2].
- Programmer testing: MISTA is not just for testers, but also for programmers. For example, it can be used to test interactions within individual classes or group of classes. MISTA can also be used in test-driven development, where test code is created before the product code is written.
- Regression testing: It is easy to deal with system changes - only the MID needs to be updated.

MISTA offers the benefits one can reap from model-based testing techniques. The modeling activity helps clarify requirements and enhances communication between developers and testers. Without a good understanding about the SUT, testers would not be able to perform high quality testing. Automated test generation and execution enable more test cycles due to repeatable tests and more frequent test runs. The generated tests assure required coverage of test models with little duplication. The automation also facilitates quick and efficient verification of requirement changes and bug fixes and minimizes human errors.

2. Terminology and Test Code Structure

Here are some important terms pertaining to test code generation in MISTA.

- Test case: a test case includes a sequence of test inputs (component/system calls) and respective assertions (test oracles). Each assertion compares the actual system state against the expected result to determine whether the test passes or fails. Each test case may call the setup method in the beginning of the test and the teardown method at the end of the test.
- Setup method: a setup method is a piece of code called at the beginning of each test case.
- Teardown method: a teardown method is a piece of code called at the end of each test case.
- Test suite: a test suite is a list of test cases. It may include alpha code executed in the beginning of the test suite and omega code executed at the end of the test suite.
- Alpha code: Alpha code refers to the code for the initialization of a test suite. It is executed before the performance of any test cases.

[2] If interested, contact the author for more information.

- Omega code: Omega code refers to the code for the cleanup of a test suite. It is executed after all test cases are performed.
- Test driver: test driver refers to the code that starts the execution of a test suite.
- Header: header refers to the code defined at the beginning of a test program. In Java, the header includes package and import statements, whereas in C#, it includes namespace and using statements. HTML/Selenium test code for web applications does not need the header.
- Local code (or code segment): local code refers to the code that user provides, in addition to setup/teardown and alpha/omega. Local code may include variable/constant declarations and methods to be used anywhere within the generated test program (e.g., called by a setup or teardown method).

Appendix A shows the general structure of a Java test program generated by MISTA. It consists of the header, object reference, setup/teardown methods, test case methods, local code, test suite method (including call to the alpha/omega code), and test driver. **These elements are optional**; not all of them are needed for a specific application. The header, alpha/omega segments, setup/teardown methods, and local code are referred to as user-defined **helper code**. Test code in other object-oriented languages, including C#, C++, and VB, has a similar structure. Test code in HTML/Selenium does not have the header or the class structure.

3. Test Models and Test Generation

A test model describes what will be tested against a SUT. It is used to generate tests according to a certain criterion. Before test generation, MISTA also allows the test model to be analyzed so as to find potential errors.

A function net consists of places (represented by circles), transitions (represented by rectangles), labeled arcs between places and transitions, and initial states. A place represents a condition or state, whereas a transition represents an operation or function (e.g., component call). Arc labels represent parameters associated with transitions and places. There are three types of arcs:

- Directed arc from a place to a transition (representing transition's input or precondition) or from a transition to a place (representing transition's output or postcondition). A special output arc labeled by "RESET" is called a RESET arc. All the data in the output place connected by this arc will be cleared when the transition is fired.
- No-directed (or bi-directional) arc between a place and a transition (representing both input/output or pre-/post-condition of the transition)
- Inhibitor arc from a place to a transition (representing a negative precondition of the transition)

A transition includes an event, its formal parameters, a guard, and an effect. Guard and effect are optional conditions. A condition is a list of predicates separated by ",",

which means logical "*and*". A predicate is for the form *[not] p* (x_1, x_2, ...x_n), where "*not*" (negation) is optional. The built-in predicates for specifying guard conditions include =, <> (!=), >, >=, <, <=, +, -, *, /, %, etc.

An initial state represents a set of test data and system settings. It is a distribution of data items (called tokens) in places. A data item is of the form p(x1, …, xn), where (x1, …, xn) is a token in place p. "()" is a non-argument token. There are two ways to specify an initial state. One is to specify tokens in each place. The other way is to use an annotation, which starts with the keyword "INIT", followed by a list of data items (multiple items are separated by ",").

A hierarchy of function nets can be built by linking a transition to another function net (called subnet).

Consider a blocks game, where a single-handed robot or software agent tries to reach the given goal state of stacks of blocks on a large table from the initial state by using four operators: *pickup*, *putdown*, *stack*, and *unstuck*. These operators are software components (e.g., methods in Java) in a repository style of architecture. They are called by a human or software agent to play the blocks game. The applicability of the components depends on the current arrangement of blocks as well as the agent's state. For example, "*pick up block x*" is applicable only when block *x* is on table, it is clear (there is no other block on it), and the agent is holding no block. Once this operation is completed, the agent will hold block *x*, and block *x* is not on table, and it is not clear. These conditions form a contract between the component "*pick up block x*" and its agents. Figure 2 shows the function net. Annotation "INIT clear(1), clear(6), on(1, 3), ontable(3), ontable(6)" specifies an initial state. Similarly, annotation "GOAL [G1] clear(6), on(6, 3), ontable(3)" specifies a goal state. Goal states can be used for reachability analysis of the test model so as to find potential errors in the test model.

