

MISTA

Model-based Integration and System Test Automation

User Manual

MISTA User Manual

© Copyright 2009-2013. All rights reserved.

Table of Contents

Chapter 1 Overview

1.1	Introduction	1
1.2	Installation.....	2
1.3	Managing MID Files	3
1.3.1	Creating a New MID File.....	3
1.3.2	Editing MID	3
1.3.3	Save.....	5
1.3.4	Save As.....	5
1.3.5	Refresh	5
1.3.6	Recent Files	5
1.3.7	Import PNML File	5
1.3.8	Import CPN File	5
1.4	Common Toolbar Buttons	6
1.5	General Workflow	6
1.5.1	Compiling MID	7
1.5.2	Simulating the Test Model	7
1.5.3	Verifying MID	8
1.5.4	Generating Test Tree.....	8
1.5.5	Generating Test Code.....	9
1.6	Notes	9

Chapter 2 Function Nets

2.1.	Elements of Function Nets.....	11
2.1.1	Places, Tokens, and Markings.....	11
2.1.2	Transitions	12
2.1.3	Arcs.....	14
2.1.4	Initial Markings, Goal Markings, and Assertions.....	15
2.1.5	Hierarchical Nets.....	17
2.1.6	Transition Firings and Test Cases	19
2.1.7	Function Nets and Finite State Machines	20
2.2	Editing a Function Net	21
2.2.1	Adding a Place or Transition.....	22
2.2.2	Editing Place Properties.....	22
2.2.3	Editing Transition Properties.....	23
2.2.4	Adding an Arc	23
2.2.5	Editing an Arc Label.....	23

2.2.6	Adding an Annotation.....	24
2.2.7	Select, Move, Delete, Copy, Paste	25
2.2.8	Editing Subnets.....	26
2.3	Editing MIM.....	27
2.4	Editing Helper Code.....	31
2.5	Compile and Error Messages	34
2.6	Simulation	36
2.7	Verification	38
2.8	Generating Test Tree.....	39
2.8.1	Test Generation Options.....	42
2.8.2	Editing Test Parameters	43
2.9	Generating Test Code	46
2.9.1	Object-Oriented Test Code	48
2.9.2	HTML/Selenium Code.....	50
2.9.3	C Test Code.....	50
2.10	Online Testing.....	52
2.10.1	On-the-fly Testing and Test Analysis	52
2.10.2	Online Online Execution of Existing Tests	55
2.11	Spreadsheet Editor for Function Nets.....	56

1.1 Introduction

MISTA is a tool for model-based software testing. It is also known as ISTA. Given a Model-Implementation Description (MID), MISTA generates executable test code in a chosen language (e.g., Java, C, C++, C#, VB, or HTML) and/or test engine (e.g., JUnit, NUnit, Selenium IDE¹, or Robot Framework²). The generated test code can then be executed against the system under test (SUT). MISTA can be used to test not only standalone programs but also web-based applications.

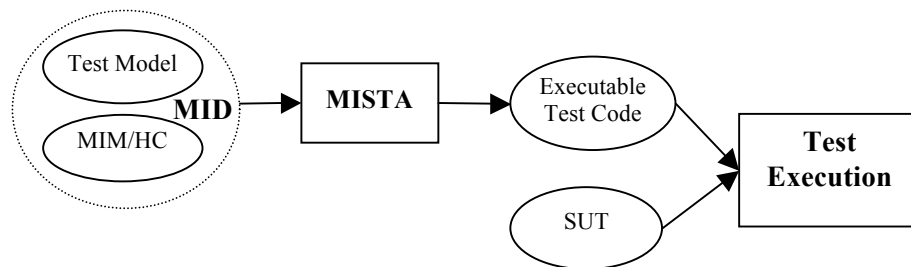


Figure 1. Test code generation with MISTA

As shown in Figure 1, a MID consists of a test model, Model-Implementation Mapping (MIM), and user-provided helper code (HC). It allows a test model together with all the required test automation code to be created, managed, and analyzed in one interface. MISTA uses function nets (lightweight high-level Petri nets) as the primary notation for specifying test models. It can generate test cases to meet a chosen coverage criterion of a test model. The coverage criteria include reachability tree coverage (all paths in reachability graph), reachability coverage plus sneak paths (negative tests), state coverage, transition coverage, depth coverage, goal coverage, deadlock/termination state coverage, random generation, and generation from given sequences. MISTA provides partial ordering and pairwise combination techniques for dealing with complex test models.

In addition to code generation for offline test execution, MISTA also supports on-the-fly testing (simultaneous generation and execution of tests) and online execution of existing tests through Selenium web driver or a RPC protocol (XML-RPC or JSON-

¹ Selenium IDE (<http://seleniumhq.org/>) is a Firefox plug for testing web applications.

² Robot Framework (<http://code.google.com/p/robotframework/>) is a keyword-based test framework.

RPC). On-the-fly testing is particularly useful for non-deterministic systems. It is easy to extend MISTA to support a new language or test engine³.

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

- Function testing and acceptance testing: MISTA can generate tests to exercise the interactions among system components and validate functions.
- 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.
- 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. In addition, MISTA can be used in test-driven development, where tests are created before the product code is written.
- Regression testing: To deal with system changes, you may update your MID files, rather than test cases or test code.

1.2 Installation

To install MISTA, unzip the file MISTA.zip and you will find an executable jar file MISTA.jar. To start MISTA, double click on MISTA.jar or use the following command from the command line: `java -jar MISTA.jar`.

If it is successful, a graphical user interface as shown in Figure 2 will show up. The dropdown menus list the supported test coverage criteria (e.g., Reachability Tree), languages (e.g., Java), and test tools (e.g., JUnit).

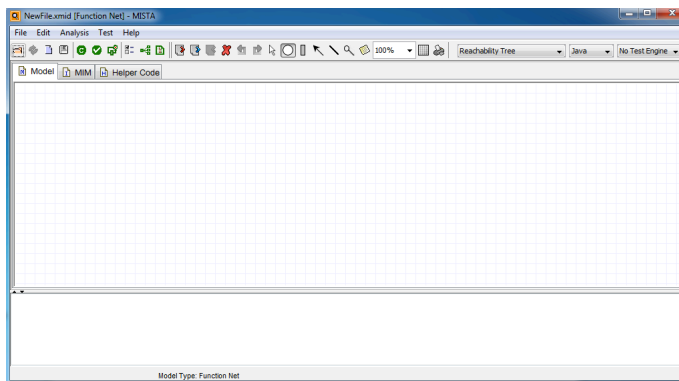


Figure 2. The GUI of MISTA.

³ Requests for new features and enhancement can be directed to the author at dianxiang.xu@gmail.com.

❏ MISTA is written in Java, a platform-independent language. It can be running on Windows, Linux, Mac OS, etc. If you cannot start MISTA, it is probably because the Java runtime environment (1.6 or higher) has not been installed properly on your computer.

1.3 Managing MID Files

1.3.1 Creating a New MID File

To create a new MID, select the menu item “New” from the “File” menu. MISTA will present an editor with an empty MID. For a particular model type, there can be multiple editors available. For example, both graphical and spreadsheet editors are provided for function nets and finite state machines. When a new MID is created, the default editor will be used. To configure the default editor, select the “Preferences” menu item from the “Edit” menu. Figure 3 shows the preferences dialog, which also includes text font settings in the MID.

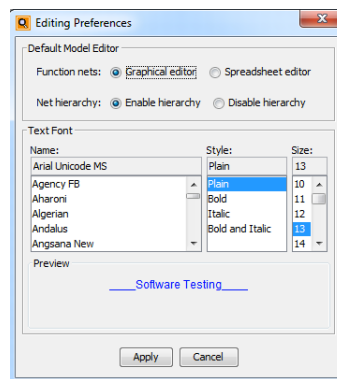


Figure 3. The preferences dialog.

❏ While other model types can be created using the graphical editor, contracts can only be edited with a spreadsheet editor.

1.3.2 Editing MID

The MID editor provides three tabs: Model, MIM, and Helper Code. For each of these tabs, the “Edit” menu has a submenu of available operations.

Model

The “Model” tab allows you to edit the test model. The editor depends on the model type. Editing test models will be detailed in subsequent chapters. Figure 4 shows a function net in the graphical editor.

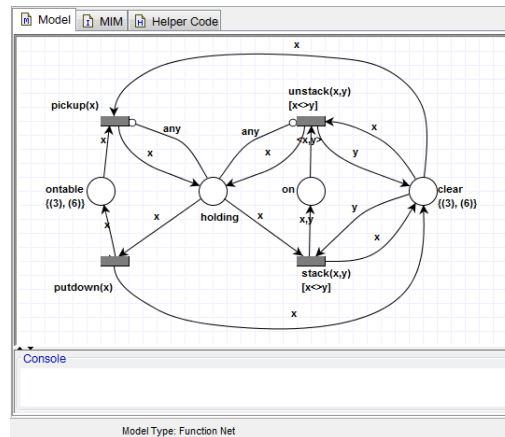


Figure 4. Function net in a graphic editor.

	Model-Level Object	Implementation-Level Object
1	6	"B6"
2	5	"B5"
3	4	"B4"
4	3	"B3"
5	2	"B2"
6	1	"B1"

Figure 5. MIM of a function net

MIM

The “MIM” tab allows you to edit the MIM of a test model like spreadsheets, as shown in Figure 5. The MIM specification maps the elements of the test model into implementation constructs for the purposes of test code generation. Generally, the MIM specification depends on the model type. More details will be described in respective chapters.


Helper Code

The “Helper Code” tab allows you to provide additional code that is necessary to make the generated test code executable. It depends on the target language. For example, Java test code generally needs package and import statements.


```
package test;


import blocks.*;
```

Figure 6. Helper code in Java

 Remember to save your changes. After finishing the editing of a spreadsheet cell, enter into a different cell, otherwise the changes may not be recognized.


1.3.3 Save

The MID being edited can be saved by either clicking on the save icon  or by selecting the “Save” menu item from the “File” menu. MID is stored in a .xmid file. When a test model is edited with a graphical editor, a separate .xml file is also created to store the test model.


 NewFile.xmid is the default file name whenever a new MID is created. You should always use a different file name for your MID.

1.3.4 Save As

This option is used to create a new copy of the current MID.

 If the test model is created through a graphical editor, you should avoid using operating system’s file “Copy/Paste” function to make another copy of a MID. In this case, MID is stored in multiple files with a special link between them.

1.3.5 Refresh

Refresh  reloads the current MID file. This is useful when it is being edited by an external editor (e.g., MS Excel).

1.3.6 Recent Files

This option allows you to open a file from the most recently used files. A file used multiple times is only listed once.

1.3.7 Import PNML File

This option in the File menu imports a PNML (Petri Net Markup Language) file into a new MID. MISTA does not provide full support for the standard PNML. Only the Petri nets that are similar to function nets can be imported. Curved arcs are also replaced with straight line arcs.

1.3.8 Import CPN File

This option in the File menu imports a CPN (Colored Petri Net) file into a new MID. MISTA does not provide full support for the standard CPN. Only the CPN nets that are similar to function nets can be imported. Curved arcs are also replaced with straight line arcs.





1.4 Common Toolbar Buttons

MISTA comes with a toolbar populated with a set of buttons. The following buttons are common to all model types. They fall into three categories: file operations, analysis, and test generation.






Figure 7. Common toolbar buttons.




File operations:

-  Open file - open an existing file
-  Refresh - refresh the current MID when it is being edited by another external editor.
-  New function net - create a new MID with function net.
-  Save file - save the working MID.

Analysis operations:

-  Compile - Compile the test model and MIM in the working MID.
-  Verify goal state reachability - verify if the given goals are reachable from any initial state in the test model.
-  Simulate - simulate the working test model.

Test generation operations:

-  Options – configure options for verification and test generation.
-  Generate test tree - generate a test tree from the working MID under current settings (e.g., test coverage criterion).
-  Generate test code – generate test code from the test tree or working MID under current settings (e.g., test coverage criterion and target language).

1.5 General Workflow

The general workflow for test code generation is as follows. First, you create, edit, save, and modify a MID. Then you compile the MID to see if there are any syntactic errors. If not, you may opt to use verification and simulation to check if there are any semantics and logic issues. If not, you can select a coverage criterion to generate test tree or test code. Then you compile and run the test code against the SUT. If the test code cannot pass the compiler, you will need to find out the reasons, modify the MID (typically MIM or helper code), and regenerate the test code. Like an iterative software

development process, many steps in the above workflow may need to be repeated. You may consider fixing the syntax errors in your IDE (Integrated Development Environment), copy the fixes back to the MID, regenerate the test code, and then recompile. Figure 8 shows the main components in the workflow. They will be detailed in the subsequent chapters.

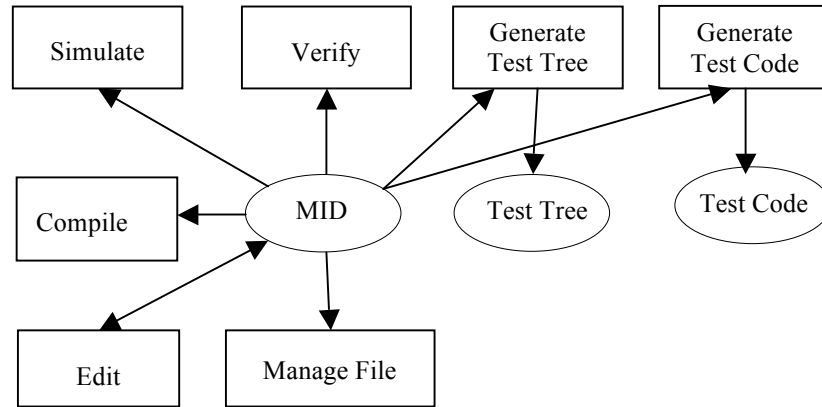





Figure 8. Main components for test generation

 The test code is always saved in one or more files in the same folder as the MID file. If you use an IDE, you may save your MID in your testing folder or package and refresh it after test code is generated.

1.5.1 Compiling MID

The Compile option is used to parse the test model and MIM. To compile, you can either click on  icon in the tool bar or select the “Compile” menu item from the “Analysis” menu. Syntactic errors will be reported in the console window.

1.5.2 Simulating the Test Model

To start stepwise execution of a test model, you can either click on  in the tool bar or select the “Simulate” menu item from the “Analysis” menu. The simulator will first call the compiler to check if there are any syntactic errors. If not, it will present a control panel as shown in Figure 9.

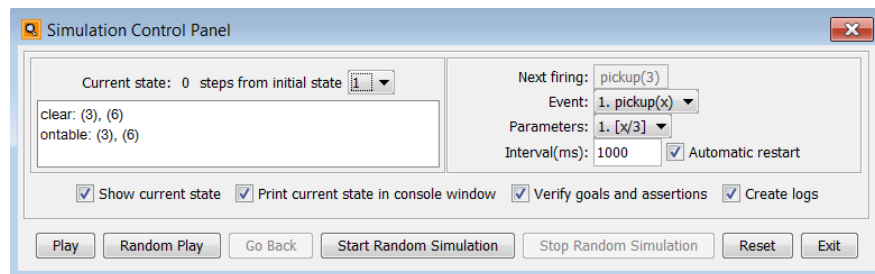


Figure 9. Simulation control panel

In the graphical editor of a function net, the simulation can be visualized: enabled transitions are in the color red and blue dots in places represent tokens (or numbers in places represent token counts). Figure 10 shows an example.

