MISTA Reference Manual: Function Nets

January 2012

Abstract: Function nets[1] are a modeling notation for building test models in MISTA. This manual introduces the basic concepts of function nets, the building blocks of function nets, methodologies for building function nets for test generation purposes, and techniques for reducing the state space of function nets. To gain a better understanding about function nets, we strongly recommend you play with the simulator in MISTA.

1. Definitions

The basic elements of function nets include places, transitions, arcs, arc labels, and initial markings (states). We use the function net in Figure 1 for illustration purposes.

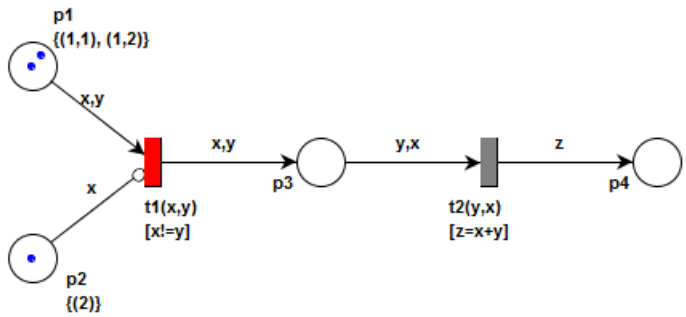

Figure 1. A sample function net

1.1 Places, Tokens, and Markings

A place (represented by a circle ○) specifies a condition or state. Its name starts with a letter and consists of only letters, digits, dots, and underscores. The function net in Figure 1 has four places: p_1, p_2, p_3, and p_4. Different places must have different names. Places can hold primitive and structured data, called tokens. Each token is a tuple $(X_1, ..., X_n)$, where $X_1,...,X_n$ are constants. A constant is a number, string, or symbol (which starts with an uppercase letter and only consists of letters and digits). "*()*" (zero-tuple) is a non-argument token. Multiple tokens in the same place are separated by ",". They should be different from each other but have the same number of arguments. In Figure 1, place p_1 has two tokens *(1,1)* and *(1,2)*, whereas p_2 has one token *(2)*. 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. $\{p_1(1,1), p_1(1,2), p_2(2)\}$ is such an initial marking in Figure 1. An initial marking can also be specified in an annotation that starts with the keyword "INIT". For example, *INIT* $p_1(1,1), p_1(1,2), p_2(2)$ represents the same initial marking in Figure 1.

[1] Function nets are a simplified version of high-level Petri nets (predicate/transition nets and colored Petri nets), which have been widely used for system and software modeling.

Multiple initial markings can be specified for the same net. From the test generation perspective, each initial marking will result in a test suite. This allows test data to be partitioned into smaller data sets so as to reduce the complexity of test models. Section 4.3 will elaborate on this issue. When a function net is simulated, you may choose one initial marking to work with.

1.2 Transitions

A transition (represented by a rectangle ▯) specifies an event with parameters and conditions (e.g., method call and GUI operation). The event signature includes an event name and an optional list of variables as its formal parameters. The function net in Figure 1 has two transitions t_1 and t_2 - t_1's formal parameters are x and y and t_2's formal parameters are y and x. Different transitions can be specified for the same event. A variable is an identifier that starts with a lowercase letter or "?". Each variable in a transition's parameters must be defined in the label of some arc connected to the transition or in the transition's guard condition. In Figure 1, t_1's parameters x and y appear in the arc from p_1 to t_1. It is recommended that transition parameters should be listed explicitly, especially when a transition has multiple input variables. If the specified parameter list is *()*, it means that there is no formal parameter no matter how many variables are contained in the input arcs. If the list of parameters is not given, all variables collected from the arcs connected to the transition and the transition's guard condition will become the formal parameters. The ordering of these variables depends on the order in which arcs are created. To avoid confusion, this should be used only when a transition has zero or one input variable.

A transition can be associated with a guard condition, which is a conjunction of arithmetic and relational predicates. Multiple predicates are separated by ",", which means logical "*and*". A predicate is for the form *[not]* $p(x_1, x_2 \ldots, x_n)$, where "*not*" is optional and x_i is a variable or constant. If x_i is a variable, it must appear in the label of some arc associated with the transition. Suppose x and y are variables or integers and p is a place name. The predicates for specifying guard conditions are as follows:

- x=y or equals(x,y)
- x!=y or x<>y or not equals(x,y)
- x>y or gt(x, y)
- x>=y or gte(x, y)
- x<y or lt(x,y)
- x<=y or lte(x, y)
- z=x+y or add(x, y, z)
- z=x-y or subtract(x, y, z)
- z = x*y or multiple(x,y,z)
- z=x/y or divide(x,y,z)
- z=x%y or modulus(x,y,z)
- isOdd(x): x is odd
- isEven(x): x is even
- belongsTo(x, y1, y2,…, yn): x belongs to the set {y1, y2,…, yn}
- assert(p): p has at least one token,
- tokenCount(p, x): the number of tokens in p is x (variable or integer constant)
- bound(x): variable x is bound

In Figure 1, t_1's guard condition is $x!=y$, and t_2's guard condition is $z=x+y$. Currently, MISTA does not support general expressions in guard conditions.