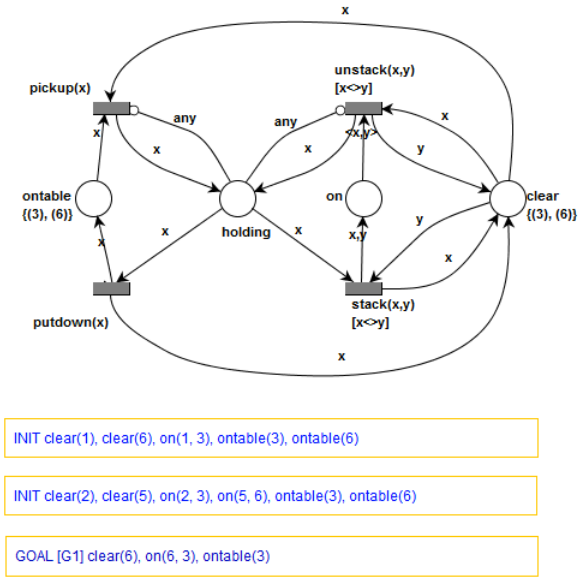

Figure 2. Function net for the blocks game

3.1 Analysis of Function Nets

MISTA provides three techniques for analyzing function nets: verification of transition reachability, verification of goal reachability, and model simulation (animation). Verification of transition reachability is to check whether there are transitions that are unreachable (not applicable) from the given initial states. Usually, all transitions in a test model should be reachable, otherwise it is likely that the test model contains certain errors. Verification of goal reachability is to check whether the given goal states are reachable from the given initial states in the test model. If the result is different from your expectation, then it is likely that the test model contains certain errors. Model simulation refers to stepwise execution of the test model from a given initial state. As it demonstrates which transitions are applicable at each state, it is very useful for debugging test models. After starting the simulation of the test model in Figure 2, MISTA presents the simulation control panel as shown in Figure 3:

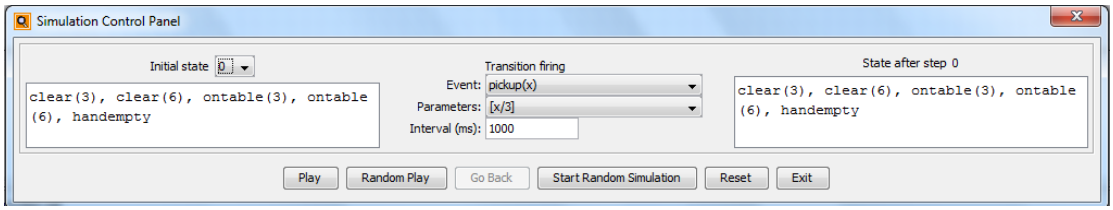

Figure 3. Simulation control panel

It shows the possible transition firings under the initial state. The user can choose a possible firing (event and variable bindings) or let MISTA to randomly pick one. Continuous random simulation is allowed at any step. The effects of transition firings are visualized in the function net. Figure 4 shows how tokens of the given state are distributed (blue dots or numbers in individual places) and which transitions are firable (in red). For example, ontable has two data items - ontable(3) and ontable(6) and pickup(3) is the only firable transition. After an enabled transition is selected to fire, the tokens and firable transitions will be updated.

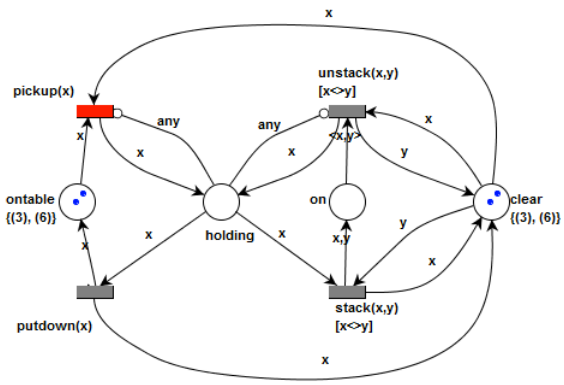

Figure 4. Simulation of the function net in Figure 2

3.2 Test Generation from Function Nets

MISTA generates test cases to meet a chosen coverage criterion. The coverage criteria for function nets are as follows:

- Reachability tree. MISTA first generates the reachability graph of the function net with respect to all given initial states and, for each leaf node, creates a test from the corresponding initial state node to the leaf node.
- Reachability + invalid paths: MISTA generates an extended reachability graph including invalid firings as additional leaf nodes. A test from the corresponding initial state to such a leaf node is called a dirty test.
- State coverage: MISTA generates tests to cover each state that is reachable from any given initial state.
- Transition coverage: MISTA generates tests to cover each transition.
- Goal coverage: MISTA generates a test for each given goal that is reachable from the given initial states.
- Random generation: MISTA generates tests in a random fashion.
- Depth coverage: MISTA generates all tests whose lengths are no greater than the given depth.
- Deadlock/termination states: MISTA generates tests that reach each deadlock /termination state in the function net. A deadlock/termination state is a marking under which no transition can be fired.