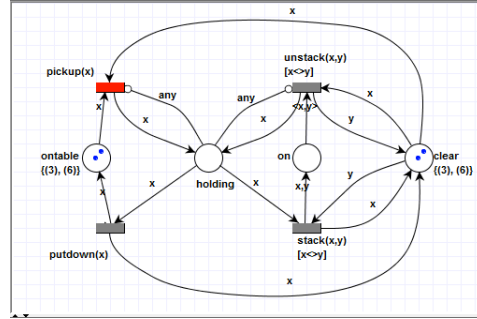



Figure 10. Simulation of a graphical function net

 Simulation of finite state machines is not supported in MISTA.

1.5.3 Verifying Test Model

A test model can be verified through reachability analysis of transitions and goal states and analysis of deadlock states. “Verification of Transition Reachability” checks to see whether there are transitions that are unreachable from the given initial states. “Verify Goal State Reachability” checks whether the given goal states are reachable from the given initial states in the test model. “Check for Deadlock/Termination States” checks to see if there are any deadlock/termination states. A deadlock/termination state refers to a state under which no transition is fireable. It does not necessarily mean the occurrence of deadlock. It can be a normal termination state.

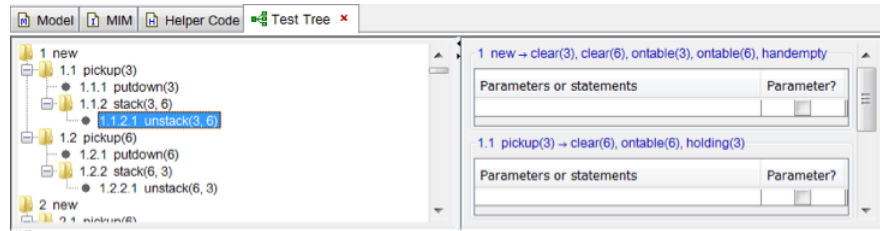





Figure 11. Sample test tree

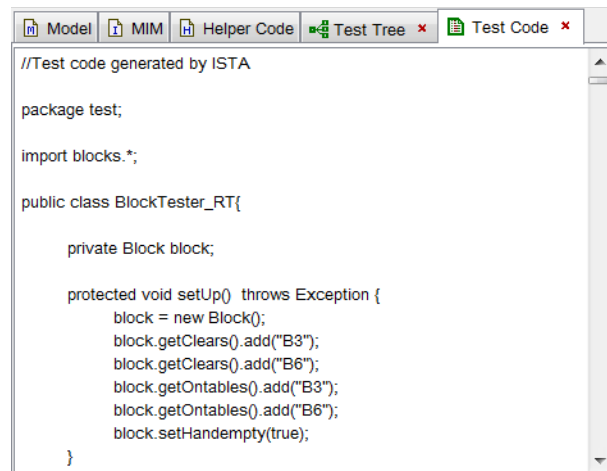
1.5.4 Generating Test Tree

To generate a test tree, you can either click on  icon in the tool bar or select the “Generate Test Tree” menu item from the “Test” menu. Before this, you may also select a test coverage criterion from the given list for the current model type. In the generated test tree, each path from a root to a leaf is a test sequence. Figure 11 shows a sample test tree, where node “1 new” and its descendants are corresponding to the tests generated for the first initial state and node “2 new” and its descendants are corresponding to the tests for the second initial state.

 If the Test Tree tab is shown in the interface, the coverage criteria dropdown menu in the tool bar is disabled, and you cannot generate another test tree before the current tree is closed.

1.5.5 Generating Test Code

To generate test code from the current test tree (if exists) or from the working MID, you can either click on  in the tool bar or select the “Generate Test Code” menu item from the “Testing” menu. If there is no test tree, the chosen coverage criterion will be used. The test code also depends on the target language and unit test framework, which can be chosen from the corresponding dropdown menus. The test code is saved under the same folder as the MID.



```
//Test code generated by ISTA

package test;

import blocks.*;


public class BlockTester_RT{

    private Block block;

    protected void setUp() throws Exception {
        block = new Block();
        block.getClearts().add("B3");
        block.getClearts().add("B6");
        block.getOntables().add("B3");
        block.getOntables().add("B6");
        block.setHandempty(true);
    }
}
```


Figure 12. Sample test code.

1.6 Notes

 You may add a new test framework for an object-oriented language by editing the configuration file Frameworks.dat. For example, the following text specifies that JfcUnit is a test framework for Java:

```
TEST      FRAMEWORK      "Java",      "JfcUnit",      "import
junit.extensions.jfcunit.*;", "extends JFCTestCase",
```


where TEST FRAMEWORK is the keyword and JfcUnit is the name of the test framework. “import...” and “extends...” define how the test framework is used in test code.


 When a task (including verification of goal reachability and deadlock/termination states, test tree generation, test code generation, and test code presentation) is time-consuming, you are allowed to cancel the task by clicking on the “Cancel” button in a progress window. Normally, the cancellation should take effect in a few seconds. If the cancellation failed, it is likely because a runtime problem (e.g., out-of-memory error) has occurred. In this case, MISTA was unable to complete the task. You can close the progress window, quit MISTA, and restart it.

This chapter describes how to use function nets for test code generation.

2.1. Elements of Function Nets

The basic elements of function nets are places (predicates), transitions, arcs, arc labels, guard conditions of transitions, and initial markings (states).

 Function nets are a simplified version of high-level Petri nets (Predicate/Transition nets and colored Petri nets). If you are unfamiliar with Petri nets, we recommend you see our documentation for concise explanations and examples rather than looking to the Petri net literature.

 The graphical editor of function nets is adapted from the open source Petri net editor PIPE3 (Platform Independent Petri net Editor, <http://pipe2.sourceforge.net/>). Although function nets are significantly different from the Petri nets in PIPE3, the editing operations are similar.

2.1.1. Places, Tokens, and Markings

A place (circle) represents a condition or state. It is named by an identifier, starting with a letter and consists of only letters, digits, dots, and underscores. Figure 13 shows the dialog for editing a place.

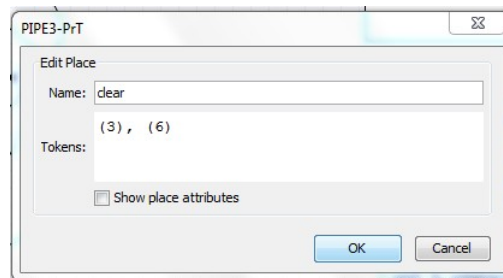


Figure 13. Editing a place.

Places can hold data, called tokens. Each token in a place p is a tuple (X_1, \dots, X_n) , where X_1, \dots, X_n ($n \geq 0$) are constants. The special token “()” with no argument is similar to the tokens in traditional place/transition nets. A constant can be an integer (e.g., 3, -2), named integer (e.g., ON) defined through a CONSTANTS annotation, string (e.g., “hello” and “-10”), or symbol starting with an uppercase letter or digit (e.g., Hello and 2hot). Multiple tokens in the same place are separated by “,”. They should be

different from each other but have the same number of arguments. A distribution of tokens in all places of a function net is called a marking of the net. In particular, if any tokens are specified in the working net, the tokens collected from all places of the net will be viewed as an initial marking. In Figure 14, both places “ontable” and “clear” have two tokens (3) and (6). These tokens form an initial marking because there is no token in any other place. Initial markings can also be specified in annotations and an Excel file (to be introduced below). Multiple initial markings can be specified for the same net.

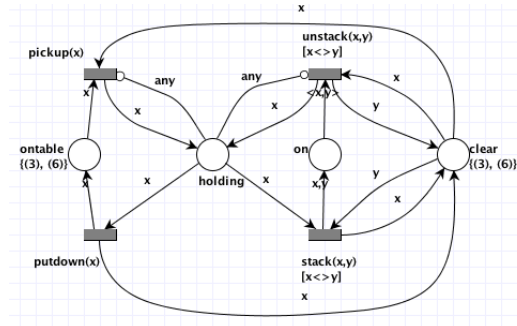



Figure 14. Tokens in places

 Markings are states of a function net (test model). An initial marking typically describes system settings and test data for test generation purposes. In this chapter, markings and states are interchangeable terms. Function nets and test models are also interchangeable.

2.1.2 Transitions

A transition (rectangle) represents an event or function. The event signature of a transition includes a name and an optional list of variables as its formal parameters. A variable is an identifier that starts with a lowercase letter or “?”. Each variable must be defined in some arc label that is connected to the transition or in the guard condition of the transition. If the list of formal parameters is not given, all variables collected from the arcs connected to the transitions will become the formal parameters. The variables are listed according to the order in which the arcs are drawn. So, if there are two or more variables from the input and inhibitor arc labels (to be discussed below), you should explicitly list the variables. If the specified list is “()”, it means that there is no formal parameter no matter how many variables appear in the input arcs.

A transition can be associated with a guard condition and an effect. Guard and effect are optional conditions. A condition is a list of predicates separated by “;”, which means logical “AND”. For example, “z=x+1, z>10”. A predicate is of the form [not] p (x1, x2 ..., xn) where “not” (negation) is optional. Figure 15 shows the dialog for editing a transition.

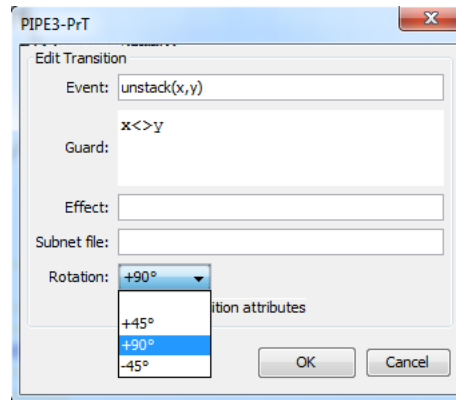


Figure 15. Editing a transition.

The guard condition of a transition can be built from arithmetic or relational predicates, where variables must be defined in the labels of arcs connected to the transition or arithmetic operations in the guard condition. Suppose x and y_i are variables or integers (including named integers and integer strings), s is a string, and exp is an arithmetic expression, which may use $+$, $-$, $*$, $/$, and $\%$ (modulus) for integer operations. The following predicates can be used in guard conditions.

- $x=exp$: If x has already been defined, it refers to comparison of x with the result of exp . If x has not occurred before, it defines a new variable x , which can be used in another predicate, such as $z>x+5$.
- $x!=exp$ or $x<>exp$: x is not equal to the result of exp .
- $x>exp$: x is greater than the result of exp .
- $x>=exp$: x is greater than or equal to the result of exp .
- $x<exp$: x is less than the result of exp .
- $x<=exp$: x is less than or equal to the result of exp .
- $isOdd(x)$: x is odd.
- $isEven(x)$: x is even.
- $belongsTo(x, y1, y2, \dots, yn)$: x belongs to the set $\{y1, y2, \dots, yn\}$. It means $x=y1$ or $x=y2, \dots$, or $x=yn$.
- $OR(s)$: s is a string that encloses two or more predicates separated by “;”, which means logical OR. $OR(s)$ is true if and only if at least one of the predicates in s evaluates to true. For example, $OR(“isOdd(x), y>2, z<x+y”)$ means that x is odd, $y>2$, or $z<x+y$.
- $bound(x)$: variable x is bound.
- $assert(p)$: place p has at least one token.
- $tokenCount(p, x)$: the number of tokens in place p is x . It is typically followed by an arithmetic operation that uses x . For example, $tokenCount(p,x), x<2$.

The predicate names, $isOdd$, $isEven$, $belongsTo$, OR , $bound$, $assert$, and $tokenCount$, are case-insensitive.

The effect of a transition provides a way to define test oracles. Each predicate in the effect can be mapped to a test oracle when tests are generated from a function net.

2.1.3 Arcs

An arc represents a relationship between a place and a transition. An arc can be labeled by one or more lists of arguments. Each argument is a variable (e.g., x and $?y$) or constant (e.g., 5, -1, ON, “Hello”, “-3”). Each list contains zero or more argument. For an unlabeled arc, the default arc label is $\langle \rangle$, which contains no argument. This arc is similar to the arcs in a place/transition net with 1 as the weight. It is important to note that the labels of all arcs connected to and from the same place must have the same number of arguments, although the variables can be different. This is because all tokens in the same place have the same number of arguments. Thus, multiple lists of labels on the same arc, separated by “&”, must have the same number of arguments. Figure 16 shows the editing of an arc label, where the arc is labeled with two lists of variables: $\langle a, b, c \rangle$ and $\langle c, d, e \rangle$.

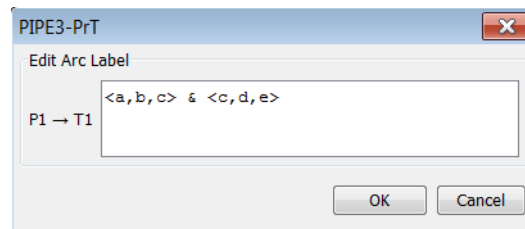



Figure 16. Editing an arc label

 An arc label may contain multiple lines of text. All arc labels are positioned automatically. Multiple lines, empty lines, or space characters in the text of an arc label can be used to help re-locate an arc label.

Arc types:

↖ Directed arc: If a directed arc is from a place p to a transition t , p is called an input arc or a precondition of t . If a directed arc is from a transition t to a place p , p is called an output arc or a postcondition of t .

↔ Non-directed or bi-directional arc: a bi-directional arc between a place and a transition is equivalent to two directed arcs with the same arc label – one from the place to the transition, and the other from the transition to the place. This arc is both an input and output arc and the place is both a precondition and postcondition of the transition. If a place is both input and output of a transition, but the transition changes the input value, you may use two directed arcs with different variables in the arc labels.

🔍 Inhibitor arc: an inhibitor arc from a place to a transition represents a negative precondition of the transition.

⚠ RESET arc is represented by a directed arc from a transition to a place with an arc label “RESET” (it is case-insensitive although uppercases are recommended). The firing of the transition will remove all tokens from the place.

Scope of variables: Variables of the same name may appear in different transitions and arc labels. The scope of a variable in an arc is determined by the associated transition. Variables of the same name refer to the same only when they are associated with the same transition. For example, in Figure 17, x in the arc from P1 to T1 and x in the guard condition of T1 are identical. However, x in the arc from P1 to T1 and x in the arc from P2 to T3 are independent.

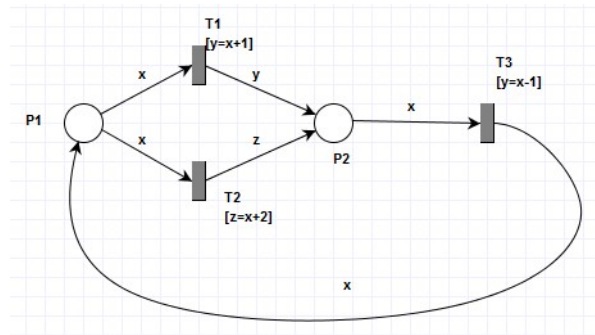


Figure 17. Scope of variables.

