NModel Reference

Colin Campbell
Modeled Computation LLC

Margus Veanes
Microsoft Research

Jonathan Jacky
University of Washington

April 2007
(Revised October 2008)

Contents

Chapter 1

Modeling library reference

A model program created from C# source consists of actions and variables defined by the types declared in a single namespace. The namespace name is the model program name. For example:

```
namespace MyModelProgram
{
    static class Contract
    {
        static bool MyStateVariable1 = true;

        [Action]
        static void Reset() { /* ... */ }
    }

    class Client : LabeledInstance<Client>
    {
        bool isEntered = false;
        // ...
     }
 }
```

The model program named `MyModelProgram` has two state variables, `MyStateVariable1` and `isEntered`, as well as one action named `Reset`. The `isEntered` state variable can be seen as a field map of `Client` object IDs to Boolean values, since it has a value for each instance of the class.

The tools allow you to provide a factory method as a command-line argument. The factory method should be a static public method with return type `ModelProgram`. For example,

```
namespace MyModelProgram
```

```
{
    public static class Factory
    {
        public static ModelProgram Create()
        {
            return new LibraryModelProgram(typeof(Factory).Assembly,
                                           "MyModelProgram");
        }
    }
}
```

The `Create` method in this example instantiates a `ModelProgram` object from the definitions for the `MyModelProgram` namespace found in the current assembly.

In Release 1.0.20312.00 and later, the factory method is no longer required for model programs. If the factory method is omitted, use the `/mp:` option to identify the model program name (namespace) on the command line: `/mp:MyModelProgram` in this example. See chapter 2.

The rest of this appendix describes the attributes and classes provided by the modeling library. The attributes label types, members, and parameters in model program source files. The classes define structural and collection data types for model programs. As described in section 3.1, a model program must use these classes (instead of the usual .NET collection types) in order to work with the tools.

The final section of this appendix describes how to construct and access the terms used to represent actions, particularly when writing test harnesses.

1.1 Attributes

1.1.1 Features

Classes defined within a model program may be labeled as belonging to named feature sets using the `[Feature]` attribute. A feature is a cluster of related state variables and actions. Only classes may have the `[Feature]` attribute. More than one `[Feature]` attribute may be used on a given class.

Feature attributes may appear in either of two forms: `[Feature]` and `[Feature("`*name*`")]`. If no name is given, then the class name is used as the feature name.

The modeling tools provide a way to selectively load all classes tagged with a given feature name. A typical use of features is to strengthen enabling conditions of the model for scenario control. Another use might be to model slices of functionality in a way that does not require explicit composition of separate model programs.

The following example shows a case of an enabling condition based on the state of other model elements. Note that feature interaction may occur if state defined within

one feature is referenced outside of that feature.

```
namespace MyModelProgram
{
    [Feature]
    static class CreditsRestriction
    {
        static int nMsg = 0;

        static bool ResetEnabled()
        {
            return (nMsg < 10 && Contract.MyStateVariable1);
        }

        [Action]
        static void Reset()
        {
            nMsg += 1;
        }
    }
}
```

The factory method to instantiate this feature is the following:

```
namespace MyModelProgram
{
    public static class Factory
    {
        public static ModelProgram Create()
        {
            return new LibraryModelProgram(typeof(Factory).Assembly,
                                           "MyModelProgram",
                                           new Set<string>("CreditsRestriction"));
        }
    }
}
```

The features to be loaded are included as a `Set`-valued argument to the constructor. All classes in the given assembly and namespace that contain a matching feature attribute will be loaded as part of the model.

1.1.2 State variables

The state vocabulary of a model program consists of all fields declared within the model program's namespace. Both instance and static fields are included. Public and

nonpublic fields are included. Fields inherited from base classes (even those outside of the model namespace) are included.

Sometimes it is useful to exclude a field from state. This might arise for debugging or other purposes, including the implementation of the modeling tool itself. The `[ExcludeFromState]` attribute can be used to annotate fields that should not be part of state. This attribute may only be applied to fields.

Fields excluded from the set of state variables must have no behavior affect on the model, or unpredictable results will occur.

```
namespace MyModelProgram
{
    static class Operations
    {
        [ExcludeFromState]
        static int debugCount = 0;

        // ...
    }
}
```

1.1.3 Actions

The action vocabulary of the model is given by the `[Action]` attribute. The `[Action]` attribute applies only to methods.

There are three forms:

```
[Action]
[Action("label")]
[Action(Start="label", Finish="label")]
```

where *label* is a string in the form

```
label :== action-name [ "(" [ arg { "," arg }* ] ")" ]
action-name :== id
arg :== "this" | "result" | "_" | id
id :== ( letterOrDigit | '@') { letterOrDigit | '_' }*  // lexical
letterOrDigit :== ... as defined by Char.IsLetterOrDigit method in .NET
```

The simple form of the `[Action]` attribute indicates that a default action label should be used.

- If the target method has no outputs (that is, it has a `void` return value and does not have any `byref` or `out` parameters), then a single action with the same name

and arguments as the target method is produced. For example,

```
[Action] static void Foo(int x) { /* ... */ }
```

produces an action label in the form `Foo(x)`. The action is atomic.

- If the method has outputs, then the default vocabulary is a start action (whose arguments are the inputs) and a finish action (whose arguments are the outputs). In that case, the default action names are `methodName_Start` and `methodName_Finish`, respectively. For example,

  ```
  [Action]
  static bool Foo(int x, byref string s, out int val) { /* ... */ }
  ```

  produces in start action label with the input parameters

  ```
  Foo_Start(x, s)
  ```

  and a finish action label whose arguments are the outputs with `result` first and the remaining `out` or `byref` arguments in the order they appear in the target method.

  ```
  Foo_Finish(result, s, val)
  ```

  Note that that `byref` argument `s` appears in both input and output labels.

If a name (without parentheses) is given in the action attribute, then the same default behavior occurs as above, except that the user-provided action name is used instead of the method name. For example

```
[Action("Bar")]
static bool Foo(int x, byref string s, out int val) { /* ... */ }
```

produces labels `Bar_Start(x, s)` and `Bar_Finish(result, s, val)`.

The symbol `this` is used for the implicit first argument of an instance method. By default, the parameter `this` appears as the first input action parameter for an action whose target method is instance based.

The symbol `result` is used to denote the return value of the method. By default, the parameter `result` appears as the first output action parameter.

The extended forms of the `[Action]` attribute give the user control over the action names, order of parameters and the possibility of adding placeholder parameters. The idea is that in cases when you need to give more detail, you can use a form of the `[Action]` attribute that lets you define the action vocabulary more explicitly.

"Don't care" parameters Sometimes (especially when composition is involved, but also for certain test harnessing requirements) you want to include a placeholder parameter in an action's signature. In this case, you would indicate the placeholder with a underscore, _ in the signature.

```
[Action("Foo(_, x)")] static void
MyFoo(int x) { _ }
```

The action label takes two parameters, but the first parameter is ignored.