1.3 Arcs

An arc represents an input/output relationship between a place and a transition. There are three types of arcs:

- 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. In Figure 1, p_1 is t_1's input place and p_3 is t_1's output place.

- 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 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.

- Inhibitor arc: an inhibitor arc from a place to a transition represents a negative precondition of the transition. In Figure 1, p_2 is a negative precondition of t_1.

An arc can be labeled by one or more tuples (lists) of arguments. Each tuple contains one or more arguments. An argument can be a variable or constant. For an unlabeled arc, the default arc label is <>, which contains no argument. In Figure 1, The input arc from p_1 to t_1 and the output arc from t_1 to p_3 are labeled by x,y, respectively. The inhibitor arc from p_2 to t_1 is labeled by x. Each variable in the label of an output arc of a transition should be defined in some input arc of the transition or in the transition's guard condition. When an arc label has multiple tuples, separated by "&", all the tuples must have the same number of arguments. An example is $<a, b, c>$ & $<c, d, e>$. This is identical to two arcs of the same type that are labeled by $<a, b, c>$ and $<c, d, e>$, respectively. For simplicity, the following discussions assume that an arc label is one tuple of arguments.

1.4 Scope of Variable

Variables of the same name may appear in different locations, such as arc labels, transition arguments, and guard conditions. They are identical only when they are associated with the same transition. To determine the scope of a variable, you can first identify the transition associated with the variable and then find all components associated with the transition (labels of all arcs connected to or from the transition, transition's formal parameters, and transition's guard condition). In Figure 1, variables associated with transition t_1 are x and y, including arc label x, y of the arc from p_1 to t_1, arc label x of the arc from p_2 to t_1, arc label x,y of the arc from t_1 to p_3, t_1' arguments (x, y), and t_1's guard condition $x!=y$. All occurrences of x (or y) in these locations are the same. However, they are independent of and different from that x in the arc from p_3 to t_2, t_2's arguments, or t_2's guard condition.

1.5 Transition Firing

Given a function net, the places, transitions, arcs, arc labels, and initial markings describe its static structure. The semantics (i.e., dynamic behaviors) are determined by possible transition firings. A transition firing is an occurrence of the event with specific input values (like a method call and GUI operation). To determine whether a transition is firable with specific input values,

we try to match the label of each input arc (a tuple of variables and constants) with the tokens (tuples of constants) in the corresponding input place. Consider transition t_1 in Figure 1. Its input place p_1 has tokens *(1,1)* and *(1,2)* and the input arc label from p_1 to t_1 is <*x, y*>. Thus, there are two matches for *x* and *y*: {*x=1, y=1*} and {*x=1, y=2*}. Each match is a set of variable bindings. Such variables (or bound values) are typically formal parameters (or actual parameters) of a transition (or transition firing). Given a set of variable bindings, we can derive a token from an arc label by replacing each variable in the arc label with the corresponding value. Suppose the variable bindings are {*x=1, y=1*}, the label <*x*> of the inhibitor arc from p_2 to t_1 is corresponding to token <*1*>. It does not match the token <*2*> in place p_2. Given variable bindings, we can also evaluate predicates in a guard condition. For example, *x!=y* is false if the variable bindings are {*x=1, y=1*}.

A transition is firable with specific input values under marking M_0 if variable bindings can be found according to the tokens in the input and inhibitor places, the labels of the corresponding arcs, and the transition's guard condition. The variable bindings must satisfy the following conditions:

(1) Each input place has a token that matches the label of the input arc;
(2) Each inhibitor place has no token that matches the inhibitor arc label;
(3) the guard condition evaluates to true.

Note that a non-argument token *()* matches a default arc label <>. It is possible that a transition is associated with no variable. The variable bindings for firing such a transition may have no variable when only non-argument tokens and default arc labels are involved. Under the same marking, there can be a number of firable transitions and one transition can be fired with a number of different sets of variable bindings. MISTA automatically finds all possible sets of variable bindings for all transitions under a given marking. From the test generation perspective, which one is chosen to fire depends on the test generation strategy. From the simulation perspective, you may select one to fire or have it randomly chosen by the simulator.