⚠ Place names, event names, variables, and constants are case sensitive. To avoid confusion, we recommend using different names for different purposes. For example, you should not use “p1” and “P1” to name two places or “t1” and “T1” to name two transitions in the same net.

2.1.4 Initial Markings, Goal Markings, and Assertions

In addition to specifying an initial marking by enumerating tokens in individual places, you can also specify initial markings in annotations (📝). An annotation specifies an initial marking if the text starts with the keyword “INIT”, followed by an optional name and a list of tokens separated by “,”. Here each token must be of the form $p(X_1, \dots, X_n)$, which refers to token (X_1, \dots, X_n) in place p . The zero-argument token in p is represented by p . In Figure 18, the initial marking “clear(1), clear(6), on(1,3), ontable(3), ontable(6)” specifies two tokens (1) and (6) in “clear”, one token (1,3) in “on” and two tokens (3) and (6) in “ontable”. Since an initial marking specifies a concrete state, no variables should be used.

Figure 18. Specifying initial and goal markings using annotations.


Initial markings can also be imported from an Excel file specified by a “DATA” annotation. The Excel file should be in the same folder as the xmid file. The annotation in Figure 19 will import initial markings from the Excel file BlocksNetData.xls.

Figure 19. Importing initial markings from an Excel file

In the Excel file, each spreadsheet defines one initial marking (a set of system settings and test data). Each row enumerates the tokens in one place – the first entry is the place name and each non-empty cell is a token (a tuple of data). Figure 20 shows a spreadsheet from BlocksNetData, where “clear”, “on”, and “ontable” in the first column are place names. There are two tokens (1), and (6) in “clear”, one token (1,3) in “on”, and two tokens (3) and (6) in “ontable”. This initial marking is equivalent to “clear(1), clear(6), on(1,3), ontable(3), ontable(6)”. Here, the parentheses “()” can be omitted except for zero-argument tokens. Numbers in a spreadsheet should be in the general text format. All tokens in the same place must have the same number of arguments.

	A	B	C
1			
2	clear	1	6
3	on	(1,3)	
4	ontable	3	6

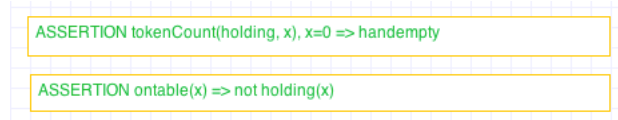
Figure 20. Initial marking in a spreadsheet

 MISTA will report syntactic errors when the net is compiled (refer to Section 2.5). To check if the initial states in a spreadsheet are specified correctly, you can view individual initial states through in the simulation control panel (refer to Section 2.6).

Properties of desirable markings (called goal states or goal properties) can be specified by annotations with the keyword “GOAL”. Goal properties can be used for semantics analysis of a test model (e.g., through reachability analysis of goal properties) or for generating tests to exercise specific states. A goal property can be a concrete marking, which consists of specific tokens. In Figure 18, the “GOAL” annotation specifies a goal marking “clear(6), on(6, 3), ontable(3)”. The name is G1. Goal names can be used

to generate tag code that indicates the points in test cases where the given goal markings have passed.

In goal properties, you can use variables, negation, and predicates (similar to those in guard conditions of transitions) to describe certain markings of interest. For example, “clear(x), ontable(y), x!=y” and “clear(x), ontable(x), not isOdd(x)” are valid goal specifications. The multiple occurrences of the same variable in the same goal specification refer to the same object.



```

ASSERTION tokenCount(holding, x), x=0 => handempty
ASSERTION ontable(x) => not holding(x)

```

Figure 21. Sample assertions

Assertions can be specified by annotations with the keyword “ASSERTION”. Each assertion is of the form: $P \Rightarrow Q$, which means, for any reachable marking, P implies Q - Q must hold whenever P holds. P and Q are a conjunction of predicates, similar to the syntax of goal properties. P is optional. MISTA can verify each assertion against the function net and report a counterexample if the assertion is not true. Figure 21 shows two sample assertions: “tokenCount(holding, x), x=0 \Rightarrow handempty” means that no token in the holding place implies that handempty is true. “ontable(x) \Rightarrow not holding(x)” means that, if block x is on table, then the robot is not holding x .

2.1.5 Hierarchical Nets

A hierarchy of function nets consists of one top-level net and its subnets. In a parent net, a transition can be linked to a child net, called subnet or submodel. For example, transition T1 in green in Figure 22 is linked to an existing subnet file abc.xml as shown in Figure 23. The subnet in abc.xml is shown in Figure 24, where transition T6 in green is linked to another existing subnet. MISTA composes a net hierarchy into one net by substituting each transition for its subnet.

The following constraints are imposed on each transition linked to a subnet:

- Each input, output, and inhibitor place of the transition in the parent net must appear in the subnet, i.e., the subnet has a place with the same name. A bi-directional place is considered as both input and output. These places are involved in the parent-child relationship. Other places in the parent net not connected to the transition are not involved in the parent-child relationship. In Figure 22, p1 is T1’s input place and p2 is T1’s output place.
- For each input place of the transition in the parent net, the corresponding place in the subnet must be used as an input place of some transition. In Figure 24, p1 is T4’s input place.

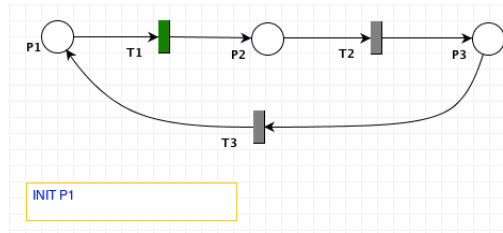


Figure 22. A top-level net of a net hierarchy

- For each inhibitor place of the transition, the corresponding place in the subnet must be used as an inhibitor place of some transition.
- For each output place of the transition, the corresponding place in the subnet must be used as an output place of some transition. In Figure 24, p2 is T4's output place.
- For each place connected by a bidirectional arc in the parent net, the corresponding place in the subnet must be connected by a bidirectional or used as input place of some transition and output place of some transition.

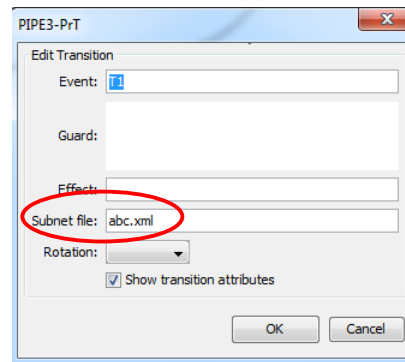


Figure 23. Linking a transition to a subnet

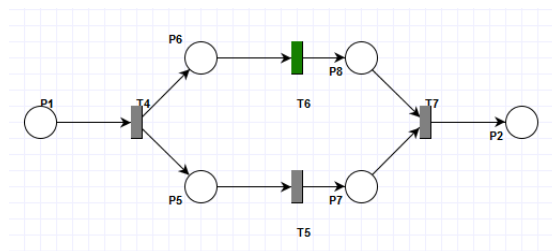





Figure 24. A subnet in a net hierarchy


- The first initial (goal) marking of each subnet will be merged with the first initial (goal) marking of the top-level net. The first initial marking is either defined in an INIT annotation or specified by tokens in individual places. Other initial (goal) markings in subnets are ignored. If you need multiple initial

(goal) markings for the composed net of a net hierarchy, specify them in the top-level net.

 A green transition indicates that it is linked to an existing subnet file. A yellow transition indicates that the transition is linked to a subnet file that does not exist yet.

 If you want to use only the top-level net of a net hierarchy, you may disable the hierarchy through the “Preferences...” dialog of the “Edit” Menu.

 A subnet file cannot be linked to more than one transition in a net hierarchy.

 In a net hierarchy, transitions in different subnets should have different names. Places in different subnets should also have different names, except for those used in a parent-child relationship.

2.1.6 Transition Firings and Tests

A transition is said to be firable or enabled under a marking if there is a match value for each variable involved in the input arc, inhibitor arc, and guard condition such that

- Each input place has a token that matches the arc label,
- Each inhibitor place does not have any token that matches the arc label, and
- The guard condition evaluates to true.

In Figure 25, the initial marking is $\{p1(1,1), p1(1,2), p2(2)\}$, where $p1(1,1)$ represents token $(1,1)$ in place $p1$. Consider transition $t1$. Its input place $p1$ has tokens $(1,1)$ and $(1,2)$ and the input arc label from $p1$ to $t1$ is (x, y) . The inhibitor place $p2$ has token (2) and the inhibitor arc label is (x) . The guard condition is $x \neq y$. If $x=1$ and $y=2$, then $t1$ is firable because token $(1,2)$ in $p1$ matches arc label (x,y) , any token in $p2$ does not match arc label (x) , and the guard condition $x \neq y$ is true. If $x=1$ and $y=1$, however, $t1$ is not firable because the guard condition $x \neq y$ is false although token $p1(1,1)$ matches the arc label and there is no token $p2(1)$.

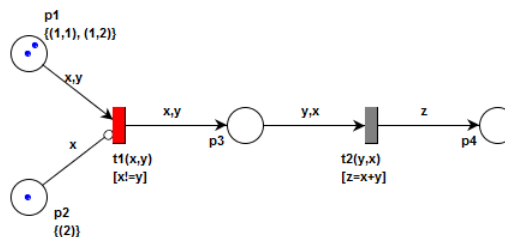




Figure 25. Transition firing in function nets

Firing an enabled transition removes the matched token from each input place and adds a token to each output place according to their arc labels and variable values. Therefore, it leads to a new marking. If we fire t_1 with $x=1$ and $y=2$ in Figure 25, token $(1, 2)$ will be removed from p_1 . A new token $(1,2)$ is added to p_3 because p_3 is an output place of t_1 and the corresponding arc label is (x, y) . Therefore, the new marking is $\{p_1(1,1), p_2(2), p_3(1,2)\}$. If a place is connected to the fired transition by a RESET arc, all tokens in the place are removed. Note that transition firing does not remove any token from an inhibitor place.

 The default arc label (no label or arc label with no argument) matches the non-argument token \emptyset .

A goal marking is said to be reachable from an initial marking if there is a sequence of transition firing that results in a marking that includes each token in the given goal marking. Suppose $\{p_1(1,1), p_2(2), p_4(3)\}$ is a goal marking. It can be reached from the initial marking $\{p_1(1,1), p_1(1,2), p_2(2)\}$ in Figure 25 by the following sequence of transition firings: t_1 with $x=1$ and $y=2$ and t_2 with $y=2$, $x=1$, and $z=3$. The firing of t_1 with $x=1$ and $y=2$ can be denoted as $t_1(1,2)$ because t_1 's arguments are x and y . It is like a method or function call with actual parameters.

In MISTA, a clean (dirty) test is a valid (invalid) sequence of transition firings from an initial marking, where each transition firing is a test input (e.g., $t_1(1,2)$) and its resultant marking is a set of test oracles (each token in the marking is a test oracle). MISTA provides different strategies for generating tests. In a generated dirty test, the last transition firing is invalid (not firable under the marking before the transition firing), whereas all transitions firings before it are valid.

 The simulator is very useful for understanding transition firings, firing sequences, and tests.

2.1.7 Function Nets vs Finite State Machines

Function nets are a super set of finite state machines. A function net reduces to a finite state machine if (a) each transition has at most one input place and at most one outplace, (b) all arcs use the default arc label, and (c) each initial marking has one token at only one place.

It is easy to represent a finite state machine by a function net. Suppose $(s_i, e [p, q], s_j)$ is a transition in a finite state machine, where s_i is the source state, e is the event, s_j is the destination state, p is the guard condition, and q is the postcondition. For each of such transitions, you can create a place s_i (if it does not yet exist), a place s_j (if it does not yet exist), and a transition with event e , guard condition p , and effect q . s_i is the transition's input place and s_j is the transition's output place. If $s_i=s_j$, then s_i is both input and output place (i.e., there is a bi-directional arc between s_i and the transition). Figure 26

shows a sample finite state machine, where circles and arcs represent states and transitions, respectively. Figure 27 shows the corresponding function net. For example, transition (INACTIVE, engineOn, ACTIVE) in Figure 26 is corresponding to transition engineOn with INACTIVE as the input place and ACTIVE as the output place in Figure 27. Transition (ACTIVE, brake, ACTIVE) in Figure 26 is corresponding to transition brake that is connected with place ACTIVE by a bidirectional arc in Figure 27.

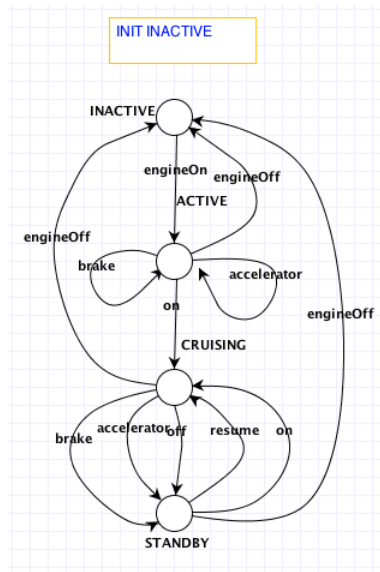


Figure 26. A finite state machine

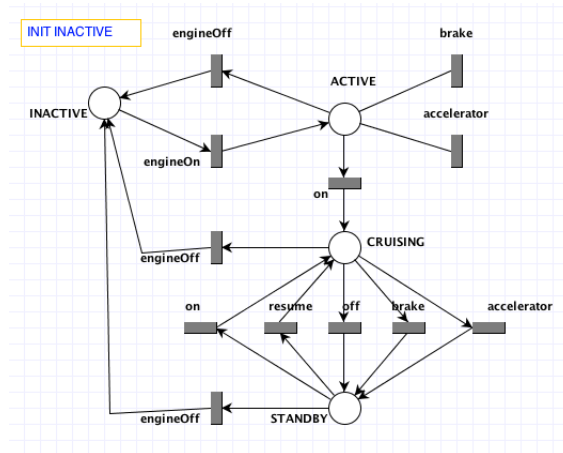


Figure 27. Function net corresponding to a finite state machine
















2.2 Editing a Function Net

The graphical editor provides an editing toolbar as shown in Figure 28:





Figure 28. Tools for editing a function net.

Icons of the graphical editor

-  Cut - Remove selected elements (arcs, places, transitions and annotations).
-  Copy - Copy currently selected elements. It is similar to cut, except that the elements will not be removed.
-  Paste - Paste the last copied.
-  Delete – Delete the currently selected portion form the editor.
-  Undo - Undo the changes done to a function net.
-  Redo - Redo the changes done to a function net.
-  Select - Select one or more elements.
-  Add a place.
-  Add a transition.
-  Add a directed arc.
-  Add a bidirectional arc.
-  Add an inhibitor arc from a place to a transition.
-  Add an annotation.
-  Change the grid size.
-  Print.