Reordering of arguments Overriding the default action label lets you specify a different order of arguments for the action label than is provided by the method being attributed. For example:

```
[Action("Foo(y, x)")]
static void MyFoo(int x, int y) { /* ... */ }
```

In this example the order of `x` and `y` are swapped. Note that all of the input parameters of the method must be mentioned in the action label. No output parameter may be mentioned in the input action label.

Similarly, the order of outputs may be customized:

```
[Action(Start="Foo_Start(x, s)",
        Finish="Foo_Finish(s, result)")]
static bool Foo(int x, byref string s, out int val) { /* ... */ }
```

In this example the order of `result` and `s` are swapped in the label for the output action. It is possible to omit some output parameters of the target method from the output action label if desired. Also, input parameters may appear in the output label.

"Split" actions Sometimes (especially in describing scenarios) it is necessary to say that a method that produces no outputs should be split into start and finish actions. In this case the start and finish actions can be given explicitly.

```
[Action(Start="Foo_Start(x)", Finish="Foo_Finish()")]
static void Foo(int x) { /* ... */ }
```

Multiple attributes per method More than one action attribute may appear for a given method. This is interpreted as separate actions that share a common update rule.

```
[Action("Foo(_, x)"]
[Action("Bar(x)"]
static void MyFoo(int x) { /* ... */  }
```

The example above defines two actions with differing signatures that share a common enabling condition and update. In the example, the `MyFoo` method is the implementation of the `Foo` action and the `Bar` action. (This case arises when modeling feature slices.)

Multiple methods per action The same action attribute (with user-provided label) may appear in front of methods split across features. This is interpreted as defining a single action whose preconditions and updates are composed. (This case arises when modeling feature slices.)

```
[Feature("Feature1")]
static class C1
{
    [Action("Foo(y, x)"]
    static void MyFoo(int x, int y) { /* ... */  }
}

[Feature("Feature2")]
static class C2
{
    [Action("Foo(_, z)"]
    static void MyBar(int z) { /* ... */  }
}
```

In the example above, a single action with two parameters is defined. The enabling condition of action `Foo` is the conjunction of the enabling conditions given in the body of `MyFoo` and `MyBar`. The update of action Foo is the state change produced by invoking `MyFoo` and `MyBar` in any order (the model must be written such that order of update of partial action methods does not matter).

1.1.4 Enabling conditions

Actions are enabled by Boolean-valued methods called *enabling conditions*, *preconditions*, or *guards*.

By convention, the enabling conditions of an action method have the same name as the action method plus `Enabled`, possibly followed by one or more decimal digits. For example:

```
static void FooEnabled() { return mode == Mode.Start; }
```

```
static void FooEnabled(int x) { return x > 0; }
static void FooEnabled2(int x) { return x < 100; }

[Action]
static void Foo(int x)
{
    mode = Mode.Running;
}
```

In this example, the methods `FooEnabled()`, `FooEnabled(int)` and `FooEnabled2(int)` are enabling conditions of action `Foo`.

Note that an action method may have more than one enabling condition by overloading. An enabling condition method may have fewer arguments than (but no more than) its associated action method. The type of each parameter of enabling condition method must match the type of the corresponding parameter in the action method.

A static enabling condition method may be used for an instance-based action method. If the static enabling condition method takes parameters, the first parameter must be represent the `this` parameter:

```
class Bar
{
    static void FooEnabled(Bar obj)
    {
        return obj != null && obj.mode == Mode.Start;
    }

    static void FooEnabled(Bar obj, int x)
    {
        return x > 0;
    }

    [Action]
    void Foo(int x)
    {
        this.mode = Mode.Running;
    }
}
```

1.1.5 Parameter Domains

Model programs may be given finite domains for exploration. This is done using the `[Domain]` attribute applied to method parameters. The form of the domain attribute is

```
[Domain("name")]
```

where *name* is a string naming a method, property or field in the containing class. An example of its use is:

```
[Action]
static void Foo([Domain("SomeInts")] int x)
{
    /* ... */
}

static readonly Set<int> SomeInts = new Set<int>(1, 2, 3, 4);
```

If a method name is given as the domain name and the action method is static, then the domain method must also be static. For an instance-based action method, domain methods may either be static or instance-based. Domain methods must take no parameters. Their return type must be a `Set<T>` where `T` is the type of the parameter being attributed.

Only types with finite domains have default domains. This includes `System.Boolean` and all `enum` types.

The domain attribute may be applied to instance methods to give the possible values of the implicit `this` paramter of the instance method.

1.1.6 Accepting State Conditions

Runs of a model program may only terminate in an accepting state. The predicates that determine whether a given state is an accepting state are given by the `[AcceptingStateCondition]` attribute. The target of this attribute may be a method, field or property.

An example of the use of this attribute is:

```
namespace MyModelProgram
{
    static class Contract
    {
        static Set<int> pendingRequests = Set<int>.EmptySet;

        [AcceptingStateCondition]
        static bool NoPendingRequests()
        {
            return pendingRequests.IsEmpty;
        }
    }
}
```

If the target of an `[AcceptingStateCondition]` attribute is a method, it is possible for the method to be a static method or an instance-method. In the case of an instance method, the state is accepting if the condition holds for all reachable instances of the class. The return type must be `System.Boolean`.

1.1.7 State Invariants

It is possible to indicate conditions called state invariants that must hold in every state. The failure of any state invariant indicates a modeling error.

The `[StateInvariant]` attribute indicates the target method, field or property must hold in every state.

An example of the use of this attribute is:

```
namespace MyModelProgram
{
    static class Contract
    {
        static Set<int> pendingRequests = Set<int>.EmptySet;
        static int nPendingRequests = 0;

        [StateInvariant]
        static bool IsConsisitent()
        {
            return pendingRequests.Count == nPendingRequests;
        }
    }
}
```

If the target of an `[StateInvariant]` attribute is a method, it is possible for the method to be a static method or an instance-method. In the case of an instance method, the state invariant is said to hold if the return value is true for all reachable instances of the class. The return type must be `System.Boolean`.

1.1.8 State Filters

A state filter is a Boolean condition that indicates that a state, although valid, should be excluded from exploration. The `[StateFilter]` attribute indicates that its target method, property or field is a state filter. A value of `true` indicates that the state will be included; `false` means exclusion from exploration.

```
namespace MyModelProgram
{
    static class Contract
```

```
    {
        static Set<int> pendingRequests = Set<int>.EmptySet;

        [StateFilter]
        static bool LimitNumberOfRequests()
        {
            return pendingRequests.Count < 4;
        }
    }
}
```

State filters are often used in conjuction with a `[Feature]` attribute so that they may selectively applied.