In Figure 1, t_1 is firable by variable bindings *{x=1, y=2}* because p_1 has token *(1, 2)*, p_2 does not have token *(1)*, and guard condition *x!=y* holds. However, t_1 is not enabled by variable bindings *{x=1, y=1}* although p_1 has token *(1, 1)* and p_2 does not have token *(1)*. This is because guard condition *x!=y* is false.

Firing an enabled transition with given variable bindings removes the matched token from each input place and adds a new token to each output place. The new token is corresponding to the label of the output arc, where each variable in the label is substituted for its bound value. So transition firing may lead to a new marking. Note that function nets do not support duplicate tokens in the same place - the tokens in the same places are a set, not a bag. If the outplace has already had a token that is identical to the new token, addition of the new one to the output place takes no effect. In this case, the net is called unsafe. This is not necessarily wrong, depending on what is being modeled.

In Figure 1, firing t_1 with variable bindings *{x=1, y=2}* removes token *(1, 2)* from p_1 and add new token *(1,2)* to p_3. The new token is *(1,2)* because the output arc label is <*x, y*>. The resultant marking is *{p_1(1,1), p_2(2), p_3(1,2)}*. Under this new marking, t_2 becomes enabled by *{y=1, x=2,*

$z=3\}$. Firing this transition leads to marking $\{p_1(1,1), p_2(2), p_4(3)\}$. We use $t_1(1,2)$ to denote transition firing $t_1\{ x=1, y=2\}$ because t_1's parameters are x and y. Similarly, $t_2(1,2)$ means $t_2\{ y=1, x=2, z=3\}$ because t_2's parameters are y and x (z is not listed in the specification).

Generally, each transition firing involves four components: current marking M_{i-1}, transition t_i, a set of variable bindings θ_i, and resultant marking M_i. A sequence of transition firings looks like $M_0, t_1\theta_1, M_1, t_2\theta_2, M_2, \ldots, t_n\theta_n, M_n$, or simply $t_1\theta_1, t_2\theta_2, \ldots, t_n\theta_n$, where firing transition t_i with variable bindings $\theta_i(1 \le i \le n)$ under M_{i-1} leads to M_i $(1 \le i \le n)$. A marking M is said to be reachable from M_0 if there is such a firing sequence that leads to M from M_0. In Figure 1, $\{p_1(1,1), p_2(2), p_4(3)\}$ is reachable from the initial marking by the firing sequence $t_1\{ x=1, y=2\}, t_2\{ y=1, x=2, z=3\}$. Given a goal marking, MISTA can automatically check whether or not it is reachable from an initial marking. This is called reachability analysis.

In MISTA, test cases are essentially firing sequences from the given initial markings. A positive test is corresponding to a valid firing sequence, whereas a dirty test is corresponding to an invalid firing sequence (typically the last one in the sequence is an invalid firing). The transition firings in the sequence are test inputs, whereas the tokens in the resultant marking of each firing are test oracles.

The behaviors of a function net can be interpreted by its reachability graph/tree, where each initial marking is corresponding to a root node. The graph can be built by expanding each root node as follows: (1) find *all possible* firings $t_i\theta_j$ under the marking of the current node under expansion; (2) for each firing $t_i\theta_j$, create a new child node according to the new marking. The edge from the current node to the new child is labeled with $t_i\theta_j$. (3) if the new marking has not expanded before, then expand the new node. MISTA only deals with function nets with a finite reachability graph.

1.6 Hierarchical Nets

A test model may consist of a hierarchy of function nets. A transition in a parent net can be linked to another function net (called subnet) by observing the following rules. 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. 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. 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. 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.

2. Building Blocks

Building a function net is somehow similar to designing a program with the basic building blocks (data structures, sequences, conditions, loops, etc.) in the given language. In the following, we introduce the basic building blocks of function nets.

2.1 Test Data and Data Processing

In function nets, tokens represent data items. They do not have data types. Data items fall into two categories: primitive data (such as integers, strings, and symbols) and structured data. Structured data are represented by tuples. For example, *(EMAIL, PSWD)* represents a pair. In principle, nested tuples are possible but they are not supported in the current version of MISTA. We do not anticipate that test models need to use complex data structures. To associate test data to a test action, you may represent the test action as a transition, the data name as an input place of the transition, and data items as tokens in the place.

A transition can represent data processing. The inputs of the processing are data items from the input places and the outputs are the data items added to the output places. Processing of the input data is expressed by arithmetic and relational predicates in the transition's guard condition, such as the following:

+, -, *, /, =, !=, >, <, >=, <=

Predicates in a guard condition are evaluated from left to right. All variables in a relational predicate (e.g., *x* and *y* in *x*>*y*) and variables on the right side of an arithmetic operation (*x* and *y* in *z*=*x*+*y*) must be bound when the predicate is evaluated. Variable bindings are first obtained by matching tokens in input places and variables in related arc labels.