2.2.1 Adding a Place or Transition

To add a place or transition, you first click on  or . Then you can add a place or transition by one left-click at the destination location. A default name will be used for the new place or transition.

2.2.2 Editing Place Properties

To edit the properties of a place, including its name and tokens in an initial marking (refer to Figure 16 on page 12), you may either double-click on the place or right-click on the place and select “Edit place” from the popup menu.

2.2.3 Editing Transition Properties

As shown in Figure 17 on page 13, the properties of a transition include event signature (name and a list of parameters), guard, effect, and rotation. To edit transition properties, you may double-click on the transition or right-click on the transition and select “Edit transition” from the popup menu. You may rotate a transition by +45, +90 or -45 degrees.

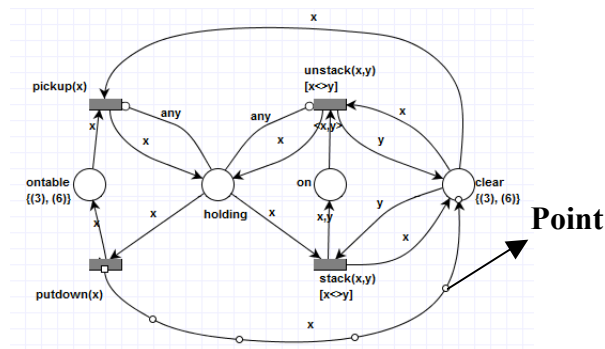



Figure 29. Points in a curved arc.

2.2.4 Adding an Arc

To add an arc, you first need to select your arc type (directed ↗, bi-directional ↔, or inhibitor ⊘) from the tool bar. Then click on the source place (or transition), drag the pointer towards the destination transition (or place), and then release at the destination. An inhibitor arc can be drawn from a place to a transition, but not from a transition to a place. By default, an arc uses a straight line. You can draw a curved arc by holding the “Shift” key when you are creating it. In the drawing process, you can insert a point by a left click at each point of interest. Figure 29 shows a curved arc with four points.

You can also convert a straight line arc to a curved one by inserting points into the arc. To insert a point, you need to right-click on the arc and choose “Insert point” or “Split arc segment” from the popup menu.

 You may quickly draw a sequence of places, transitions, and directed arcs as follows: (1) choose place or transition to start with by clicking on the respective icon, (2) locate the cursor at the desired place, (3) press and hold the “control” key, (4) click to create a place (or transition), (5) drag to create a directed arc, (6) click to create a transition (or place). (4)-(6) can be repeated until the “control” key is released.

2.2.5 Editing an Arc Label

To edit an arc label, double-click on the target arc or right-click on the target arc and select “Edit arc” from the popup menu.

Constants can be used in arc labels. In Figure 30, 1 is the label of the arc from P1 to T1. The net in Figure 30 is equivalent to that in Figure 31, where 1 is replaced with variable x and $x=1$ is added to the guard condition. Using constants in arc labels, when possible, is preferred.

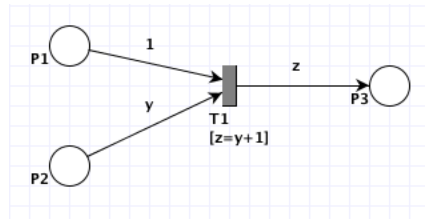


Figure 30. Constants in arc labels

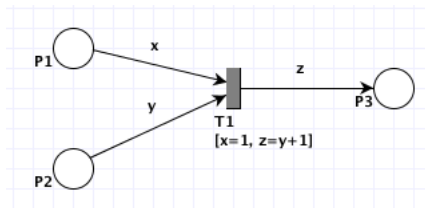


Figure 31. Arc label and guard condition

2.2.6 Adding an Annotation

Annotations are used to specify initial states, goal states, global places, constants (named integers), comments, and so on. Information on initial and goal states can be found in Section 2.1.4.

A GLOBAL annotation starts with the keyword “GLOBAL”, followed by a list of predicates. Multiple predicates are separated by “,”. Each predicate is of the form $p(x_1, \dots, x_n)$. It means that there is a bi-directional arc between place p and each transition and the arc is labeled by (x_1, \dots, x_n) . The purpose of global annotations is to make test models more readable when there are global places.

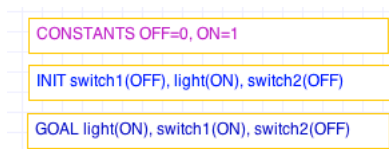


Figure 32. Named integers

A CONSTANTS annotation defines a list of named integers separated by “,” such as OFF=0, ON=1 in Figure 32. The named constants can be used in tokens, arc labels, guard conditions, initial markings, and goal markings. In particular, they can be used in arithmetic predicates of guard conditions, such as $x_2 = ON - x_1$, $y_2 = ON - y_1$ in Figure 33. The resultant value will be translated into a named constant if possible. For example, if

x_1 is OFF (0), then $x_2 = \text{ON} - x_1$ is 1. The result will be translated into ON. You should avoid using different names to represent the same value.

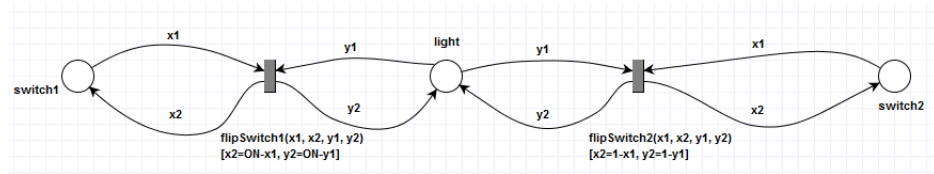



Figure 33. Two-switch light

Similar to a CONSTANTS annotation, an ENUMERATION annotation defines list of non-negative integers, starting from 0. For example, “ENUMERATION OFF, ON” is the same as “CONSTANTS OFF=0, ON=1”. You should not provide more than one ENUMERATION annotations.

A sequence annotation starts with the keyword “SEQUENCE”, followed by the name of a text file, which contains sequences of events used for test generation purposes.

Annotations are also used to provide textual descriptions about the function net. If an annotation does not contain a keyword (e.g., INIT, GOAL, GLOBAL), the text and its frame use the black color.

2.2.7 Select, Move, Delete, Copy, Paste

Select elements: you may select an individual element by left-click on the target element. You may also select multiple elements as follows: click on the  icon, locate the pointer and drag. The selected elements will be highlighted. Then you can move, copy, and delete selected elements. Figure 34 shows an example.

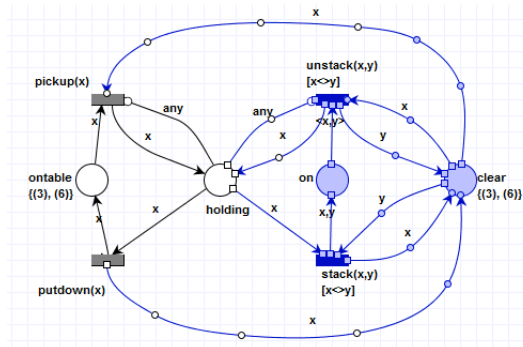




Figure 34. Select elements.

Move selected elements: you may move the selected elements by dragging and then releasing at the destination location.

Delete selected elements: you may delete the selected elements by clicking on  .

Copy and paste selected elements: you may copy and paste the selected elements by clicking on  and , respectively. Default names will be used for the new places and transitions.

2.2.8 Editing Subnets

If you intend to create a subnet of a newly created top-level net (which has the default file name NewFile.xml), we recommend that you save the top level net into a non-default file before you start creating a subnet. Subnets are usually saved under the same directory as the top-level net.

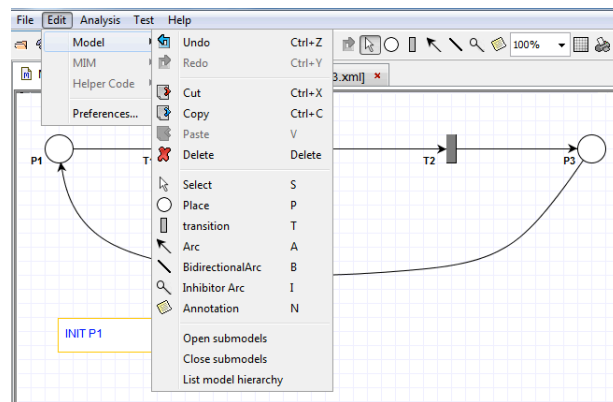


Figure 35. Sub-model operations

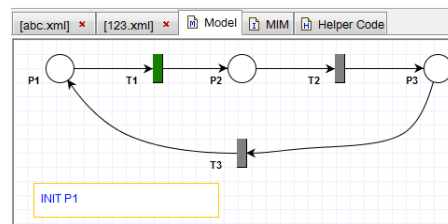





Figure 36: Opening a MID file with a net hierarchy



You may use the following steps to add a subnet to a transition:

- Double click on the transition to trigger the editing dialog (refer to Figure 23).
- Provide a valid xml file name of the subnet, no matter whether the file exists or not (refer to Figure 23). After the file name is specified, the transition will go yellow if the file does not exist. It would go green if the file already exists.
- Select “Open submodels” from the “Model” submenu of the “Edit” menu, as shown in Figure 35. It opens all nets in the hierarchy.
- Find a (new) tab named after the xml file.

- Edit the subnet in this tab.
- Save the subnet by saving the MID file or closing the subnet tab to trigger the “Save changes” dialog. The parent transition will go green after the subnet is saved.

When you open a MID file with a net hierarchy, all subnets will be presented in respective tabs, as shown in Figure 36. The “Model” tab is corresponding to the top-level function net.

-  You may copy/paste net objects across different nets in a net hierarchy.
-  The global predicates specified by a “GLOBAL” annotation in a subnet apply only to the transitions in the subnet.
-  “CONSTANTS” and “ENUMERATION” annotations can be used in subnets.

-  It is possible that the top-level net and subnet files in a net hierarchy are from different directories. Particular paths can be specified for subnets. However, it is recommended that all nets in the same net hierarchy should be saved in the same directory. Otherwise, you would probably need to manage the files through the file utilities of your operating system.
-  The “Save As” operation does not change the file names of the subnets. To rename subnet files, you need to use the file utilities of your operating system. In this case, you can edit a transition to change its subnet file name and bring up the tab for the new subnet file by “Close submodels” and then “Open submodels”.

2.3 Editing MIM

The MIM for a function net includes the following optional elements:

Identity of the SUT: It can be a class name for an object-oriented program, a function name of a C program, or a URL of a web application. It is not used when the target platform is Robot Framework.

Hidden events/conditions: This optional area specifies a list of transitions (events) and places that will not produce test code. All events and places listed must be defined in the test model. Multiple events and places are separated by “;”. You may right-click to bring up the list of events and places in the function net, as shown in Figure 37.

After you have selected one or more events and places from the list, click on the “Apply” button. MISTA will translate your choices into text.

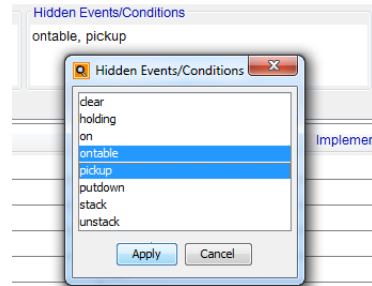


Figure 37. A dialog for selecting hidden events and conditions

Options: This optional area specifies a list of places that are used as system options and settings. Such an option in a test often needs to be set up properly through some code, called mutator. The places listed must have been defined in the function net. You may right-click to bring up the list of places in the function net, as shown in Figure 38. After you have selected places from the list, click on the “Apply” button. MISTA will translate your choices into text.

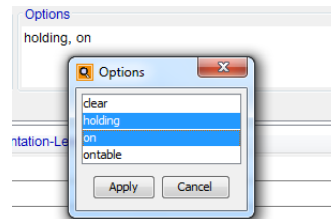


Figure 38. A dialog for selecting options

Objects: This optional spreadsheet maps constants in all tokens of the function net (called model-level objects) to the objects (numbers, strings, named constants, etc.) in the target SUT. If a constant in the function net is not mapped, it will remain the same in the test code. For example, we may use JavaBlocks as a constant in a test model. In the implementation or test code, it can be the following named constant in the SUT or helper code:

```
static final String JavaBlocks= "..\\examples\\java\\blocks \\JavaBlockNet.xls";
```

Figure 39 shows an example of object mapping, where the constant 6 in the function net is mapped to the string “B6” in the SUT. When editing in a model-level object cell, right-click will trigger a popup menu that lists all the constants defined in the transitions, initial states, and goal states of the function net if the function net does not have errors. You may or may not choose a constant from the popup menu. The constants may not be listed correctly if the function net has errors.

Objects			Methods	Accessors	Mutators
No	Model-Level Object		Implementation-Level Object		
1	6	Edit Cell in New Window	"B6"		
2	5		"B5"		
3	4		"B4"		
4	3		"B3"		
5	2		"B2"		
6	1		"B1"		
7					

Figure 39. An object-mapping example

Methods: This optional spreadsheet maps individual events of the function net (**called model-level events**) to a block of code in the SUT. If an event is not mapped and not listed in the hidden events/conditions, it will remain the same in the test code. Each event specified here is of the form $e(?x_1, \dots, ?x_m)$, where e is the name and $(?x_1, \dots, ?x_m)$ are parameters. The parameters $(?x_1, \dots, ?x_m)$ are corresponding to the transition's formal parameters in the function net but the names are independent. The number of parameters must be the same as that in the corresponding event signature in the function net. The parameter names $(?x_1, \dots, ?x_m)$ are used as placeholders in the specified block of code for the event (you do not have to use them, though).

Objects			Methods	Accessors	Mutators
No	Model-Level Event		Implementation Code		
1	stack(?x, ?y)	Edit Cell in New Window pickup(?x) putdown(?x) stack(?x, ?y) unstack(?x, ?y)	stack(?x, ?y)		
2					
3					
4					
5					
6					
7					

Figure 40. An example of method mapping

Figure 40 shows an example of method mapping. When editing in a model-level event cell, right-click will trigger a popup menu that lists all the events and their signatures defined in the function net if the function net does not have errors. You may choose a model-level event from the popup menu. The events may not be listed correctly if the function net has errors.

Suppose the actual event occurrence in a test is $e(X_1, \dots, X_m)$ and constants (X_1, \dots, X_m) in the function net are mapped into implementation objects (O_1, \dots, O_m) . The test code of this event occurrence is corresponding to the specified block of code where $?x_1, \dots, ?x_m$ are replaced with O_1, \dots, O_m , respectively. Thus, it is recommended that that parameters start with $?$. For example, the method for $\text{parse}(?x)$ in a test model may be corresponding to `mainframe.getFileManager().parse(?x)`.

Accessors: This optional spreadsheet maps parameterized tokens or places (**called model-level states**) into a block of code that typically verifies the state of the SUT. If a token is not mapped and its place name is not listed in the hidden events/conditions, it will remain the same in the test code. Each model-level state specified here is of the form $p(?x_1, \dots, ?x_m)$, where p is the place name and $(?x_1, \dots, ?x_m)$ are parameters. The

parameter names ($?x_1, \dots, ?x_m$) are independent of the variables in the function net. However, the number of parameters must be the same as the number of arguments of the place (i.e., number of arguments in associated arc labels) in the function net. The parameter names ($?x_1, \dots, ?x_m$) are used as placeholders in the specified block of accessor code.