1.1.9 State properties

The `[StateProperty]` attribute indicates additional information that may be used by the exploration algorithm when deciding which states to explore. It may be applied to a method, property or field. Properties are named. If a name is not given in the attribute, the property name defaults to the target name.

```
namespace MyModelProgram
{
    static class Contract
    {
        static Set<int> pendingRequests = Set<int>.EmptySet;

        [StateProperty("NumberOfRequests")]
        static int RequestCount()
        {
            return pendingRequests.Count;
        }
    }
}
```

1.1.10 Requirements

It is sometimes useful to document the link between the elements of a model program and a natural language requirements document. We use an attribute in the form `[Requirement("`*string*`")]` for this purpose. The string *string* provides traceability back to the an external requirement source.

Requirement strings are printed in error contexts such as conformance failures and state invariant violations. They can also be used to check for coverage of requirements.

The `[Requirement]` attribute may be applied to any .NET attributable element. More than one `[Requirement]` attribute may be provided for an entity.

```
static class Bar
{
    [Requirement("XYZ spec 3.2.4: Foo parameter must be nonnegative")]
    static void FooEnabled(int x)
    {
        return x > 0;
    }

    [Action]
    static void Foo(int x)  { /* ... */ }
}
```

1.2 Data types

1.2.1 Compound Value

`CompoundValue`

Base class for data records with structural equality.

Syntax

```
public abstract class CompoundValue :  AbstractValue, IComparable
```

Methods

`CompareTo(Object)`
: Term order. Comparision is based on type and recursively on fields.

`ContainsObjectIds()`
: Determines if this value has an object id (that is, is of type `LabeledInstance`), or has a subvalue that has an object id (for example, a set of instances).

`Equality(CompoundValue, CompoundValue)`
: Term equality

`FieldValues()`
: Returns an enumeration of all (readonly) field values of this compound value in a fixed order.

`GreaterThan(CompoundValue, CompoundValue)`
: Term greater than

`GreaterThanOrEqual(CompoundValue, CompoundValue)`
: Term greater than or equal

`Inequality(CompoundValue, CompoundValue)`
: Inequality of terms

`LessThan(CompoundValue, CompoundValue)`
: Term less than

`LessThanOrEqual(CompoundValue, CompoundValue)`
: Term less than or equal

Properties

`AsTerm`
: Returns the term representation of this value.

`Sort`
: Returns the sort (abstract type) of this value.

Remarks

A `CompoundValue` is similar to a .NET `struct`, but unlike a `struct` subtypes of `CompoundValue` may implement tree structures. By construction, such values may be recursive but must have no cycles. (From the point of view of mathematical logic, values of this type can be thought of as a term whose function symbol is the type and whose arguments are the field values.)

`CompoundValues` are commonly used in modeling abstract state. They should be used instead of mutable classes whenever practical because they can reduce the amount of analysis required when comparing if two program states are equal. They also tend to promote a clear style, since they are read-only data structures. Aliased updates can thus be avoided.

Invariant: All field values (returned by the `FieldValues` method) must have a fixed order and contain elements of type `IComparable` that satisfy the predicate `IsAbstractValue(Object)`. All fields of subtypes of this class must be `readonly`. This invariant holds for subtypes of `CompoundValue` such as the collection types desribed below. The type parameter `T` for the collection types must support interface `IComparable`.

1.2.2 Set

`Set<T>`

An unordered collection of distinct values. This type is immutable and uses structural equality. Add/remove operations return a new set.

Syntax

```
public sealed class Set<T> :  CollectionValue<T>
```

Constructors

`Set<T>()`
: Constructs an empty set of type `T`. The static field `Set<T>.EmptySet` should be used instead of this constructor.

`Set<T>(IEnumerable<T>)`
: Constructs a set containing all enumerated elements.

`Set<T>(T[])`
: Constructs a set from the elements given in the argument list.

Methods

The following entries describe some of the methods of this type. Many more are described in the online library documentation, which is generated from the source code.

`Add(T)`
: Returns a set with all of the elements of the current set, plus the value given as an argument.

`BigIntersect(Set<Set<T>>)`
: (Static method.) Distributed intersection. Returns the set that is the intersection of all the sets contained in the argument value, i.e. the result contains the elements that are shared among all of the sets.

`BigUnion(Set<Set<T>>)`
: (Static method.) Distributed set union. Returns a set containing all of the elements of each set contained in the argument value.

`Choose()`
: Selects an arbitrary value from the current set using internal choice.

`Contains(T)`
: Tests whether the given element is found in the set.

`Difference(Set<T>)`
: Set difference. Same as operator `-`.

`Exists(Predicate<T>)`
: Existential quantification. Returns `true` if the predicate holds for at least one element of the current set.

`ExistsOne(Predicate<T>)`
: Unique quantification. Returns `true` if the predicate holds for exactly one element of the set, `false` otherwise.

`Forall(Predicate<T>)`
: Universal quantification. Returns `true` if the predicate holds for all elements of the set and `false` otherwise.

`Intersect(Set<T>)`
: Set intersection. Same as operator `*`. This is the set of all elements shared by the current set and the set given as the argument.

`IsProperSubsetOf(Set<T>)`
: Proper superset relation.

`IsProperSupersetOf(Set<T>)`
: Proper subset relation.

`IsSubsetOf(Set<T>)`
: Subset relation.

`IsSupersetOf(Set<T>)`
: Superset relation.

`Maximum()`
: Returns the greatest value in the set under the term ordering defined by `CompareValues(Object, Object)`.

`Minimum()`
: Returns the least value in the set under the term ordering defined by `CompareValues(Object, Object)`.

`Reduce<S>(Reducer<T, S>, S)`
: Iteratively applies a reducing function to map a set into a single summarized value.

`Remove(T)`
: Returns a set containing every element of this set except the value given as a parameter. If value is not in this set, then returns this set.

`Union(Set<T>)`
: Same as operator `+` (set union).

etc. ...

Many more methods are described in the online library documentation.

Properties

The following entries describe some of the properties of this type. Several more are described in the library documentation.

`Count`
: Returns the number of elements in the set (also known as the cardinality of the set).

`IsEmpty`
: Returns `true` if the set has no elements. Returns `false` otherwise.

etc. ...

Fields

`EmptySet`
: (Static.) The empty set of sort `T`.

Remarks

Set is an immutable type; Add/remove operations return a new set. Comparison for equality uses `Object.Equals`.

Formally, this data type denotes a pair (elementType, untyped set of values), where the element type is given by the type parameter `T`. As a consequence, sets are only equal if they are of the same sort (element type) and contain the same elements. For example, `Set<int>.EmptySet != Set<string>.EmptySet`

1.2.3 Map

`Map<T,S>`

A finite mapping of keys to values. This type is immutable and uses structural equality. Add/remove operations return a new map.

Syntax

```
public sealed class Map<T, S> :  CollectionValue<Pair<T, S>>
```

Constructors

The following entries describe some of the constructors of this type. Several more are described in the online library documentation.

`Map<T,S>()`
: Default constructor.. The static field `Map<T,S>.EmptyMap` is preferred and should be used instead of this constructor.