2.2 Sequential Structure

A sequential structure specifies a number of transitions performed in a sequential order. Figure 2 shows an example. $t_1(1), t_2(1)$ is a firing sequence. t_2 cannot be fired before t_1. Here we assume no other transition can put tokens to p_2.

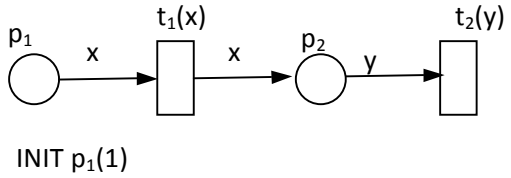

Figure 2. A sequential structure

2.3 Condition and Choice

Conditions in a conditional structure can be represented by guard conditions of different transitions. Figure 3 is a function net that simulates the following conditional statement:

```
if (x==1)
  …
else if (x==2)
  …
else if (x>2)
  …
```

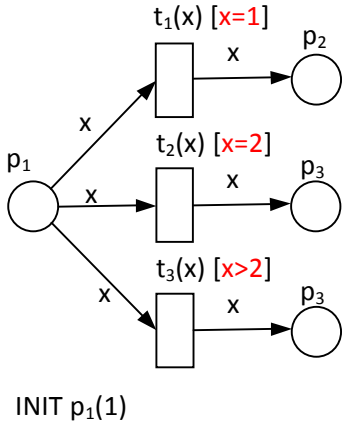

Figure 3. A conditional structure

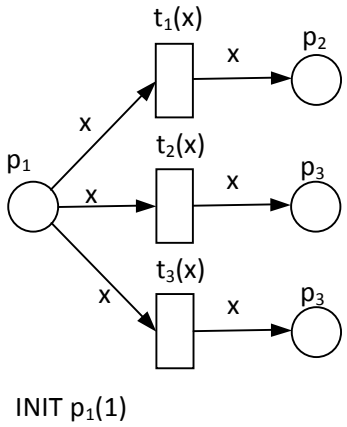

Figure 4. A choice structure

If $M=\{p_1(1)\}$, then t_1 can be fired with $x=1$; If $M=\{p_1(2)\}$, the t_2 can be fired with $x=2$; If $M=\{p_1(5)\}$, the t_3 can be fired with $x=5$; If $M=\{p_1(1), p_1(2)\}$, then t_1 can be fired with $x=1$; t_2 can be fired with $x=2$; they do not conflict. The function net in Figure 3 becomes a free choice structure if there are no guard conditions (refer to Figure 4). If $M=\{p_1(1)\}$, then t_1, t_2, OR t_3 can be fired with $x=1$; but firing one of them will disable others because there is only one token in p_1.

2.4 Loops

The net in Figure 5 simulates the following loop statements.

```
        for (x=1; x<10; x=x+1)
                t1(x);
        t2(x);
or
        x=1;
        do {
                t1(x);
                x=x+1;
        } until x>=10
        t2(x);
```

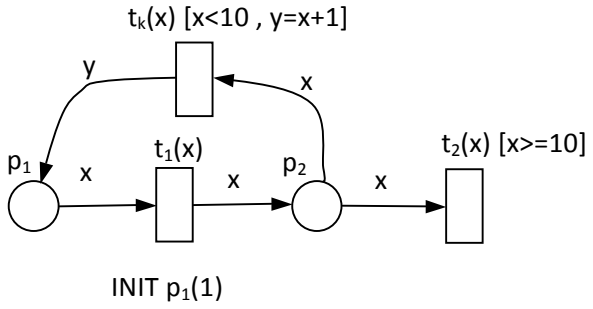

Figure 5. A loop structure

y=x+1 in the guard condition of t_k, *x<10, y=x+1*, is used to update the loop control value (we cannot use *x=x+1* as in a programming language). If the arc from t_k to p_1 is labeled with *x*, t_1 can be fired for an undetermined number of times because the loop control value is not updated at all. The guard condition *x>=10* of transition t_2 represents the termination/exit condition of the loop.

2.5 Concurrency Structure

Figure 6 shows a typical concurrency structure, where t_1/t_4 are fork/join transitions (start and end points of concurrency). After t_1 fires, t_2 and t_3 are both enabled. Firing t_2 does not affect t_3 and vice versa. They are concurrent/parallel transitions. Concurrency is not simply determined by the structure. It also depends on the marking. A seemingly non-concurrent structure may be a concurrent one as shown in Figure 7 and Figure 8.