Figure 41 shows an example of accessor mapping. When editing in a model-level event cell, right-click will trigger a popup menu that lists all the places (together with required number of arguments) defined in the function net. The places may not be listed correctly if the function net has errors.

Objects		Methods		Accessors		Mutators	
No	Model-Level State				Implementation Accessor		
1	ontable(?x)				isOntable(?x)		
2	clear(?x)				x)		
3	on(?x,?y)				?y)		
4	holding(?x)				(?x)		
5					on(?x, ?y)		
6					ontable(?x)		

Figure 41. An example of accessor mapping

Suppose the actual token in a test is $p(X_1, \dots, X_m)$ and constants (X_1, \dots, X_m) in the function net are mapped into implementation objects (O_1, \dots, O_m) . The actual accessor code is corresponding to the specified block of code where $?x_1, \dots, ?x_m$ are replaced with O_1, \dots, O_m , respectively. It is recommended that that parameters start with $?$. For example, the accessor for $tree(?x, ?y)$ in a test model may be corresponding to a query function $isValidTree(?y)$ defined in the SUT or helper code:

```
private boolean isValidTree(CoverageCriterion coverage) {
    TransitionTree tree = mainframe.getFileManager().getEditor().getTransitionTree();
    if (tree==null)
        return false;
    if (coverage!=SystemOptions.ReachabilityTreeWithDirtyTests)
        return false;
    return true;
}
```

Mutators: This optional spreadsheet maps tokens (i.e., **model-level states**) into a block of code that achieves the desired state of the SUT. The syntax is the same as that for accessors. Mutators are typically used for places that are listed as options. A token in an option place in a function net will be transformed into mutator code. The transformation is similar to that of accessor code. For example, we can define the mutator of $coverage(?x)$ as `mainframe.getToolBar().setCoverageCriterion(coverage);`

The spreadsheet operations are described in the following.

Edit a cell: you can double-click on a cell to edit it. Each cell may have multiple lines. After finishing editing a cell, you should exit the cell so that the changes will be recognized. To edit a cell in a separate window when the cell is being edited, right-click

on the cell and select “Edit cell in New Window” from the popup menu. A new window as shown in Figure 42 will be presented.

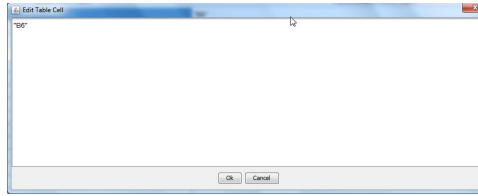


Figure 42. Edit a cell in a separate window.

Select a row: you can select a row by clicking on the target row.

Delete the selected row: to delete the selected row, right-click on the spreadsheet and select “Delete Row” from the popup menu as shown in Figure 43.

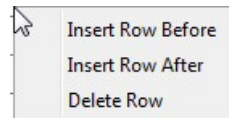



Figure 43. Popup menu for inserting and deleting a row.

Insert a row: to insert a row, right-click on the spreadsheet and select “Insert Row Before” or “Insert Row After” from the popup menu as shown in Figure 43.

 In Methods/Accessors/Mutators, you may use multiple rows for the same model-level event/state. MISTA will put them together in the order they are specified. If your target language is an object-oriented language or C, we recommend you use a single row for each model-level event/state. If your target language is HTML/Selenium IDE, each row can only be corresponding to one Selenium command. If you map a model-level event/state to multiple Selenium commands, you have to put each of them in a separate row. Because MISTA’s spreadsheet editor is not yet powerful, you may use Excel to edit MIM, which is the second sheet after the MID file is opened with Excel. In this case, you may use the refresh button or menu item to load the changes into MISTA.

2.4 Editing Helper Code

In general, the helper code may include the header (for non-web applications), alpha/omega segments, setup/teardown methods, and local code (code segments, for non-web applications). Figure 44 presents an example in Java.

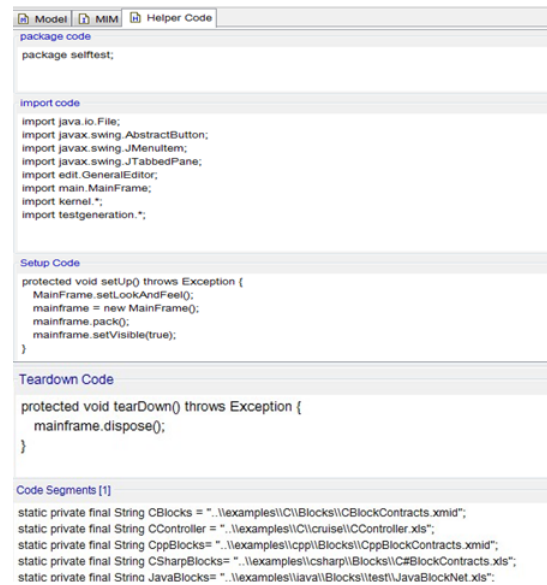


Figure 44. Helper code tab of a function net.

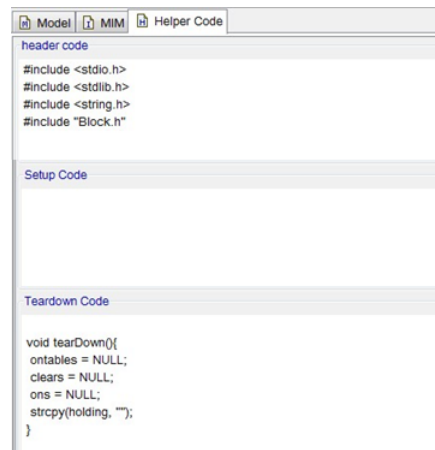
- 📖 **Header code:** Header refers to the code defined at the beginning of a test program. In Java, the header includes “package and import statements”, whereas in C#, it includes namespace and using statements. HTML/Selenium test code for web applications does not need the header. For Robot Framework, the header code refers to “settings”.
- 📖 **Setup code:** It is a piece of code called at the beginning of each test case.
- 📖 **Teardown Code:** It is a piece of code called at the end of each test case.
- 📖 **Alpha code:** It is a piece of code called at the beginning of a test suite.
- 📖 **Omega code:** It is a piece of code called at the end of a test suite.
- 📖 **Code Segment:** A code segment refers to a block of code to be included the generated test code. This code can be declarations of constants, variables, and methods/functions, which can be referenced in the MIM.

 It is important to select the right target language before editing the helper code.

Generally the components of helper code depend on your target language.

- **Java:** package code, import code, setup code, teardown code, alpha code, omega code, and code segments.

- **C/C++:** header code, setup code, teardown code, alpha code, omega code, and code segments. Figure 45 shows a C example.
- **C#:** namespace code, using code, setup code, teardown code, alpha code, omega code, and code segments.
- **PHP:** header code (include statements), setup code, teardown code, alpha code, omega code, and code segments.
- **VB:** name space, imports code, setup code, teardown code, alpha code, omega code, and code segments.
- **HTML/Selenium:** setup code, teardown code, alpha code, and omega code. They need to be provided as HTML source code. Typically they are copied from existing tests recorded by Selenium IDE.
- **Robot Framework:** only settings (as a header) will be used. Setup and teardown methods are defined outside MISTA.



```

header code
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Block.h"

Setup Code

Teardown Code

void tearDown()
{
    ontables = NULL;
    clears = NULL;
    ons = NULL;
    strcpy(holding, "");
}

```

Figure 45. Helper code in C.

If your target language is an object-oriented language (Java, C++, C#, VB) or C and you have not defined the setup method/function, MISTA will generate it. The signature of the setup method/function is as follows:

void setUp() for Java, C++, and C

SetUp() for C# and VB

If you define your own setup method/function, use the same signature.

Teardown is not called in test cases unless you define the teardown method/function. The teardown method/function should use the following signature:

void tearDown () for Java, C++, and C

TearDown() for C# and VB

2.5 Compile and Error Messages

You can compile your MID to see if there are any syntax errors, i.e., violations of syntactic constraints. MISTA will report a message once it has detected an error. For example, the test model in Figure 46 has a syntax error in “P-1”, which is used as a place name. “P-1” is an illegal identifier because it cannot contain “-”. An identifier should start with a lower-case letter and consists of letters, digits, “_” and “.”.

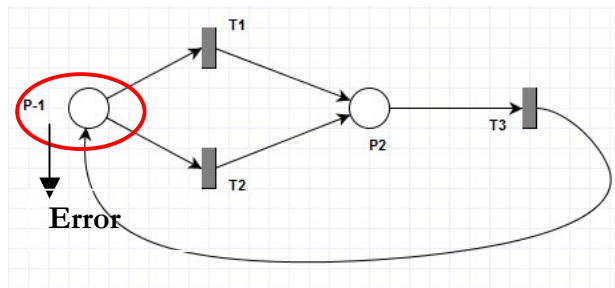


Figure 46. Incorrect place name.

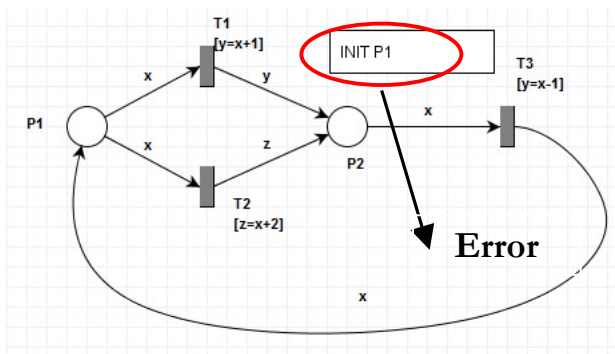


Figure 47. Inconsistent number of arguments.

Figure 47 shows another type of syntax error. Place “P1” has one argument according to the arcs from P1 to T1 and T2 (both arcs are labeled by x). In the initial state annotation “INIT P1”, P1 has no argument. Compiling the function net in Figure 47 will result in an error message “inconsistent number of arguments”. When a place used as an input/inhibitor/output place of a transition has an inconsistent number of arguments, the error message may also point out the involved transition. In such an error message, “precondition” of a transition refers to input places and inhibitor places, whereas “postcondition” refers to output places. A place connected to a transition by a bidirectional arc is considered both input and output place.

Figure 48 shows a sample error in MIM. In a spreadsheet for objects, methods, accessors or mutators, both model-level and implementation-level entries should be

specified. In row 1, the implementation-level object is not given for model-level object “6”. This is a syntax error.

Model				MIM		Helper Code			
Class				Hidden Events/Conditions				Options	
Block									
Objects									
Methods		Accessors		Mutators					
No		Model-Level Object				Implementation-Level Object			
1		6							
2		5				"B5"			
3		4				"B4"			
4		3				"B3"			
5		2				"B2"			
6		1				"B1"			

Figure 48. Sample error in MIM.


The following are additional syntactic constraints:

- All arc labels associated with the same place must have the same number of arguments.
- In initial markings and goal markings, the place names must appear in the function net. The arguments of tokens may not contain variables. The tokens in the same place must have the same number of arguments as that of all arc labels associated with the place.
- In the “Methods” section of MIM, each “model-level event” is corresponding to an event and its arguments. The event must appear in the function net. The number of arguments must be the same as the number of the parameters defined for the event in the function net.
- In the “Accessor” and “Mutators” section, each “model-level state” is corresponding to a place and its arguments. The place must appear in the function net. The number of arguments must be the same as defined in the function net.



A warning message, not error message, is reported when the same transition name appears in different nets of a net hierarchy or the parent and child nets share a place name that is not involved in the parent-child relationship. In this case, check your net hierarchy to fix the problem or make sure it is what is intended.

2.6 Simulation

You may simulate your function net to see if it operates as expected. To bring up the simulation window as shown in Figure 49, either click on  in the tool bar or select the “Simulate” menu item from the “Analysis” menu. The control panel consists of the following components.

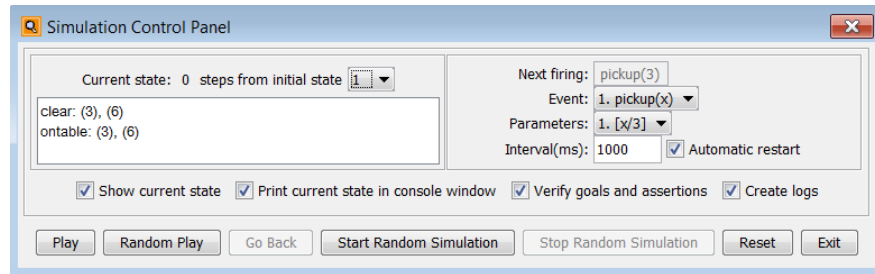


Figure 49. Simulation control panel

Current State and Initial State: This allows you to select which initial state is used for simulation in case multiple initial states are specified. The text area presents the current state after a number of firings starting from the selected initial state. If no firing is performed (i.e., the number of steps is 0), the current state is the selected initial state.

Transition Firing

- Next firing: Currently selected firing (event and parameters).
- The Event section allows you to select a transition (event) that can be fired at current state.
- The Parameters section allows you to select the actual parameters for the selected event.
- The Interval (ms) allows you to configure the time interval between two consecutive firings. By default, it is set at 1 second (1000 milliseconds).
- Automatic restart: This applies only to automated random simulation. When no transition is fireable under the current state or the simulation has reached the maximum number of steps, it will restart the simulation from a randomly selected initial state.

Checkbox Options: This shows the current marking after previous transition firing.

- Show current state: this checkbox indicates whether the simulation control panel presents the current state. It can be unchecked to save space for presenting complete events and parameters when events have a large number of parameters or lengthy arguments.

- **Print current state in console window:** This indicates whether or not the console window prints the current state after the firing is performed. The console window always prints each firing. So it contains the entire firing sequence.
- **Verify goals and assertions:** When goals and/or assertions are specified in the net, this option will verify whether each goal is reached and whether each assertion is violated at each simulation step. The integrated simulation and verification can be performed no matter how large the net's state space is.
- **Create logs:** This indicates whether or not the next firing sequence will be logged into a file. This checkbox is enabled only when the current state is an initial state. You can start a new firing sequence by clicking on the "Reset" button or selecting a different initial state. The log file will be created when you exit the simulation session. The logged firing sequences do not include the firings that are undone by "Go back".

Play: This is to perform the firing selected by the user.

Random Play: This is to fire a transition randomly selected from the given lists of firable events and parameters.

Go Back: This option allows you to go back one step at a time.

Start Random Simulation: This is similar to Random Play, but once it starts, it continues until the Stop Random Simulation button is clicked or, when the "Automatic restart" checkbox is not checked, no transition is enabled at the current state or the maximum number of steps is reached.

Stop Random Simulation: This stops the execution of the Random Simulation initiated by Start Random Simulation. If you select Start Random Simulation again, it will start where it left off.

Reset: It resets the simulation to the selected initial state.

Exit: This terminates the simulation.