`Map<T,S>(T,S)`
: Creates a map from the key and value given as arguments.

`Map<T,S>(IEnumerable<Pair<T,S>>)`
: Creates a map from pairs of keys and values.

`Map<T,S>(Pair<T,S>[])`
: Create a map with a `params` array of key/value pairs.

etc. ...

Methods

The following entries describe some of the methods of this type. Many more are described in the online library documentation.

`Add(T, S)`
: Produces a map that contains all the key/value pairs of the current map, plus the given key/value pair.

`Add(Pair<T,S>)`
: Produces a map that contains all the key/value pairs of the current map, plus the given key/value pair.

`Contains(Pair<T,S>)`
: Tests whether the given key-value pair is found in the current map. See `ContainsKey(T)` instead if you want to check if a given key is in the map.

`ContainsKey(T)`
: Returns `true` if the map contains the given key, `false` otherwise

`Difference(Map<T,S>)`
: Map difference. Returns all of the key-value pairs in the current map that are not found in the map given as the argument. If the two maps share keys, the corresponding values must be the same, or an exception is thrown. This is the same as C# infix operator `-`.

`Override(T, S)`
: Map override. Returns a map with the same key-value pairs as the current map, plus the key-value pair given by the arguments. If the key exists in the current map, the value of this key in the result will be the value of the second argument to this method.

`Override(Map<T, S>)`
: Map override. Returns the map that contains all key-value pairs of the map `s` given as the argument and those key-value pairs of the current map for which there is no corresponding key in `s`. In other words, this operation combines the current map and `s` in a way that gives priority to `s`.

`Override(Pair<T, S>)`
: Map override. Returns the map that contains all key-value pairs of the current map along with key-value pair `d` given as the argument, except that if the key of `d` is in the current map, the value of `d` will replace (i.e., override) the corresponding value from the current map in the result.

`Remove(Pair<T, S>)`
: Returns a map with the same key-value pairs as the current map, except for the key-value pair given as the argument. If the key is present in the current map but is associated with a different value, then an exception is thrown.

`RemoveKey(T)`
: Returns a map with the same key-value pairs as the current map, except for the key-value pair whose key is given as the argument.

`RestrictKeys(IEnumerable<T>)`
: Returns a map consisting of the elements in the current map whose key is *not* in the collection given as an argument.

`RestrictValues(IEnumerable<S>)`
: Returns a map consisting of the elements in the current map whose value is *not* in values.

etc. ...

Many more methods are described in the online library documentation.

Properties

The following entries describe some of the properties of this type. Several more are described in the online library documentation.

`Count`
: The number of key-and-value pairs contained in the map.

`EmptyMap`
: The empty map of sort `<T, S>`.

`IsEmpty`
: Returns `true` if the map has no elements. Returns `false` otherwise.

`Keys`
: The set of keys in the map (i.e., the domain of the map)

`Values`
: The set of values in the map (i.e., the range of the map).

etc. ...

Remarks

Maps associate keys with values. Maps are similar to the .NET `Dictionary` objects but are immutable and use structural equality. Add/remove operations return new maps.

1.2.4 Sequence

`Sequence<T>`

An ordered collection of (possibly repeating) values. This type is immutable and uses structural equality. Add/remove operations return a new sequence.

Syntax

```
public sealed class Sequence<T> :  CollectionValue<T>
```

Constructors

`Sequence<T>()`
: Constructs an empty sequence. The static field `Sequence<T>.EmptySequence` is preferred instead of this constructor.

`Sequence<T>(IEnumerable<T>)`
: Constructs a sequence from the elements in a collection.

`Sequence<T>(T[])`
: Constructs a sequence from `params` arguments.

Methods

The following entries describe some of the methods of this type. Many more are described in the online library documentation.

`AddFirst(T)`
: Returns a sequence whose `Head` is the element given as the argument and whose `Tail` equals the current sequence.

`AddLast(T)`
: Returns a sequence whose `Front` equals the current sequence and whose and whose `Last` is the element given as the argument.

`BigConcatenate(IEnumerable<Sequence<T>>)`
: (Static method.) Distributed concatenation. Returns the sequence formed by concatenating each sequence from a collection of sequences is turn. Returns the empty sequence if the collection given as the argument is empty.

`Concatentate(Sequence<T>)` Sequence concatenation. Returns the sequence consisting of the elements of the current sequence followed by the elements of the sequence given as the argument. This is the same as the C# infix operator `+`.

`Contains(T)`
: Tests whether the given element is found in the sequence.

`IndexOf(T)`
: Returns the zero-based index of the first occurrence of the given element in the sequence, or `-1` if it doesn't occur.

`IsPrefixOf(Sequence<T>)`
: Returns `true` if the elements of the current sequence are the first elements of the sequence given as the argument, and `false` otherwise.

`LastIndexOf(T)`
: Returns the zero-based index of the last occurrence of the given object in the sequence, or `-1` if it doesn't occur.

`Remove(T)`
: Returns a sequence that is identical to the current sequence but without the first occurrence of the element given as the argument.

`Reverse()`
: Returns the sequence whose elements are the same as the current sequence but in reverse order.

`Unzip<T2>(Sequence<Pair<T, T2>>)`
: (Static method.) Returns a pair of sequences whose elements are drawn from a sequence of pairs.

`Zip<T2>Sequence<T>, Sequence<T2>`
: (Static method.) Returns the sequence of pairs of elements from the sequences given as arguments.

etc. ...

Many more methods are described in the online library documentation.

Properties

The following entries describe some of the methods of this type. Several more are described in the online library documentation.

`Count`
: Gets the number of elements contained in the sequence.

`Front`
: Return the subsequence of the current sequence where the last element is removed

`Head`
: Return the first element of the sequence. Throws an exception if the sequence is empty.

`IsEmpty`
: Returns `true` if the sequence has no elements. Returns `false` otherwise.

`Item(int)`
: Gets the element at the specified index. This is the C# indexer operator. Note: this is a linear-time lookup provided for convenience when sequences have only a few elements or when accessing elements near the beginning or end of a larger sequence. The data type `ValueArray<T>` is better suited to large numbers of elements that require frequent random access but infrequent addition or removal of elements. The data type `Sequence<T>` is better suited to frequent addition or deletion and recursive subdivision via `Head` anad `Tail` operations.

`Last`
: Returns the last element of the current sequence.

`Tail`
: Return the subsequence of the current sequence where the first element is removed. Throws an exception if the sequence is empty.

etc. ...

Several more properties are described in the online library documentation.

Fields

`EmptySequence`
: The empty sequence of sort `T`.

Remarks

Sequences contain indexable elements. Sequences are similar to ArrayLists, but unlike ArrayLists, they are immutable. Sequences are implemented as doubly linked lists (concatenation to the beginning or end is constant time) Lookup is linear time; if possible, callers should use `foreach(T val in sequence) ...` instead of `for(int i = 0; i < sequence.Count; i += 1) ... sequence[i] ...`

1.2.5 Value array

`ValueArray<T>`

Immutable type that provides structural equality for arrays.