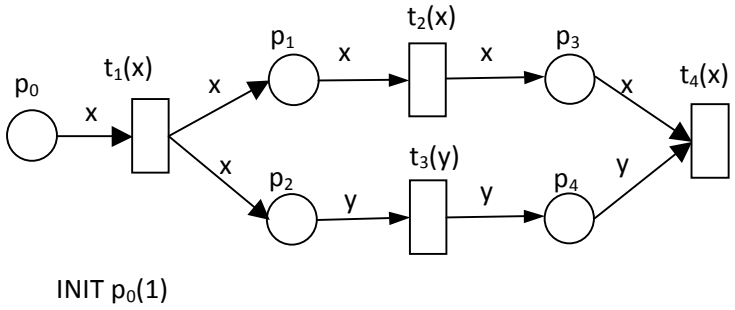

Figure 6. Fork/join structure

The structure of the net in Figure 7 appears to be a choice or synchronization, but it could be part of a concurrent structure because of the given marking. For instance, it is similar to the client-server architecture, where a number of clients access the server at the same time. Transitions t_1, t_2, and t_3 can be fired by variable bindings *{x=1}*, *{x=2}*, *{x=3}*, respectively. The different inputs are all provided in p_1. $t_1(1)$, $t_2(2)$, and $t_3(3)$ can be fired concurrently. MISTA takes the interleaving semantics of concurrency, which means only one transition can be fired at a time. From the test generation perspective, concurrent and independent transition firings can lead to a large number of interleaving sequences. For example, there are 6 different firing sequences of $t_1(1)$, $t_2(2)$, and $t_3(3)$ in Figure 7.

The structure of the net in Figure 8 appears to be a sequential one (refer to Figure 2); but it is not because of the given initial marking. In fact, it is a typical pipeline structure for concurrency. Under the initial marking, t_1 and t_2 are enabled by variable bindings *{x=1}* and *{y=2}*, respectively. Transition firings $t_1(1)$ and $t_2(2)$ do not interfere with each other. Such firings can also cause a number of interleaving sequences.

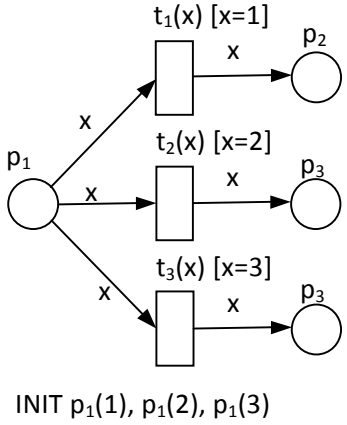

Figure 7. Implicit concurrency structure I

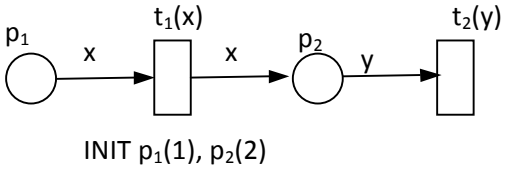

Figure 8. Implicit concurrency structure II

2.6 Logic Operators in Preconditions

Here we refer to logic operators as "and", "or", and "not" (negation). For a given transition, its input places and associated arc labels represent its precondition.

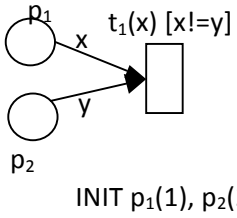

Figure 9. Logic operator "and" for precondition

- **and**

According to the definition of transition firing, "and" is the logic operator between multiple input places and the guard condition. In Figure 9, transition t_1 has two input places p_1 and p_2. Its precondition is $p_1(x)$ *and* $p_2(y)$ *and* $x!=y$.

- **not**

A negative input condition is represented by an inhibitor arc. In Figure 10, t_1 has three input places, p_1, p_2, and p_3. p_2 and p_3 are connected by inhibitor arcs. t_1's precondition is $p_1(x)$ *and not* $p_2(x)$ *and not* p_3.

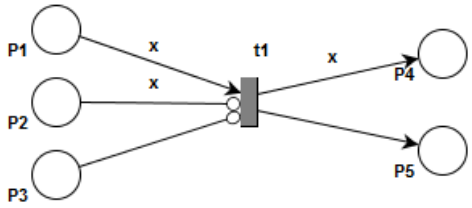

Figure 10. Logic operator "not" for precondition

- **or**

To represent the "or" relation between multiple input conditions of an event, multiple transitions with the same event name can be used. In Figure 11, there are two transitions with respect to event t_1. Either $p_1(x)$ or $p_2(y)$, will enable one of them. Event t_1's precondition can be interpreted as $p_1(x)$ *or* $p_2(y)$. If $p_1(x)$ and $p_2(y)$ are both true, firing both transitions would lead to two tokens in p_3. To represent "*xor*", inhibitor arcs can be used from p_3 to the transitions as shown in Figure 12.