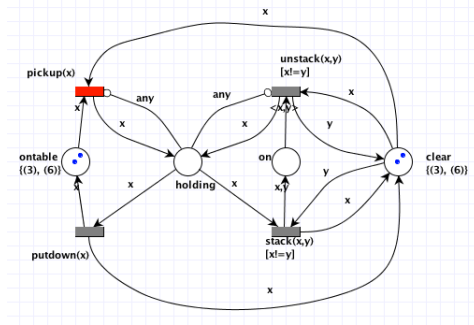


Figure 50. Firable transitions

When a simulation is active, the function net shows which transitions can be fired and how many tokens reside in each place. As shown in Figure 50, transition pickup is in the color red because it is firable. The simulation allows only one firing at a time. Once pickup is selected to fire, the matched tokens in ontable and clear are removed and a new token is added to holding. As a result, putdown and stack become enabled, as shown in Figure 51.

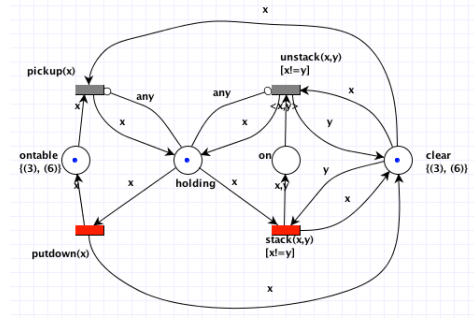


Figure 51. Token updates during simulation

Log files are named as MIDFILENAMELogY.txt, where MIDFILENAME is the name of the MID file and Y is the index of the log file. The traces (firing sequences) in a log file can be used to generate tests. In this case, make sure all the traces were created from the same version of the function net; otherwise test generation may fail.

When a net hierarchy is simulated, MISTA always presents the net where the last transition was fired. You can also check the state of each subnet.

2.7 Verification


The following functions can help find semantic errors in a function net.

- **Verify goal state reachability:** It checks whether or not the given goal states are reachable from the given initial states in the test model. The related search







options are breadth-first vs depth-first search, maximum search depth, and check for home states (refer to Section 2.8). You can use some goal states that you know they are reachable or unreachable and compare your expectations with the verification results. This can help you find potential errors in your function net. If there exist multiple paths from an initial state to a goal state, only one path is presented - it may be different from what you expect. Note that a theoretically reachable state may not be reported reachable because the search is bounded due to the maximum search depth.

- **Verify transition reachability:** It checks to see whether there are transitions that are unreachable from the given initial states. Usually, when you build a function net, you would expect that all transitions in a test model are reachable. If there are unreachable transitions, the function net probably contains errors.
- **Check for deadlock/termination states:** It checks to see if there are deadlock/termination states and if so, what sequences of transition firings will reach these states. It depends on the following search options: breadth-first vs depth-first search and maximum search depth. A deadlock/termination state refers to a state under which no transition is firable. It does not necessarily mean the occurrence of deadlock. It can be a normal termination state.
- **Verify assertions:** It checks the specified assertions against the function net. Assertions typically represent the properties that are required of the function net. If an assertion is not satisfied, the verification will report a counterexample. The related search options are breadth-first vs depth-first search and maximum search depth.

2.8 Generating Test Tree

To generate a test tree, you can either click on  icon in the tool bar or select the “Generate Test Tree” menu item from the “Test” menu. MISTA generates test cases to meet the coverage criterion chosen from the dropdown menu of test coverage. The test cases are organized in a test tree. The test coverage criteria are as follows:

- 📖 **Reachability tree coverage:** 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.
- 📖 **Reachability + invalid paths (sneak paths):** MISTA generates an extended reachability graph - for each node, MISTA also create child nodes of invalid firings (they are leaf nodes). A test from the corresponding initial marking to such a leaf node is called a dirty test.
- 📖 **Transition coverage:** MISTA generates tests to cover each transition.
- 📖 **State coverage:** MISTA generates tests to cover each state that is reachable from any given initial state. The test suite is usually smaller than that of reachability tree coverage because duplicate states are avoided.

- 
Depth coverage: MISTA generates all tests whose lengths are no greater than the given depth.
- 
Random generation: MISTA generates tests in a random fashion. The parameters used as the termination condition are the maximum depth of tests and the maximum number of tests. When this menu item is selected, you will be asked to set up the maximum number of tests to be generated. The actual number of tests is not necessarily equal to the maximum number because random tests can be duplicated.
- 
Goal coverage: MISTA generates a test for each given goal that is reachable from the given initial states. Before generating tests for this coverage, you should verify goal reachability to see if they are reachable. Typically, the firing sequences that reach the given goals will be transformed into tests (refer to Section 2.7).
- 
Assertion counterexamples: MISTA generates tests from the counterexamples of assertions that result from assertion verification (Refer to Section 2.7). Before generating tests, you may verify assertions to see if the specified assertions have counterexamples.
- 
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. Test generation makes use of the result of “Check for Deadlock/Termination States”.
- 
Given sequences: MISTA generates tests from the firing sequences in a given file (e.g., the log file of simulation or online testing under the same folder as the MID file). The file is specified by a “SEQUENCES” annotation. Make sure all the firing sequences in the file were created from the same version of the function net; otherwise test generation may fail.

Reachability tree coverage is the default coverage option. Test tree generation also depends on several search options, as shown in Figure 52. The dialog in Figure 52 brought up by selecting the menu item “Options...” from the “Test” menu.

- **Breadth first vs depth first:** This option applies to reachability tree coverage, reachability tree coverage with dirty tests, transition coverage, state coverage, depth coverage, goal coverage, and deadlock/termination state coverage. It does not apply to random test generation or given sequences.
- **Maximum depth:** This option applies to all coverage criteria except for given sequences. For depth coverage, you should not use a large maximum depth, otherwise the test suite may be too large to generate.
- **Home states:** A home state is an initial state (marking) that is reached by a non-empty sequence of transition firings from itself. The “Home states”

options apply to reachability analysis and test generation for state coverage. When verifying the reachability of a goal marking that is the same as an initial marking, “Check home states” is to check if this marking is a home state, i.e., try to find a firing sequence that reaches this marking from itself. “Do not check home states” does not check if the marking is a home state - it is simply reachable from itself with an empty firing sequence. When generating tests for state coverage, “Check home states” create tests to cover the initial markings if possible. For example, if a function net has four possible states s_0 , s_1 , s_2 , and s_3 , where s_0 is the initial state. “Check home states” will generate tests to cover four states if s_0 is a home state. “Do not check home states” will create tests to cover s_1 , s_2 , and s_3 no matter whether or not s_0 is a home state.

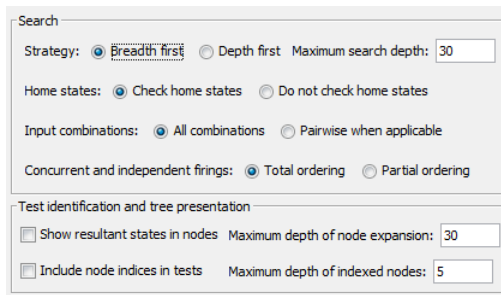


Figure 52. Options for test tree generation

- **Input combinations:** you can apply either all combinations according to the general rule of transition firings or pairwise input combinations for transition firings when applicable. Pairwise is only applicable to those transitions that have more than two input places, no inhibitor places, and no guard condition.
- **Concurrent and independent firings:** this option deals with the ordering of concurrent and independent firings. Total ordering refers to generation of all interleaving sequences, whereas partial ordering yields one sequence. For example, there are six interleaving sequences of three independent firings. When partial ordering is used, only one of them is created. This sequence can depend on the ordering in which the transitions are defined.

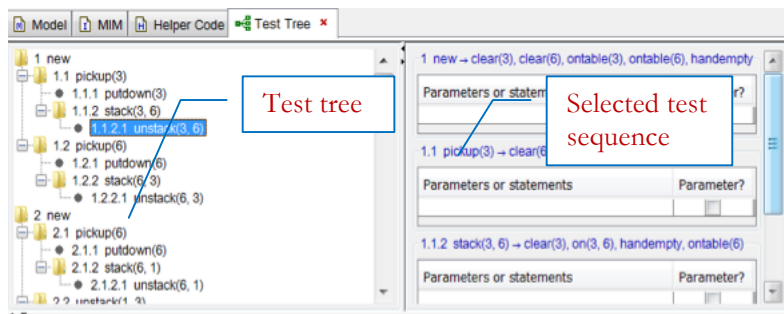


Figure 53: Test tree interface

The test tree interface is shown in Figure 53. It consists of four parts: test tree, selected test sequence (from the root to the selected node), test information, and test code for the selected sequence. The node “1 new” is the root of the test tree for the first initial state, whereas the node “2 new” is the root of the test tree for the second initial state. To select a test node, you can click on the target node. The information about this node will be shown. If it is a leaf node, the selected test sequence and corresponding code will also be presented.

The following options in Figure 52 are about test identification and tree presentation.

- **Show resultant states in nodes:** This checkbox indicates whether to show the resultant states (markings) of all nodes in a test tree. If there is already a test tree, this option will take effect immediately. Figure 54 shows an example, where the resultant state of each node is presented.

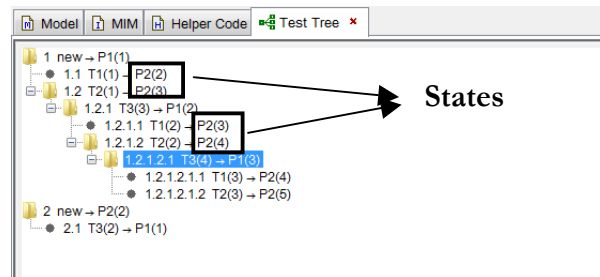


Figure 54. States in test nodes.

- **Maximum depth of node expansion:** It means how many nodes of test sequences are presented in the test tree.
- **Maximum depth of indexed nodes:** indexed node refers to a node with an index. Nodes beyond the specified depth will have no index. When generating test code, you may opt to include node indices in the test names. This may help debug function nets. When you randomly generate tests with a large maximum depth, you should use a small maximum depth of indexed nodes to improve the performance.

2.8.1 Test Tree Operations

MISTA allows a test tree to be edited before test code generation.

- **Edit test tree:** Edit test tree refers to deleting, cloning, and moving a selected node. Deleting a node also implies deleting all descendants. Cloning a node creates a copy of the current node and its descendants. Moving a node is to change the order of sibling nodes.
- **Save test tree:** This operation saves the current tree, including test parameters and code edited manually. By default, the name of the test tree file is the same as the MID file name, but the file extension is .test. Opening a test tree file is similar to opening a MID file, except that the saved tree is also presented.

- **Print test tree:** This option is used to print the current test tree.
- **Close test tree:** This option is used to close the test tree tab if the test tree is generated (not imported from a test tree file).
- **Show model-level tests:** This option is used to display the model-level tests in the current test tree. Figure 55 shows an example. If the checkbox “Show resultant states in nodes” is checked (refer to Figure 52), the resultant state of each test is also printed.

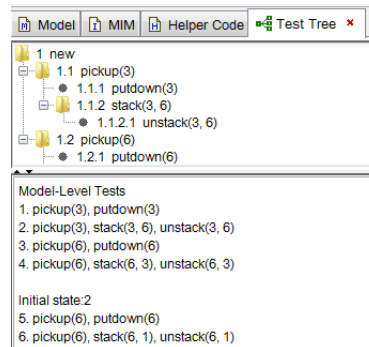


Figure 55. Model-level test sequences in a test tree.

2.8.2 Editing Test Parameters

Generally, test parameters are generated automatically from test models. MISTA also allows test parameters and even code to be edited manually. Once a test tree has been generated, you may specify test parameters for any test nodes.

Let us consider the function net in Figure 56. It represents a basic communication protocol between two parties. It is not concerned about the contents of the message. Figure 57 shows a test tree generated from the net in Figure 56. The code for test sequence 1.1.1.1.1.1 is shown in Figure 58. It is not executable because the actual parameters are not given.

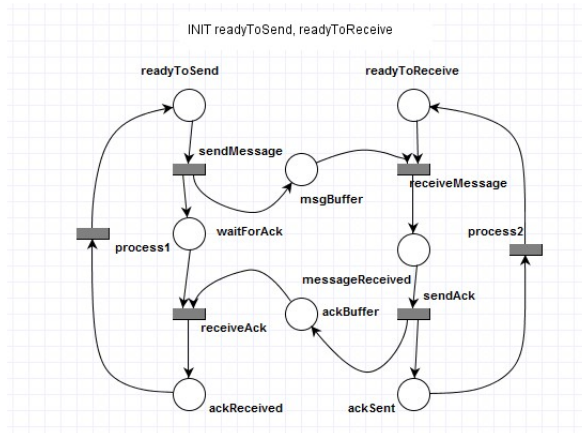


Figure 56. Function net of a communication protocol

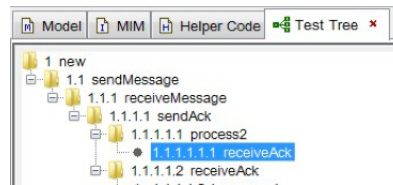


Figure 57. Test tree generated from the function net in Figure 56.

```

void test1_1_1_1_1_10 {
    printf("Test case 1_1_1_1_1_10\n");
    setUp();
    sendMessage
    assert(waitForAck(), "1_1");
    assert(readyToReceive(), "1_1");
    assert(msgBuffer(), "1_1");
    receiveMessage
    assert(waitForAck(), "1_1_1");
    assert(messageReceived(), "1_1_1");
    sendAck
    assert(waitForAck(), "1_1_1_1");
    assert(ackSent(), "1_1_1_1");
    assert(ackBuffer(), "1_1_1_1");
    process2
    assert(waitForAck(), "1_1_1_1_1");
    assert(readyToReceive(), "1_1_1_1_1");
    assert(ackBuffer(), "1_1_1_1_1");
    receiveAck
    assert(ackReceived(), "1_1_1_1_1_1");
    assert(readyToReceive(), "1_1_1_1_1_1");
}

```

Figure 58. Test code before editing test parameters.

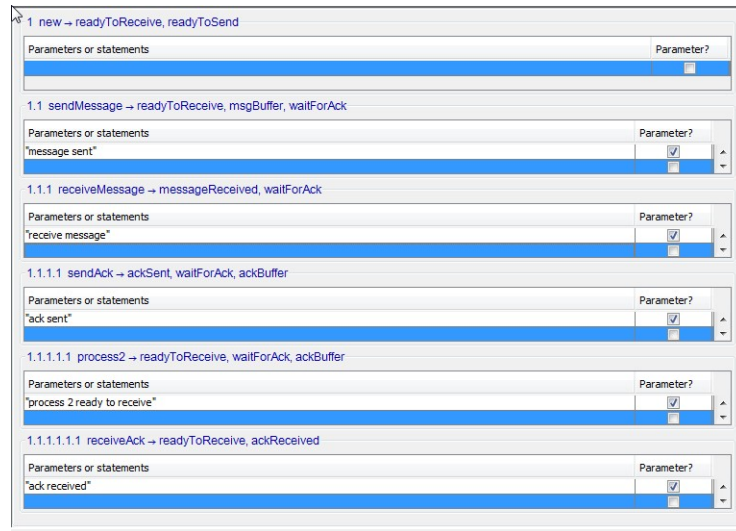


Figure 59. Manual editing of actual parameters.