Syntax

```
public sealed class ValueArray<T> :  CollectionValue<T>
```

Constructors

`ValueArray<T>(T[])`

Methods

The online library documentation describes the methods of this type. They are similar to the methods for the other NModel collection types.

Properties

The following entries describe some of the methods of this type. Several more are described in the online library documentation.

`Count`
: Returns the number of elements in the value array

`IsEmpty`
: Returns `true` if the value array has no elements. Returns `false` otherwise.

`Item(int)`

`Length`

etc. ...

1.2.6 Bag

`Bag<T>`

An unordered collection of possibly repeating elements. This is also known as a multiset. The type is immutable; add/remove methods return a new bag.

Syntax

```
public sealed class Bag<T> :  CollectionValue<T>
```

Constructors

`Bag<T>()`
: Constructs an empty bag. The static field `Bag<T>.EmptyBag` is preferred instead of using this form of the constructor.

`Bag<T>(IEnumerable<T>)`
: Constructs a bag with elements taken from a collection. The `Count` of the resulting bag will the same as the number of values in the collection given as the argument.

`Bag<T>(T[])`
: Constructs a bag with elements given as `params` arguments. The `Count` of the bag will be the same as the number of arguments to the constructor.

`Bag<T>(IEnumerable<Pair<T, int>>)`
: Constructs a bag with elements and their corresponding multiplicities given as a pair enumeration.

`Bag<T>(Pair<T, int>[])`
: Constructs a bag with `params` arguments that are pairs of elements and their corresponding multiplicities.

Methods

The following entries describe some of the methods of this type. Many more are described in the online library documentation.

`Add(T)`
: Creates a bag that is the same as this bag except that the multiplicity of the value given as the argument is one larger.

`CountItem(T)`
: Returns the number of times the value given as the argument appears in this bag. This number is called the item's multiplicity.

`Difference(Bag<T>)`
: Creates a bag where the multiplicity of each element is the difference of the multiplicities in the current bag and the bag `s` given as the argument; that is, it returns a bag where all the elements of `s` have been removed from this bag. This is the same as the C# infix operator `-`.

`Intersection(Bag<T>)`
: Bag intersection. Returns the bag containing the elements that are shared by the current bag and the bag given as the argument. The multiplicities of the result are pairwise minimums of those of the arguments. This is the same as the C# infix operator `*`.

`Remove(T)`
: Creates a bag with the same elements as the current bag, except that the multiplicity of the element `x` given as the argument is decremented by one. This operation returns the current bag if the multiplicity of `x` is zero.

`RemoveAll(T)`
: Creates a bag with the same elements as this bag, except that all occurrences of the element given as the argument are omitted.

`Union(Bag<T>)`
: Creates a bag where the multiplicities of each element are the sum of the multiplicities of the elements of the current bag and the bag given as the argument.

etc. ...

Many more methods are described in the online library documentation.

Properties

The following entries describe some of the properties of this type. Several more are described in the online library documentation.

`Count`
: Returns the number of elements in the bag. This is sum of all multiplicities of the unique elements in this bag..

`CountUnique`
: Returns the number of unique elements in this bag. (Note: This is less than the number of elements given by `Count` if some elements appear more than once in the bag.)

`IsEmpty`
: Returns `true` if the bag has no elements. Returns `false` otherwise.

`Keys`
: Returns a set of all elements with multiplicity greater than zero

etc. ...

Several more properties are described in the online library documentation.

Fields

`EmptyBag`
: (Static field.) The bag of type T that contains no elements.

Remarks

For any value x, the multiplicity of x is the number of times x occurs in the bag, or zero if x is not in the bag.

The data type is immutable; add/remove operations return a new bag.

Equality is structural. Two bags are equal if they contain the same elements with the same multiplicities. Order does not affect equality.

1.2.7 Pair

`Pair<T,S>`

Binary tuples with structural equality.

Syntax

```
public struct Pair<T, S> :  IAbstractValue, IComparable
```

Constructors

`Pair<T, S>(T, S)`
: Initializes a new instance of a pair with the given arguments

Methods

`Equality(Pair<T, S>, Pair<T, S>)`
: Deep structural equality on Pairs

etc. ...

Properties

`First`
: The first value

`Second`
: The second value

etc. ...

1.2.8 Triple

`Triple<T,S,R>`

Triples with structural equality.

Syntax

```
public struct Triple<T, S, R> :  IAbstractValue, IComparable
```

Constructors

`Triple<T, S, R>(T, S, R)`
: Initializes a new instance of a triple with the given arguments

Methods

`Equality(Triple<T, S, R>, Triple<T, S, R>)`
: Structural equality.

etc. ...

Properties

`First`
: The first value

`Second`
: The second value

`Third`
: The third value

etc. ...

1.2.9 Labeled Instance

`LabeledInstance<T>`

Base class for types with instance fields as state variables. Each type `T` that includes instance fields that act as state variables of a model program must inherit the base class `LabeledInstance<T>`.

Syntax

```
public class LabeledInstance<T> :  LabeledInstance
```

Constructors

`LabeledInstance<T>()`
: Each subtype `T` of `LabeledInstance<T>` must provide a public default constructor. This constructor may not change state.

Methods

`CompareTo(Object)`

`ContainsObjectIds()`
: Returns `true`.

`Create()`
: (Static method.) Factory method that allocates a new object. This factory must be used in the model instead of the public constructor. Most models will create instances via the `[Domain("new")]` attribute applied to an action argument instead of this factory method. The `Create` method may only be invoked within the context of an action method.

`GetSort()`
: The sort (abstract type) of `T`. Sorts are used to match types across model programs.

`Initialize()`
: Each subtype `T` of `LabeledInstance<T>` must override the `Initialize` method. This method resets all of the instance fields of the current object to their initial values.

Properties

`AsTerm`
: Returns the term representation of this value.

`Label`

`Sort`
: Returns the sort (abstract type) of this value

Remarks

You must use a type `T` that is a subtype of `LabeledInstance<T>` to program with objects in NModel. See chapter 3.2.

1.3 Action terms

This section explains how to construct and access the terms used to represent actions.

The `Action` data type is derived from the underlying data type `CompoundTerm` that is derived from `Term`. It represents an immutable type whose values are actions.

The main methods used to create and access actions are the following. The same methods can also be used for the `CompoundTerm` type but are most relevant when manipulating actions.

`Create(string, params IComparable[])`
: Static method that creates an action with the given string name and an array of .NET values. Values that are not terms are converted into terms.

  For example, `Action.Create("A", 5, new Set<string>("c","b"))` creates the action `A(5,Set<string>("c","b"))`.

`Parse(string)`
: Static method that parses the given string into an action.

  For example, `Action.Parse("A(5, Set<string>(\"c\", \"b\"))` creates the action `A(5,Set<string>("c","b"))`.