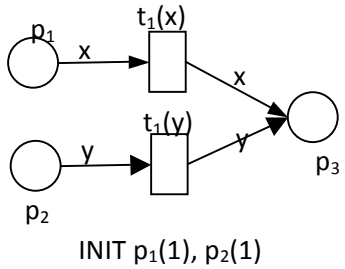

Figure 11. Logic operator "or" for precondition

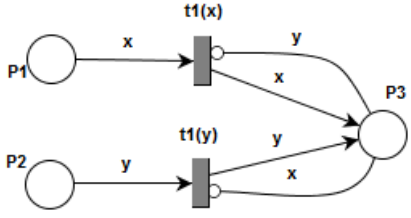

Figure 12. Logic operator "xor" for precondition

2.7 Logic Operators in Postconditions

For a given transition, its output places and associated arc labels represent its postcondition.

- **and**

  According to the semantics of transition firings, "and" is the logic operator between multiple output places. In Figure 13, transition t_1 has two output places p_3 and p_4. Its postcondition includes $p_3(x)$ *and* $p_4(y)$.

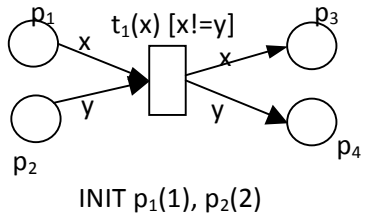

Figure 13. Logic operator "and" for precondition

- **not**

  Transition firing implies negation of input conditions because of token removal. In Figure 13, the complete postcondition of t_1 is $p_3(x)$ *and* $p_4(y)$ *and not* $p_1(x)$ *and not* $p_2(y)$. If negation is not needed for an input condition, the input place should also be used as an output place (i.e., using a bidirectional arc).

  The default negation in transition firing only applies to the input conditions of the transition – these conditions are true before firing. Negation can also be represented by a RESET arc from a transition to a place. After the transition is fired, all tokens in this place will be removed (no matter whether there is a token and no matter how many tokens are in this place before the firing).

- **or**

  The "or" relation between multiple output conditions of an event can be represented by multiple transitions of the same event. Each transition produces a particular output condition. In Figure 14, the postcondition of t_1 is $p_2(x)$ *or* $p_3(x)$.

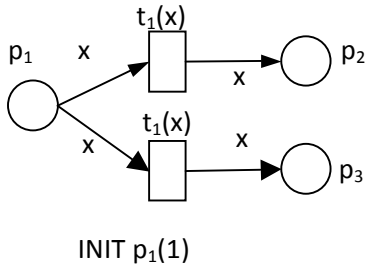

Figure 14. Logic operator "or" for postcondition

3. Building Function Nets as Test Models

A test model describes what the test targets (e.g., system functions) are and how they will be tested together. To build a test model as a function net, you can first create a "mental" model of your test design and then formalize it by using the building blocks of function nets. As function nets are a general formalism, there are various ways for developing mental models of test design. In the following, we introduce several ways for building test models.

3.1 Building Function Nets from Workflows

One paradigm for building a test model is to create various use case scenarios of your test targets and organize them in the form of a workflow. A workflow depicts a sequence of connected operations. It can be easily formalized by a function net. Let us consider system testing of MISTA itself. The main test targets are the following components: *open* a MID file, *parse* the MID specification, *verify* the MID specification, *generate* transition tree, *generate* test code, and *close* the tree. To generate a test tree, a coverage criterion should be selected. Suppose the above functions will be tested sequentially with different MID files, we can build a testing workflow as shown in Figure 15. Place *fileName* stores the MID files to be used and place *coverage* represents various coverage criteria. They are associated with *open* and *generateTree*, respectively. Each test scenario includes a sequence of the functions with a specific MID file and coverage criterion. The scenario will be repeated after the last step *closeTree* is completed.