To make the test executable, you may provide the actual parameters by clicking on the target node as shown in Figure 59. If the “parameter” checkbox is selected, the input will be used as a parameter, otherwise it is inserted as code. If there are multiple parameters or statements, they appear in the test code in the specified order.

After the parameters are provided in Figure 59, the test code looks like the following:


```
void test_1_1_1_1_10 {
    printf("Test case 1_1_1_1_1\n");
    setUp();
    sendMessage("message sent")
    assert(waitForAck(), "1_1_1");
    assert(readyToReceive(), "1_1_1");
    assert(msgBuffer(), "1_1_1");
    receiveMessage("receive message")
    assert(waitForAck(), "1_1_1_1");
    assert(messageReceived(), "1_1_1_1");
    sendAck("ack sent")
    assert(waitForAck(), "1_1_1_1");
    assert(ackSent(), "1_1_1_1");
    assert(ackBuffer(), "1_1_1_1");
    process2("process 2 ready to receive")
    assert(waitForAck(), "1_1_1_1_1");
    assert(readyToReceive(), "1_1_1_1_1");
    assert(ackBuffer(), "1_1_1_1_1");
    receiveAck("ack received")
    assert(ackReceived(), "1_1_1_1_1_1");
    assert(readyToReceive(), "1_1_1_1_1_1");
}
```


Figure 60. Test code after the actual parameters are provided.



When you provide actual parameters, make sure they satisfy the guard conditions in each test, otherwise your tests may be invalid. You may save and load parameters and statements by saving and loading the test tree.

2.9 Generating Test Code

To generate test code from the working MID or from the current test tree, you can either click on  in the tool bar or select the “Generate Test Code” menu item from the “Test” menu. Before generating test code, you can also set up test generation options by selecting the “Options” menu item from the “Test” menu. The “Options” dialog is shown in Figure 61. If there is no test tree yet, you should also check the options for test tree generation as discussed in Section 2.7.

 If test code is generated successfully, MISTA saves it into one or more files under the same folder as the MID file. If it is generated from the existing test tree, the file name is reported in the Test Information area of the test tree interface (refer to Figure 53). If it is generated directly from the working MID, the file name is reported in the Console window of the main MISTA interface (refer to Figure 2). The test code files can be viewed with a separate editor.

- **Include node indices in tests:** The test name of each test will include the index of the leaf node (the last node in the test sequence) if exists.

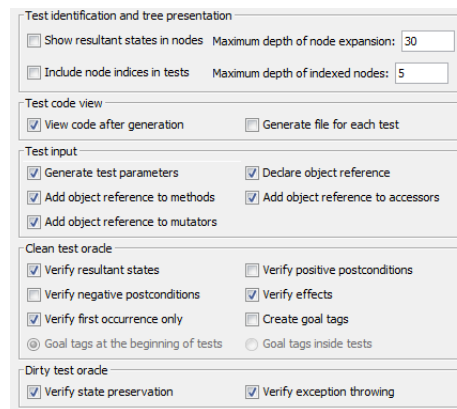


Figure 61. Options for test code generation.

- **Include node indices in tests:** The test name of each test will include the index of the leaf node (the last node in the test sequence) if exists.
- **View test code after generation:** If this option is checked, the test code will be presented in a new tab. If there are multiple test code files, only one test suite file will be shown. If the test code file is too large to be presented, a run-of-memory error may occur. In this case, you close the progress dialog window and view the test code files with a separate editor.

- **Generate file for each test:** Each test will be stored in a separate file. This feature can be used when a single file for all tests is too large.
- **Generate test parameters:** The actual parameters of transition firings are used in tests. If this option is unchecked, the actual parameters of transition firings are discarded and you may edit the test parameters manually.
- **Declare object reference:** this is used only for an object-oriented language (Java, C++, C#, PHP, and VB) when the SUT is a class or the head class of a cluster. A variable of this class will be declared. For example, the SUT is a Java class Block, the object reference is: Block block; In this case, block can be referenced in MIM and thus test code.
- **Add object reference to methods/accessors/mutators:** these options are used only when “Declare object reference” is checked. The object reference will be added automatically to the beginning of each method/accessor/mutator. For example, if the declared method for a transition in the MIM specification is pickup(?x) and the object reference is block, the actual method in the test code will be block.pickup(?x);
- **Verify result states:** each token in the resultant state of each transition firing will be used as a test oracle unless its place is listed in the hidden events/conditions of the MIM.
- **Verify positive postcondition:** new tokens from each transition firing will be used as test oracles unless their places are listed in the hidden events/conditions of the MIM.
- **Verify negative postcondition:** removed tokens due to each transition firing will be used as test oracles unless their places are listed in the hidden events/conditions of the MIM.
- **Verify first occurrence only:** Because many test sequences share some common paths (test inputs and oracles), this option avoid repeating the oracles of the same test inputs in different tests so as to improve performance. It does not affect the test code of the selected test in test tree interface (refer to Figure 53), where the oracles of all test inputs are generated. Therefore, this code may look different from the final test code.
- **Verify effects:** effects associated with transitions will be used as test oracles.
- **Create goal tags:** Goal tags are used to indicate which goal markings are covered by which test cases. A goal tag could be a print statement in a general-purpose language (e.g., Java) or a tag in a tag-supported test framework (e.g., RobotFramework). When you check this box, you may edit the tag template for the selected language and test framework as shown in Figure 62, where the

placeholder [NAME] or [NAME,] will be substituted for the list of the specified goals that are reached in the test. If [NAME] is used, multiple goal names are separated by space. If [NAME,] is used, multiple names are separated by “,”.

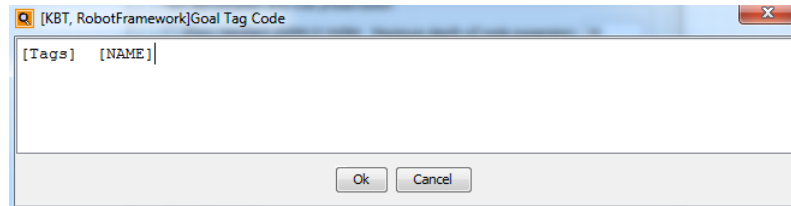


Figure 62. Tag code dialog

- **Goal tags at the beginning of tests:** This is enabled only when “Create goal tags” is checked. It attaches the goal tags to the beginning of each test. The tags list all goal markings reached in each test.
- **Goal tags inside tests:** This is enabled only when “Create goal tags” is checked. It inserts the goal tags after respective test inputs and oracles where goal markings are reached.
- **Verify state preservation:** This is used for dirty tests. In a dirty test, the last transition firing or test input is invalid. State preservation means that this invalid test input should not change the system state. Thus, the tokens in the marking before the invalid transition firing can be used as test oracles.
- **Verifying exception throwing:** This is used for dirty tests. The system is expected to throw an exception when the invalid transition firing is attempted. The dirty tests will check for an exception. This only applies to languages with an exception handling mechanism.

When the compiler reports syntax errors of your test code, you need to figure out what and where the syntax errors are. Then you modify the MID (e.g., MIM or helper code), and regenerate test code, and compile it again. This process can be repeated until there is no syntax error. The following discussions on test code structures help you understand which part of the MIM or helper code is corresponding to which portion of the test code.

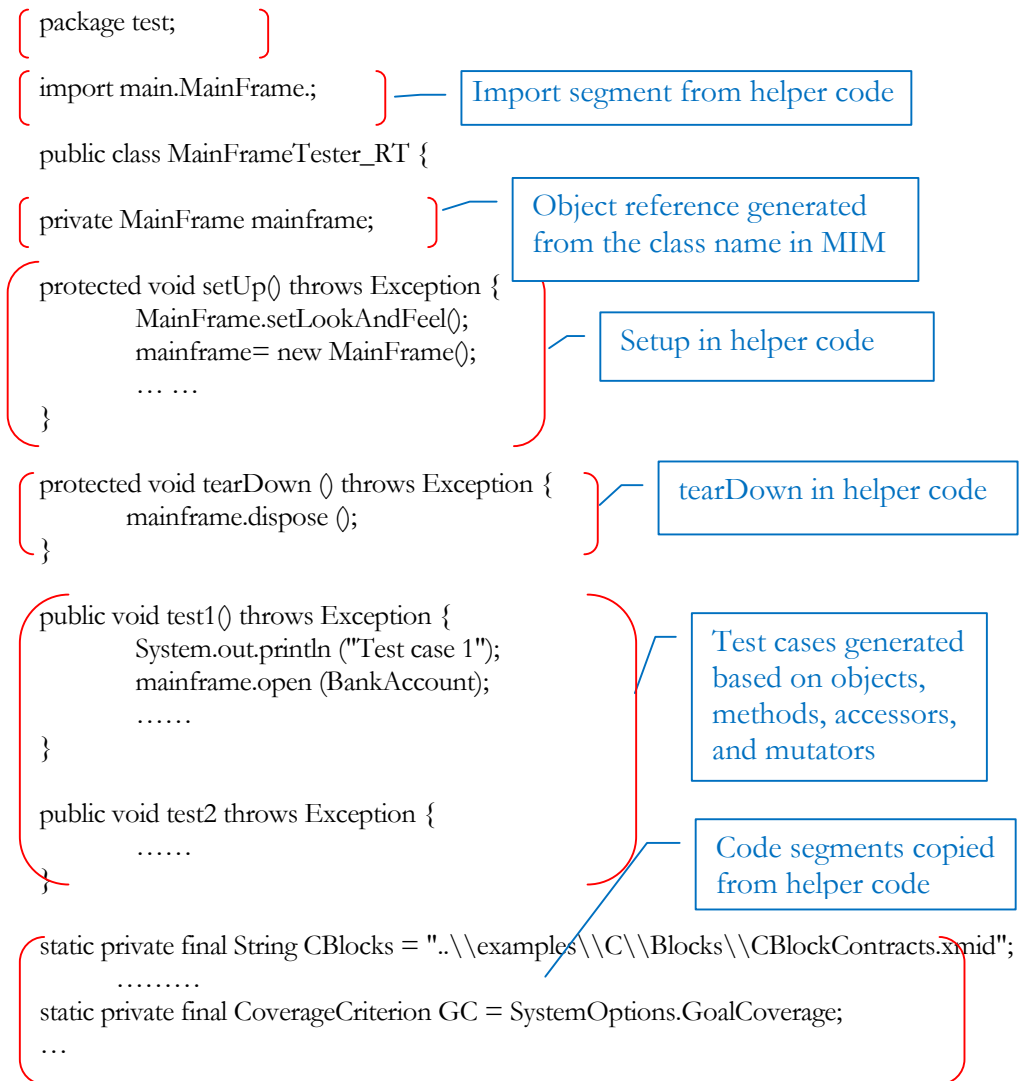
2.9.1 Object-Oriented Test Code

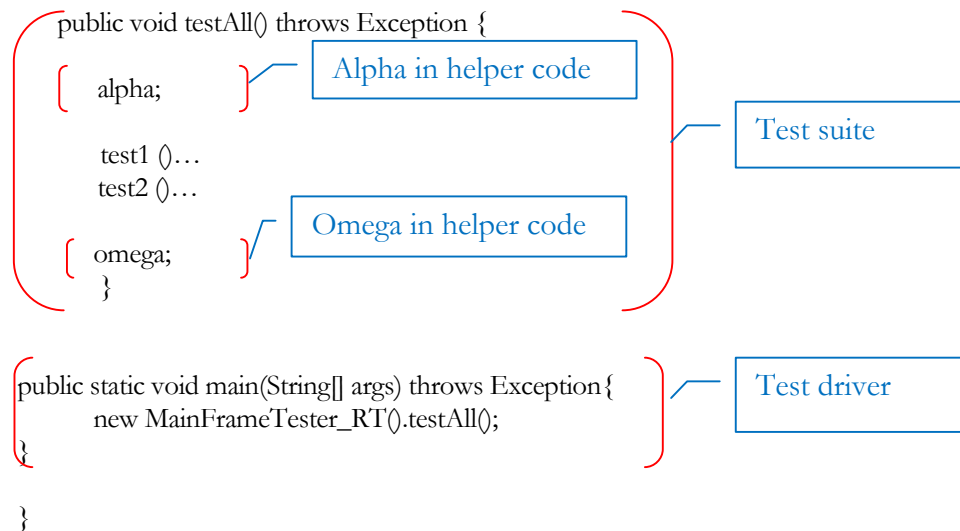
The object-oriented (Java, C++, C#, PHP, and VB) test code is one or more classes, depending on whether or not you want to generate a separate file for each test. The structure of the single test class in Java consists of the following portions:

- header (e.g., package and import statements) from the helper code,
- class declaration according to the given class name in MIM (or MID file name if class name is not specified),

- declaration of object reference according to the given class name if the option “Declare object reference” is checked,
- setup method from the helper code,
- teardown method from the helper code,
- a method for each test according to the specifications of objects, methods, accessors, and mutators in MIM,
- code segments copied from the helper code,
- test suite method (the testAll method) that invokes the alpha code in the helper code, each test method, and the omega code in the helper code.
- test driver (i.e., the main method).

When a test framework (e.g., JUnit) is used, the test suite method and the test driver are not generated. This indicates that the alpha and omega code in MIM is not used. The following shows an example. The test code in C++, C# or VB has a similar structure.





2.9.2 HTML/Selenium Code

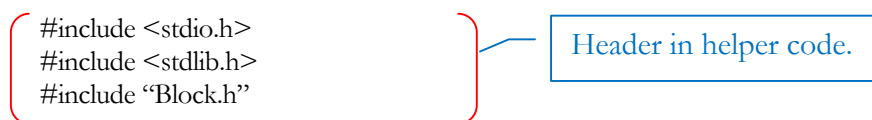
The HTML/Selenium test code consists of one or more HTML files, depending on whether or not you want to generate a separate file for each test. If a separate file is generated for each test, an HTML file for the test suite will also be generated. It includes a hyperlink to each test case file. You may open the test suite file to execute all the tests. The setup and teardown code is inserted into the beginning and end of each test, respectively. The alpha/omega code is inserted into the beginning/end of the test suite, respectively.

2.9.3 C Test Code

The structure of C test code in a single file consists of the following portions:

- header (`#include` etc.) from the helper code,
- setup method from the helper code,
- teardown method from the helper code,
- assert function,
- a function for each test according to the specifications of objects, methods, accessors, and mutators in MIM,
- code segments copied from the helper code,
- test suite method (the `testAll` method) that invokes the alpha code in the helper code, each test method, and the omega code in the helper code.
- test driver (i.e., the `main` method).

The following shows an example of C test code.