`Name`
: Returns the name of (the function symbol of) the action.

  For example given an action `a` as above, `a.Name` is the string `"A"`.

`this[int]`
: An action has an *indexer* that, for each valid argument position k of an action `a`, takes the k'th subterm of `a` and returns the underlying .NET value as an `IComparable`. If the k'th subterm of `a` does not have a valid, context independent, interpretation as a .NET value then the returned value is the term itself.

  For example, given `a` as above, `(int)a[0]` is the integer 5, `(Set<string>)a[1]` is the string set containing the strings `"c"` and `"b"`, `a[2]` throws an `InvalidOperationException`.

  As another example, let `a` be the action `Foo(bar(3),"baz")`, then `(Term)a[0]` is the term `bar(3)` and `(string)a[1]` is the string `"baz"`.

Chapter 2

Command reference

This appendix describes the command line options for the visualization and analysis tools `mpv` (Model Program Viewer) and `mp2dot` (Model Program to Dot), the test generator tool `otg` (Offline Test Generator), and the test runner tool `ct` (Conformance Tester).

2.1 Model program viewer, `mpv`

Use `mpv` to visualize and analyze the behavior of one or more model programs. The `mpv` tool performs *exploration*, which generates a *finite state machine* (FSM) from a (possibly "infinite") model program. The tool displays the graph of the generated FSM. The tool explores and displays the *composition* of all the model programs named on its command line, which can be C# model programs or FSMs. The tool provides many options (on the command line and in its GUI) for adjusting the appearance of the displayed graph, and for saving and restoring the results of exploration. It also provides facilities for inspecting the states (program variables and their values), and for exploring programs interactively (incrementally, working forward from some state of interest).

By searching the FSM, `mpv` can perform *safety analysis* that checks whether the system can reach forbidden (unsafe) states, and *liveness* analysis that identifies *dead states* from which goals cannot be reached. The `mpv` tool can also check temporal properties expressed as FSMs by using composition.

The `mpv` tool requires that the GLEE graph layout software be installed. The other tools do not require GLEE.

2.1.1 Usage

```
mpv [/reference:<string>]* [/mp:<string>]* [/initialTransitions:<int>]*
```

```
[/transitionLabels:{None|ActionSymbol|Action}]* [/nodeLabelsVisible[+|-]]*
[/initialStateColor:<string>]* [/hoverColor:<string>]*
[/selectionColor:<string>]* [/deadStateColor:<string>]* [/deadStatesVisible[+|-]]*
[/unsafeStateColor:<string>]* [/maxTransitions:<int>]* [/loopsVisible[+|-]]*
[/mergeLabels[+|-]]* [/acceptingStatesMarked[+|-]]*
[/stateShape:{Box|Circle|Diamond|Ellipse|Octagon|Plaintext}]*
[/direction:{TopToBottom|LeftToRight|RightToLeft|BottomToTop}]*
[/combineActions[+|-]]* [/livenessCheckIsOn[+|-]]* [/safetyCheckIsOn[+|-]]*
[/testSuite:<string>]* [/fsm:<string>]* [/startTestAction:<string>]*
[/group:<string>]*
<model>* @<file>
```

2.1.2 Examples

```
mpv @mpv_args.txt
mpv /fsm:M1.txt /fsm:M2.txt
mpv /testSuite:ContractTest.txt
mpv /r:NewsReaderUI.dll NewsReader.Factory.Create
mpv /r:NewsReaderUI.dll /mp:NewsReader
mpv /r:Controller.dll Reactive.Factory.Create /safetyCheckIsOn+
mpv /r:Controller.dll /mp:Reactive /safetyCheckIsOn+
```

2.1.3 Options

`/?, /help`
: Displays usage information and exits.

`[/reference:<string>]*`
: Referenced assemblies. (Short form: `/r`)

`[/mp:<string>]*`
: Model programs given in the form `M` or `M[F1,...,Fn]` where `M` is a model program name (namespace) and each `Fi` is a feature in `M`. Multiple model programs are composed into a product. No factory method is needed if this option is used.

`[/initialTransitions:<int>]*`
: Number of transitions that are explored initially up to maxTransitions. Negative value implies no bound. Default value: '-1'

`[/transitionLabels:None|ActionSymbol|Action]*`
: Determines what is shown as a transition label. Default value: 'Action'

`[/nodeLabelsVisible[+|-]]*`
: Visibility of node labels. Default value: '+'

`[/initialStateColor:<string>]*`
Background color of the initial state. Default value: 'LightGray'

`[/hoverColor:<string>]*`
Line and action label color to use when edges or nodes are hovered over. Default value: 'Lime'

`[/selectionColor:<string>]*`
Background color to use when a node is selected. Default value: 'Blue'

`[/deadStateColor:<string>]*`
Background color of dead states. Dead states are states from which no accepting state is reachable. Default value: 'Yellow'

`[/deadStatesVisible[+|-]]*`
Visibility of dead states. Default value: '+'

`[/unsafeStateColor:<string>]*`
Background color of states that violate a safety condition (state invariant). Default value: 'Red'

`[/maxTransitions:<int>]*`
Maximum number of transitions to draw in the graph. Default value: '100'

`[/loopsVisible[+|-]]*`
Visibility of transitions whose start and end states are the same. Default value: '+'

`[/mergeLabels[+|-]]*`
Multiple transitions between same start and end states are shown as one transition with a merged label. Default value: '+'

`[/acceptingStatesMarked[+|-]]*`
Mark accepting states with a bold outline. Default value: '+'

`[/stateShape:Box|Circle|Diamond|Ellipse|Octagon|Plaintext]*`
State shape. Default value: 'Ellipse'

`[/direction:TopToBottom|LeftToRight|RightToLeft|BottomToTop]*`
Direction of graph layout. Default value: 'TopToBottom'

`[/combineActions[+|-]]*`
Whether to view matching start and finish actions by a single label. Default value: '-'

`[/livenessCheckIsOn[+|-]]*`
Mark states from which no accepting state is reachable in the current view. Default value: '-'

`[/safetyCheckIsOn[+|-]]*`
: Mark states that violate a safety condition (state invariant). Default value: '-'

`/testSuite:<string>]*`
: File name of a file containing a sequence of actions sequences (test cases) to be viewed.

`[/fsm:<string>]*`
: File name of a file containing the term representation `fsm.ToTerm()` of an `fsm` (object of type FSM). Multiple FSMs are composed into a product.

`[/group:<string>]*`
: Name of a state property to use as the abstraction function for grouping, as described in section 3.3.

`[/startTestAction:<string>]*`
: Name of start action of a test case. This value is used only if a `testSuite` is provided. Default value: 'Test'

`<model>*`
: Fully qualified names of factory methods returning an object that is an instance of `ModelProgram`. Multiple model programs are composed into a product. No factory method arguments are needed if the `/mp:` option is used instead.