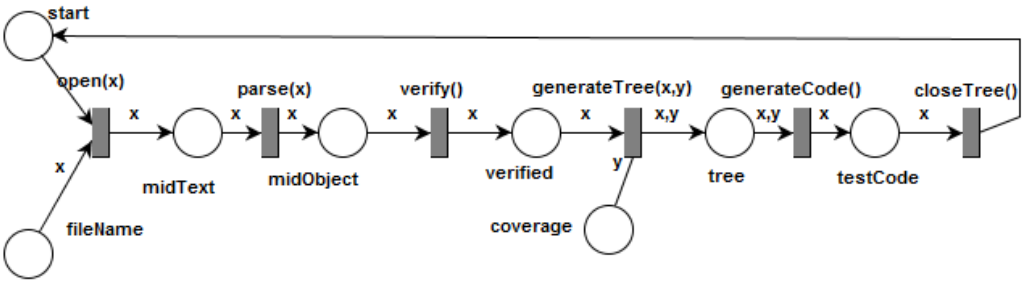

Figure 15. A testing workflow

3.2 Building Function Nets from Contracts

Another paradigm for building a test model is to define the interactions of test targets in terms of their contracts (preconditions and postconditions). In this approach, inputs and outputs are represented as predicates. A precondition is a conjunction of predicates that describes the conditions on the inputs of a test target, whereas a postcondition is a conjunction of predicates that describes the conditions on the outputs of a test target. A collection of contracts forms a test model. MISTA can transform contracts into a function net automatically. Table 1 shows a sample set of contracts.

Table 1. Sample contracts

No	Module	Precondition	Postcondition	When	Effect
1	pickup(x)	ontable(x), clear(x), not holding(any)	holding(x), not ontable(x), not clear(x)		
2	putdown(x)	holding(x)	ontable(x), clear(x), not holding(x)		
3	stack(x, y)	holding(x), clear(y), not equals(x, y)	on(x,y), clear(x), not holding(x), not clear(y)		
4	unstack	clear(x), on(x,y), not holding(any), not equals(x,y)	holding(x), clear(y), not clear(x), not on(x,y)		

3.3 Building Function Nets from Other Behavior Models

If you can specify your test design with as a flowchart or UML activity diagram, you may transform it into a function net according to the relations between their building blocks, such as sequences, conditions, and loops.

4. Reducing State Space

The state space of a function net refers to the set of markings that are reachable from any initial marking. Transition firings depend on combinations of transition's inputs. Such combinations together with the interleaving of independent transition firings may lead to the explosion of state place. For example, an endless loop in a function net implies an infinite state space.

Consider the function net in Figure 16. *T1* has two input places *X* and *Y*. Under the given initial state, *T1* is enabled by 16 different combinations of <*X*, *Y*>. After *T1* is fired by one combination of <*X*, *Y*>, there are three tokens in *X* and three tokens in *Y*. So *T1* is still enabled by *9* combinations of *X* and *Y*. As such *T1* can be fired continuously for several times. In the reachability graph, there are hundreds of different firing sequences of *T1*. Let us add one more transition as shown in Figure 17. The firing of *T1* under the initial marking also enables *T2* with different input combinations of *XY* and *Z*. The reachability graph has become much more complex than that of the net in Figure 16 – it has more than 10,000 firing sequences of *T1* and *T2*. Be careful when you create a function net with such input combinations as in Figure 16 or Figure 17. In the following, we discuss several techniques for dealing with this issue.

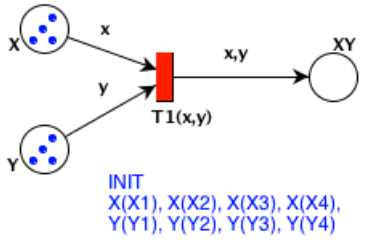

Figure 16. Input combinations of a transition

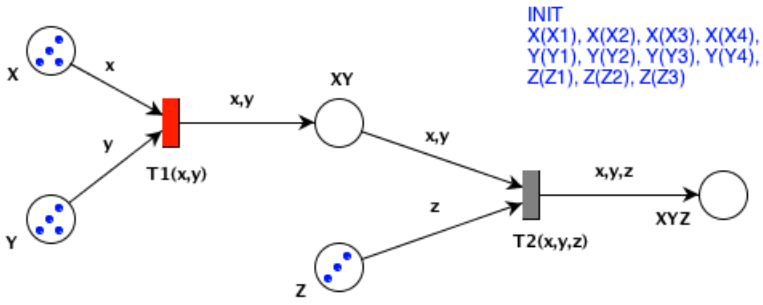

Figure 17. Sequence of transitions

4.1 Partial Ordering and Pairwise Techniques

To reduce the number of firing sequences, you may opt to use the partial ordering and pairwise techniques implemented in MISTA. For independent firings of the same transition or concurrent firings of different transitions, the partial ordering technique generates only one sequence to cover these firings. Applying this technique to the net in Figure 17 result in less than 300 firing sequences. The pairwise technique covers all input pairs, rather than all combinations, of a transition. It is only applicable to transitions that have more than two input places with distinct variables in the arc labels, no inhibitor places, and no guard condition. Consider the function net in Figure 18. Under the initial marking, there are 48 combinations of <*X, Y, Z*> for firing *T1*. Using the pairwise technique, only16 combinations of <*X, Y, Z*> are created to cover all possible pairs of <*X, Y*>, <*X, Z*>, and <*Y, Z*>.