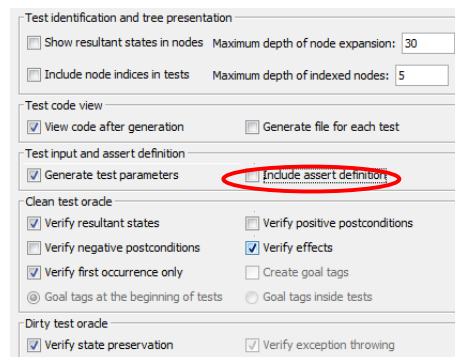
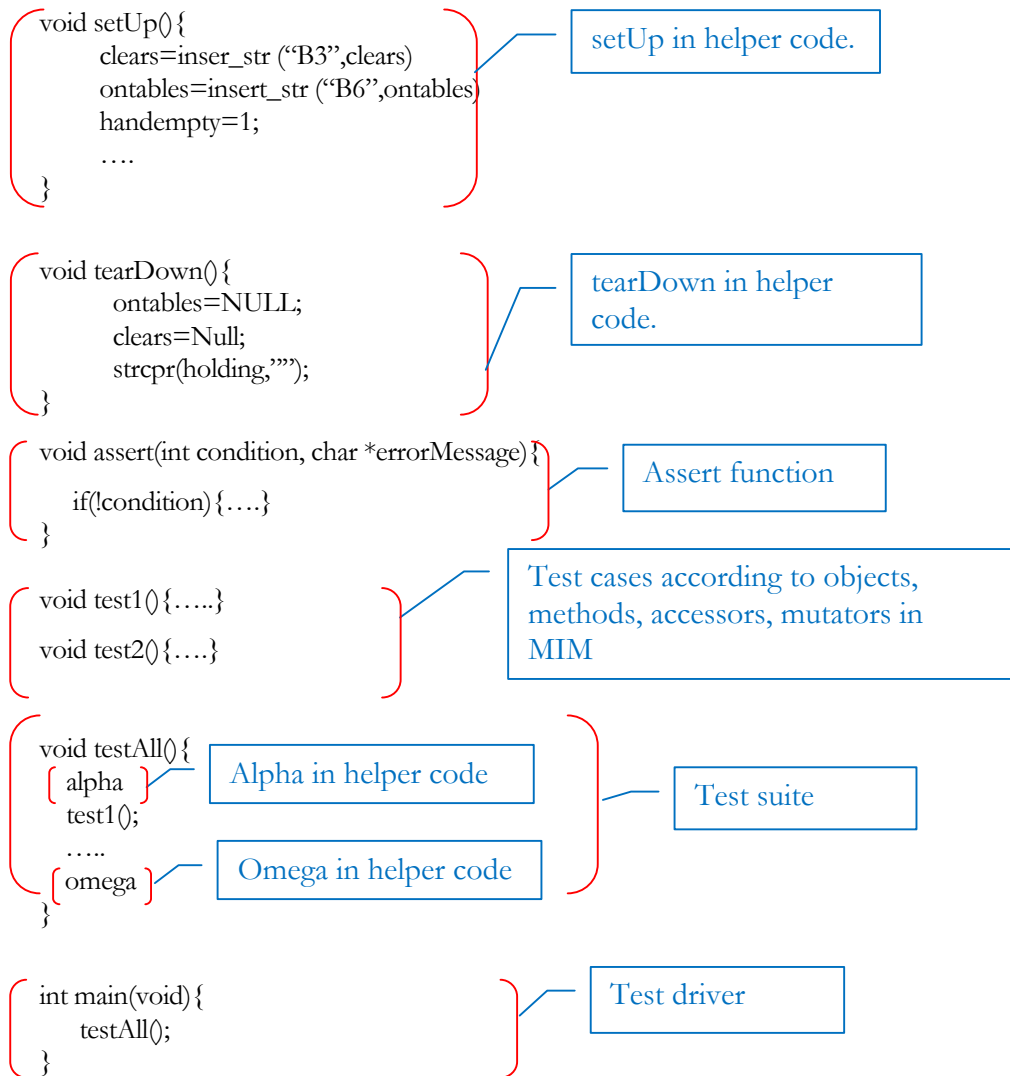


Figure 63. Code generation options for C

The sample C code has included the definition of the assert function. You may provide your own definition of the assert function in the `#include` part of the helper code. It

must have two arguments: int and char *. In this case, you need to uncheck the option “Include assert definition” as shown in Figure 63.

2.10 Online Testing

There are two ways to perform online testing: on-the-fly testing (simultaneous generation and execution of tests) and online execution of the tests in a test tree. Before performing online testing, make sure the server is running. The target language should be set to “Selenium” or “RPC”. “Selenium” means that the online test execution is done through Selenium web driver. It requires the availability of a web browser (e.g., Firefox). It might take a couple of minutes to launch a browser. “RPC” means that the online test execution will be done through remote procedure calls to the server under test. The RPC protocol supported by the server can be either JSON-RPC (<http://json-rpc.org/>) or XML-RPC (<http://xmlrpc.scripting.com/>). JSON-RPC and XML-RPC both use HTTP as the transport and XML as the encoding.

In order to perform online testing, you need to provide correct MIM and helper code. If you use Selenium web driver, MIM and helper code are in the same format as for HTML/Selenium IDE. You can use Selenium commands in the implementation code of “Methods”, “Accessors”, “Mutators” in the MIM, setUp and tearDown in the helper code. The Selenium commands are executed via doCommand. If you use a RPC protocol, the implementation code of each of the “Methods”, “Accessors”, “Mutators” in the MIM, setUp and tearDown in the helper code must be a list of method/function calls in the following forms:

methodName(arg1, ..., argn) or
methodName :arg1 ... : argn

arg1 and argn are the arguments of the remote method calls supported by the server. Arguments are optional. The implementation code of an accessor must return a boolean value (true or false). It should avoid side effects.

2.10.1 On-the-fly Testing and Test Analysis

To perform on-the-fly testing, select the menu item “On-the-fly Testing” from the “Test” menu. A control panel will be presented as in Figure 64. It looks similar to the simulation control panel. In addition, the test inputs and test oracles of transition firings are executed on the server. If you are unable to execute tests, check if there is a network problem and if the remote methods calls are indeed supported by the server. “Execute Selected Firing” and “Execute Random Firing” are for stepwise testing. The former uses the event and parameters that you have selected; the latter applies a randomly selected event and randomly selected parameters. “Start Random Testing” and “Stop Random Testing” are for continuous testing. At each step, it randomly selects an event and its parameters. If “Automatic restart” is not checked, continuous testing will terminate if one of the following conditions occurs:

- a) The test has failed,

- b) The test cannot be performed (e.g., due to a network problem),
- c) No transition is firable,
- d) The test has exceeded the maximum search depth (refer to Figure 52. Options for test tree generation).

If “Automatic restart” is checked, the continuous random testing will be repeated until you stop it or exit the simulation. If there are multiple initial markings, the repeated random testing will randomly choose an initial marking.

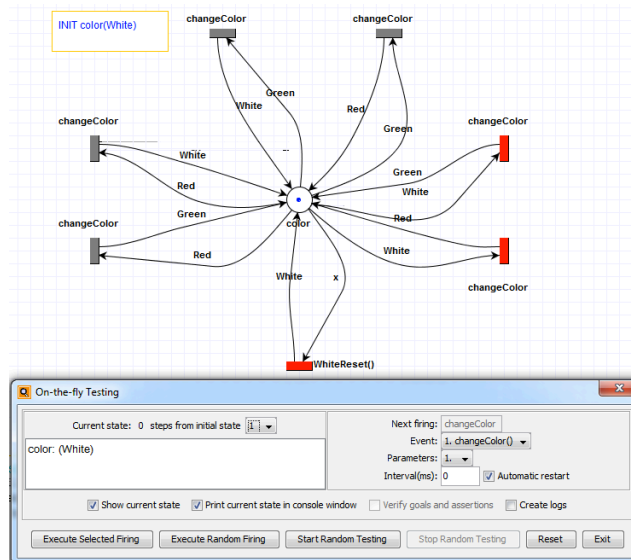


Figure 64. The control panel of on-the-fly testing

- When an on-the-fly testing session is logged, MISTA produces two log files: XXXTestLogY.txt includes all the test sequences that have tried and XXXFailLogY.txt includes the test sequences that have failed. XXX refers to the name of the MID file and Y is the index of the testing session, starting from 1.
- You may import the tests in a log file into a test tree as follows: create a “SEQUENCES” annotation in the function net (SEQUENCES is the keyword, followed by the log file name), select “Given Sequences” from the test coverage dropdown menu, and then select “Generate Test Tree” from the “Test” menu. In the test tree, the tests are combined and duplication is removed. For example, multiple identical tests will appear only once. If a test is a subsequence of another test, only the longer one will appear.
- On-the-fly testing can deal with non-deterministic systems where one event under a certain state may lead to different and even random results. Such situations can be modeled by multiple transitions with the same event. When one of the transitions is fired during on-the-fly testing,

MISTA checks the actual result to determine which is the corresponding transition firing in the test model. In comparison, online test execution does not verify non-determinism in the implementation. If the implementation under tests is non-deterministic, the pre-existing tests may fail. The reason is that their test oracles are deterministic.

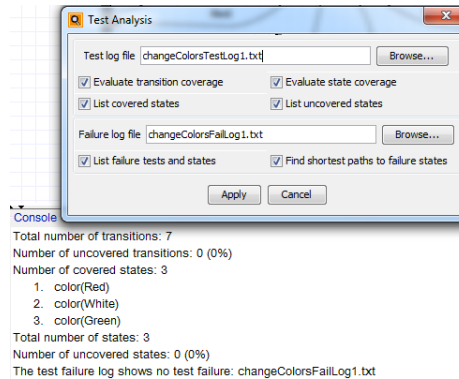


Figure 65. Analysis of on-the-fly tests

After an on-the-fly testing session is done, you may continue to analyze the executed tests by selecting the menu item “Analyze On-the-fly Tests...” from the “Test menu”. Through a dialog as in Figure 65, you can choose to perform the following analysis:

- **Select a test log for analysis:** This is the log file to which “Evaluate transition coverage”, “Evaluate state coverage”, “List covered states”, and “List uncovered states” are applied. MISTA presents the most recent test log file and you can choose to open a different file by clicking on the “Browse...” button.
- **Evaluate transition coverage:** this will report how many transitions are covered and what transitions are not covered by the on-the-fly tests.
- **Evaluate state coverage:** this will report how many states are covered and how many states are not covered by the on-the-fly tests.
- **List covered states:** This is to list the states that are covered by the tests.
- **List uncovered states:** This is to list the states not covered by the tests.
- **Select a failure log for analysis:** The log file contains a set of test cases that have failed. The log file name must contain “FailLog”, which how MISTA generates failure log files. The failure log can be applied to “List failure tests” and “Find shortest paths to failure states”. MISTA presents the most recent failure log file and you can choose to open a different one by clicking on the “Browse...” button.

- **List failure tests:** this will show all the failure tests and respective failure states.
- **Find shortest paths to failure states:** this will report the shortest path to each failure state. This path can be different from the on-the-fly tests.

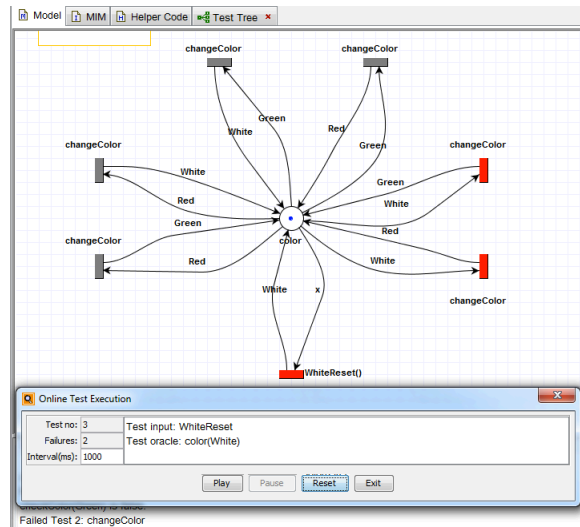



Figure 66. Control panel of online test execution

- The shortest paths to failure states, if found, are saved into the file whose name is obtained by replacing “FailLog” in the failure log file name by “PathLog”. These paths can be transformed into tests through test generation with given sequences.
- To analyze the tests from multiple testing sessions, you may merge the log files into one.

2.10.2 Online Execution of Existing Tests

To execute tests in an existing test tree generated for a selected coverage criterion, select the menu item “Online Test Execution” from the “Test” menu. The control panel will be presented as in Figure 66. Click on the “Play” button to execute the tests in the sequential order. You may stop the execution by clicking on the “Pause” button. “Reset” will point test execution back to the first test. You are allowed to configure the time interval between two test actions. The control panel shows how many of the tests have been executed and failed, as well as the current test input and test oracle. Failed tests are logged in the console area of the main interface. If you only want to execute some of tests in a test tree, you can remove the unwanted tests from the test tree before starting online execution.

 The options for clean test oracles (except for goal tags) apply to online testing. The “Generate test parameters” option is applied no matter whether it is checked. Refer to Figure 61 or Figure 63.

2.11 Spreadsheet Editor for Function Nets

A function net can be viewed as a set of transitions, where each transition is defined by event signature (name and parameters), precondition, postcondition, guard, and effect. The graphic representation of a function net can be transformed into textual format as follows: for each transition, the precondition is of the form $p_1(x_1, \dots, x_m)$ and ... and not $p_n(y_1, \dots, y_k)$, where p_1 is an input place with (x_1, \dots, x_m) as the arc label and p_n is an inhibitor place with (y_1, \dots, y_k) as the arc label. If the arc label from an input place p is $(x) \& (y)$, then $p(x)$ and $p(y)$ are both part of the precondition. The postcondition is of the form $q_1(x_1, \dots, x_m)$ and ... and $q_n(y_1, \dots, y_k)$, where each q_i is an output place. If place p is both an input and output place with arc label (x_1, \dots, x_m) , then $p(x_1, \dots, x_m)$ appears in both precondition and postcondition.

Figure 67 shows the spreadsheet editor for a function net (textual representation of the graphic model in Figure 4), where each row is corresponding to one transition and “when” refers to guard condition. In the spreadsheet editor, initial states and goal states are edited in separate text areas. You should not use the keywords “INIT” or “GOAL”, though. The Model submenu of the Edit menu provides operations for addition and deletion of initial and goal states.

Model MIM Helper Code					
No	Transition	Precondition	Postcondition	When	Effect
1	pickup(x)	ontable(x), clear(x), not holding(any)	holding(x)		
2	putdown(x)	holding(x)	ontable(x), clear(x)		
3	stack(x, y)	holding(x), clear(y)	on(x,y), clear(x)	$x \neq y$	
4	unstack	not holding(any), clear(x), on(x,y)	holding(x), clear(y)	$x = y$	
Initial state [1]					
clear(3), clear(6), ontable(3), ontable(6)					
Initial state [2]					
clear(1), clear(6), on(1, 3), ontable(3), ontable(6)					
Initial state [3]					
clear(2), clear(5), on(2, 3), on(5, 6), ontable(3), ontable(6)					

Figure 67. Spreadsheet editor for function nets.

 The spreadsheet editor of function nets does not support net hierarchy.