`@<file>`
: Read response file for more options.

2.1.4 File menu

`Save Settings...`
: Saves the values of all the options (above) in a *response file* so they can be restored into another `mpv` session with the `@<file>` command line option.

2.1.5 Toolbar

The `mpv` tool provides several operations on a *toolbar*, a row of icons across the top of its window. Some icons include a down arrow you can click to display a drop-down menu. Hovering the mouse pointer over a toolbar icon displays a *tooltip*, text that identifies its function. Here are descriptions of the toolbar operations, each labeled by its tooltip.

`(Model program name)`
: The name of the model program whose FSM is currently displayed. *M[F1,F2]* indicates model program *M* with features *F1* and *F2* included. *M1* × *M2* indicates the product of *M1* and *M2* formed by *composition*. Selecting *M1* or *M2*

from the drop down menu displays the FSM of that program projected onto the product.

`Zoom In`
: Enlarge the displayed FSM within the window. Use the scroll bars at bottom and right to pan the enlarged graph, or use `Pan` (below). You can also enlarge the entire window by dragging on its sides or corners; the graph will enlarge to fit.

`Save as image...`
: Save the displayed FSM in a file. From the drop-down menu, `Save as Image...` saves an image in JPG format, `Save as Dot...` saves a text file in the *dot* language used by *Graphviz*, and `Save as FSM...` saves a text file in the term representation used by NModel, that can be restored into another `mpv` session with the `/fsm:<file>` command line option.

`Zoom Out`
: Reduce the displayed FSM within the window.

`Pan`
: Drag enlarged FSM in window, by holding down left mouse button while moving mouse. Alternative to scroll bars.

`Print`
: Print the FSM, opens Print dialog.

`Merge Labels`
: Toggles the `mergeLabels` option. When the option is '+', *True*, the graph appears less cluttered.

`Show Self-Loops`
: Toggles the `loopsVisible` option. When the option is '-', *False*, the graph appears less cluttered.

`Transition Labels`
: Cycles through the `transitionLabels` options: `Action` (most verbose), `ActionSymbol` (less verbose), `None`. The `Action` can always be displayed by hovering the mouse over a transition arc in the graph.

`Combine Start and Finish Actions`
: Toggles the `combineActions` option. When the option is '+', *True*, the graph appears less cluttered.

`Layout Direction`
: The drop-down menu offers the `direction` options: `TopToBottom`, `LeftToRight`, `RightToLeft`, `BottomToTop`.

`State Viewer`
: Displays (or hides) the state viewer in a panel on the right side of the window. The state viewer shows the state variables and their values in the state that is selected by left-clicking in the displayed graph. The state can always be displayed by hovering the mouse over a state bubble in the graph.

`Advanced Properties`
: Displays (or hides) the display of all the option values in a panel on the right side of the window. You can toggle/cycle/edit the option values in the panel. This panel also shows *Exploration statistics*: the number of *States*, *Transitions*, *Accepting States*, *Unsafe States*, and *Dead States*, so you do not need to search the graph by eye.

2.1.6 Selections and highlighting

Hovering the mouse over a state bubble highlights that state and shows the state variables and their values. Hovering the mouse over a transition arc highlights the arc displays its action (the action method invocation, and its return value if there is one).

Left clicking on a state bubble selects that state. The selected state is indicated by the `SelectionColor`. The state variables and their values in the selected state appear in the State Viewer panel, if it has been toggled on. Context menu items and keyboard shortcuts apply to the selected state.

2.1.7 Context menu

Right-clicking on a state bubble selects that state and displays a context menu with these entries:

```
Show Outgoing (Enter)
Hide Outgoing (Backspace)
Show Reachable (Insert)
Hide Reachable (Delete)
Select Next (n)
Select Previous (p)
```

To explore interactively, select the initial state, press the `Delete` key to hide all the transitions and other states, press the `Enter` key to show the transitions enabled in the initial state, press the `n` and `p`keys to select another state, and so on.

2.2 Model program to dot, mp2dot

The mp2dot tool is an alternative to mpv that does not depend on GLEE or Windows.Forms. It is a command-line program with no graphical user interface that produces an output file in the dot graph layout language. To view this output, you must use other programs. For example, the dot program generates PostScript from a dot file, which can then be displayed by any PostScript viewer.

The mp2dot tool accepts all of the mpv command line options, so it can use any mpv response file.

In addition to all of the mpv options, mp2dot supports two more: /dotFileName and /machineFileName. These two options achieve the same effect as the Save as Dot... and Save as FSM... options on the mpv toolbar (section 2.1.5). If the option is absent, that file is not written.

The mp2dot tool always writes exploration statistics to the console, even when no output files are requested, as in this example:

```
>mp2dot @mpv_safety_liveness.txt
  121 states
  239 transitions
    2 accepting states
   61 dead states
    4 unsafe states
```

2.2.1 Usage

```
mp2dot [/dotFileName:<string>] [/machineFileName:<string>]
[/reference:<string>]* [/mp:<string>]* [/initialTransitions:<int>]*
[/transitionLabels:{None|ActionSymbol|Action}]* [/nodeLabelsVisible[+|-]]*
[/initialStateColor:<string>]* [/hoverColor:<string>]*
[/selectionColor:<string>]* [/deadStateColor:<string>]* [/deadStatesVisible[+|-]]*
[/unsafeStateColor:<string>]* [/maxTransitions:<int>]* [/loopsVisible[+|-]]*
[/mergeLabels[+|-]]* [/acceptingStatesMarked[+|-]]*
[/stateShape:{Box|Circle|Diamond|Ellipse|Octagon|Plaintext}]*
[/direction:{TopToBottom|LeftToRight|RightToLeft|BottomToTop}]*
[/combineActions[+|-]]* [/livenessCheckIsOn[+|-]]* [/safetyCheckIsOn[+|-]]*
[/testSuite:<string>]* [/fsm:<string>]* [/startTestAction:<string>]*
[/group:<string>]*
<model>* @<file>
```

2.2.2 Examples

```
mp2dot /dotFileName:graph.dot @mpv_args.txt
mp2dot /machineFileName:M1xM2.txt /fsm:M1.txt /fsm:M2.txt
```

```
mp2dot /dot:NewsReader.dot /r:NewsReaderUI.dll /mp:NewsReader
```

2.2.3 Options

The `mp2dot` tool supports all of the `mpv` options that are described in section 2.1.3. It also supports these options:

`[/dotFileName:<string>]`
: File where the dot output is saved. If this option is absent, no dot output is written. (Short form: `/dot`)

`[/machineFileName:<string>]`
: File where the FSM is saved. If this option is absent, no FSM output is written. (Short form: `/machine`)

2.3 Offline test generator, otg

The otg tool generates an offline test suite that achieves link coverage of the finite state machine (FSM) generated from a model program. The test suite can then be executed by the Conformance Tester, ct.

It is typical to compose a scenario expressed as an FSM with a contract model program in C# in order to limit the size of the generated FSM.