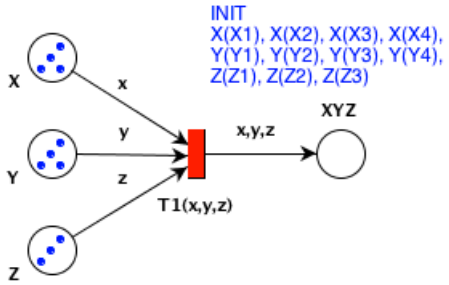

Figure 18. Pairwise example

4.2 Net Structure

Generally a function net for testing purposes does not intend to capture all behaviors of the SUT. It should focus on the components under test. When a test model involving many components is too complex, it should be divided into multiple function nets. Guard conditions and inhibitor arcs are often useful for reducing state space. Particular attention should be paid to seemingly sequential structures that imply independently firable transitions.

A control place can be used to enforce a sequential structure and prevent interleaving firings of the same transition with different input combinations. It is an input place of the first transition in the sequential structure and has only one token in the initial state. Applying the control place to the net in Figure 17 results in the net in Figure 19. *T1* is still enabled by 16 input combinations of <*X, Y*>. They are no longer independent – firing one of them will disable others. *T2* is disabled before *T1* is fired. In the reachability graph, each firing sequence consists one firing of *T1* and one firing of *T2*. The total number of firing sequences is 48.

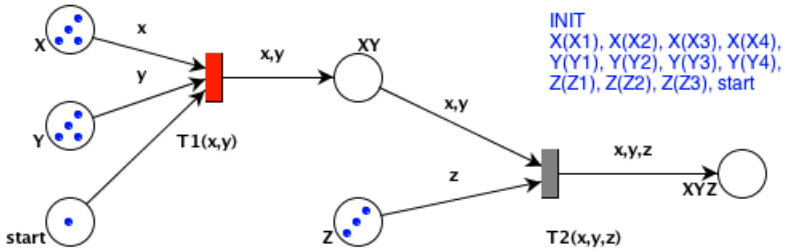

Figure 19. Enforcement of sequence

If *T2* does not depend on the concrete data from *T1*, we can further reduce the firing sequences as shown in Figure 20, where variables *x* and *y* are removed from the labels of the arc from *T1* to

XY, the arc from *XY* to *T2*, and the arc from *T2* to *XYZ*, as well as from the parameters of *T2*. Also the directed arcs from *X* to *T1*, from *Y* to *T1*, and from *Z* to *T2* are changed to bi-directional ones. As a result, the tokens in *X*, *Y*, and *Z* remain unchanged when *T1* or *T2* is fired. This reduces the number of possible markings. Although *T2* is enabled by the firing of *T1*, it does not depend on the concrete values of *X* and *Y*. In the reachability graph, the total number of firing sequence is 18. The three different firings of *Z* only occur in three firing sequences.

As a rule of thumb, a bidirectional arc should be used when the purpose of a place is for binding data to an event. In Figure 20, *start* and *XY* are called control-oriented places because they determine the sequences of firings, whereas *X*, *Y*, and *Z* are called data-oriented places. In Figure 19, XY is used for both control and data.

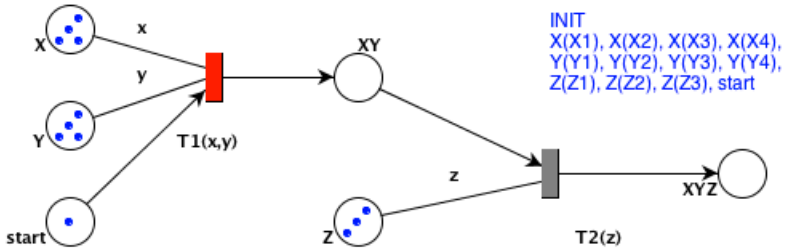

Figure 20. Data places vs control places

4.3 Initial States

The initial states of a function net typically specify test data and system settings. One of the basic software testing principles is the use of representative test data. If you believe that two values *X1* and *X2* for input *X* will test the same target, you should use only one of them unless you need to test their interaction. When building a function net in Figure 16 or Figure 17, make sure each initial state includes only representative test data.

Another important technique for the reduction of state space is to partition test data by dividing them into multiple initial states. In Figure 18, there are 48 different combinations of <*X, Y, Z*>. If the initial state is divided into the following two initial states: *INIT X(X1), X(X2), Y(Y1),Y(Y2), Z(Z1)* and *INIT X(X3), X(X4), Y(Y3),Y(Y4), Z(Z2), Z(Z3)*, then there would be only 12 (i.e., 4+8) combinations.