4. MIM and Helper Code for Test Code Generation

A MIM specification consists of the following:

- The identity of the SUT to be tested against the test model (keyword in the MIM sheet is CLASS, FUNCTION or URL).
- The list of hidden predicates in the test model that do not produce test code because of no counterpart in the SUT (HIDDEN).
- The list of option predicates in the test model that are implemented as system options in the SUT (OPTION).
- The calls of all components in the test model (METHOD).
- The implementation objects (numbers, symbols, strings etc.) corresponding to the objects (e.g., numbers, named constants etc.) in the test model (OBJECT).
- The accessors for non-hidden predicate in the test model (ACCESSOR).
- The mutators for no-hidden predicates in the test model (only needed when the predication is a system option or involved in initialization (MUTATOR).

The identity of the SUT is the class name for an object-oriented program, function name for a C program, or URL of a web application. In Figure 5, the class under test is *Block*. There are no hidden or option predicates. Object "*6*" in a test model is "*B6*" in the implantation. In Figure 6, component *stack(?x, ?y)* in a test model is corresponding to method *stack(?x,?y)* in the implementation. This is the same as other components (*unstuck*, *pickup*, and *putdown*) not specified. *ontable* in the test model means *isOntable*; The mutator to achieve *ontable(?x)* in the implementation is *getOntables().add(?x)*.

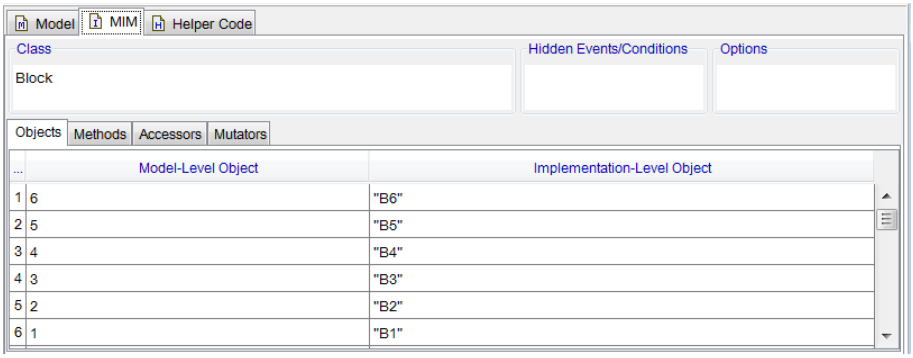

Figure 5. MIM (part 1) for the blocks game

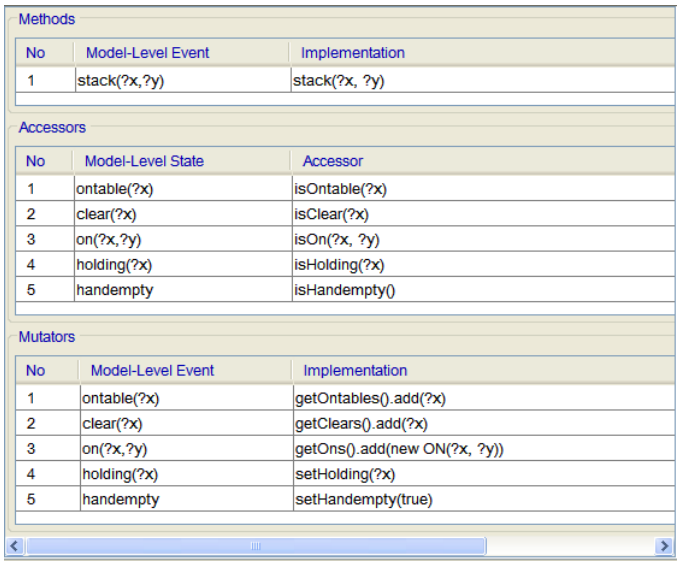

Figure 6. MIM (part 2) for the blocks game

MISTA allows the user to provide helper code in order to make test code executable. Figure 7 shows an example in Java.

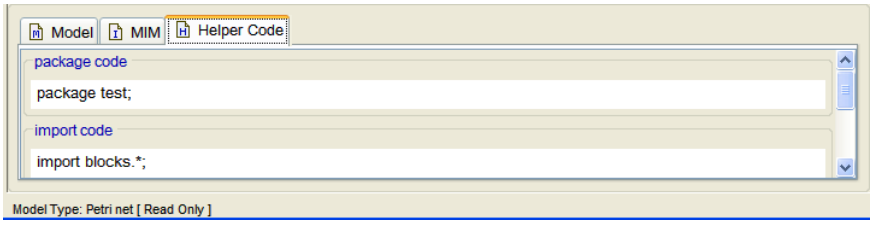

Figure 7. Helper code for the blocks game

References

1. Dianxiang Xu, A Tool for Automated Test Code Generation from High-Level Petri Nets, *Proc. of the 32nd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2011)*, LNCS 6709, pp. 308-317. Newcastle upon Tyne, UK, June 2011. Springer-Verlag Berlin Heidelberg.

Appendix A. Structure of Java Test Code