2.3.1 Usage

```
   otg [/reference:<string>]* [/mp:<string>]* [/file:<string>]* [/append[+|-]]*
[/fsm:<string>]* <model>* @<file>
```

2.3.2 Examples

```
otg @otg_args.txt
otg /r:ClientServer.dll ClientServer.Factory.Create /fsm:Scenario.txt
otg /r:ClientServer.dll /mp:ClientServer /fsm:Scenario.txt
otg /r:ClientServer.dll ClientServer.Factory.Create /file:ContractTest.txt
otg /r:ClientServer.dll /mp:ClientServer /file:ContractTest.txt
```

2.3.3 Options

`/?, /help`
: Displays usage information and exits.

`[/reference:<string>]*`
: Referenced assemblies. (Short form: `/r`)

`[/mp:<string>]*`
: Model programs given in the form `M` or `M[F1,...,Fn]` where `M` is a model program name (namespace) and each `Fi` is a feature in `M`. Multiple model programs are composed into a product. No factory method is needed if this option is used.

`[/file:<string>]*`
: File where test suite is saved. The console is used if no file is provided. (Short form: `/f`)

`[/append[+|-]]*`
: If false the file is overwritten, otherwise the generated test suite is appended at the end of the file. Default value: "-" (Short form: `/a`)

`[/fsm:<string>]*`

File name of a file containing the term representation `fsm.ToTerm()` of an `fsm` (object of type FSM). Multiple fsms are composed into a product.

`<model>*`

Fully qualified names of factory methods returning an object that is an instance of `ModelProgram`. Multiple models are composed into a product. No factory method arguments are needed if the `/mp:` option is used instead.

`@<file>`

Read response file for more options.

2.4 Conformance tester, `ct`

Execute tests using the test runner `ct`. You must write a test harness in C# that couples your implementation to `ct`. You can provide a test suite generated from a model program by the Offline Test Generator `otg`, or `ct` can generate test cases on-the-fly from a model program as the test run executes. You can write a custom *strategy* in C# that `ct` uses to maximize coverage according to criteria you define.

An NModel test harness is called a *stepper*. To test a reactive (event-driven) system, you can write an *asynchronous stepper* that handles *observable actions* (events) in the implementation that are not under the tester's direct control.

2.4.1 Usage

```
ct /iut:<string> [/modelStepper:<string>] /reference:<string>+ [/mp:<string>]*
[/coverage:<string>]* [/steps:<int>]* [/maxSteps:<int>]* [/runs:<int>]*
[/observableAction:<string>]* [/cleanupAction:<string>]*
[/internalAction:<string>]* [/waitAction:<string>]* [/timeoutAction:<string>]*
[/timeout:<int>]* [/continueOnFailure[+|-]]* [/logfile:<string>]*
[/randomSeed:<int>]*  [/overwriteLog[+|-]]* [/testSuite:<string>]*
[/fsm:<string>]* [/startTestAction:<string>]* <model>* @<file>
```

2.4.2 Examples

```
ct @ct_args.txt
ct /r:Stepper.dll /iut:ClientServerImpl.Stepper.Create /testSuite:ContractTest.txt
ct /r:ClientServer.dll ClientServer.Factory.Create /r:Stepper.dll ^
    /iut:ClientServerImpl.Stepper.Create /fsm:Scenario.txt /runs:1
ct /r:ClientServer.dll /mp:ClientServer /r:Stepper.dll ^
    /iut:ClientServerImpl.Stepper.Create /fsm:Scenario.txt /runs:1
```

2.4.3 Options

`/?, /help`
: Displays usage information and exits.

`/iut:<string>`
: Implementation under test, a fully qualified name of a factory method that returns an object that implements `IStepper`.

`[/strategy:<string>]`
: A fully qualified name of creator method that takes arguments `(ModelProgram modelProgram, string[] coverage)` and returns an object that implements `IStrategy`.

If left unspecified the default model stepper is used that ignores coverage point names (if any). (If a `testSuite` is provided, this option is ignored.)

`/reference:<string>+`
Referenced assemblies. (Short form: `/r`)

`[/mp:<string>]*`
Model programs given in the form `M` or `M[F1,...,Fn]` where `M` is a model program name (namespace) and each `Fi` is a feature in `M`. Multiple model programs are composed into a product. No factory method is needed if this option is used.

`[/coverage:<string>]*`
Coverage point names used by model stepper. (If a `testSuite` is provided, this option is ignored.)

`[/steps:<int>]*`
The desired number of steps that a single test run should have. After the number is reached, only cleanup tester actions are used and the test run continues until an accepting state is reached or the number of steps is `MaxSteps` (whichever occurs first). 0 implies no bound and a test case is executed until either a conformance failure occurs or no more actions are enabled. (If a testSuite is provided, this value is set to 0.) Default value: '0'

`[/maxSteps:<int>]*`
The maximum number of steps that a single test run can have. This value must be either 0, which means that there is no bound, or greater than or equal to steps. Default value: '0'

`[/runs:<int>]*`
The desired number of test runs. Testing stops when this number has been reached. Negative value or 0 implies no bound. (If a `testSuite` is provided, this value is set to the number of test cases in the test suite.) Default value: '0'

`[/observableAction:<string>]*`
Action symbols of actions controlled by the implementation. Other actions are controlled by the tester. (Short form: `/o`)

`[/cleanupAction:<string>]*`
Action symbols of actions that are used to end a test run during a cleanup phase. Other actions are omitted during a cleanup phase. (Short form: `/c`)

`[/internalAction:<string>]*`
Action symbols of tester actions that are not shared with the implementation and are not used for conformance evaluation. Other tester actions are passed to the implementation stepper. (Short form: `/i`)

`[/waitAction:<string>]*`
: A name of an action that is used to wait for observable actions in a state where no controllable actions are enabled. A wait action is controllable and internal and must take one integer argument that determines the time to wait in milliseconds during which an observable action is expected.

`[/timeoutAction:<string>]*`
: A name of an action that happens when a wait action has been executed and no obsevable action occurred within the time limit provided in the wait action. A timeout action is observable and takes no arguments.

`[/timeout:<int>]*`
: The amount of time in milliseconds within which a tester action must return when passed to the implementation stepper. Default value: '10000'

`[/continueOnFailure[+|-]]*`
: Continue testing when a conformance failure occurs. Default value: '+'

`[/logfile:<string>]*`
: Filename where test results are logged. The console is used if no logfile is provided.

`[/randomSeed:<int>]*`
: A number used to calculate the starting value for the pseudo-random number sequence that is used by the global choice controller to select tester actions. If a negative number is specified, the absolute value is used. If left unspecified or if 0 is provided a random number is generated as the seed. Default value: '0' (Short form: `/seed`)

`[/overwriteLog[+|-]]*`
: If true the log file is overwritten, otherwise the testresults are appended to the logfile Default value: '+'

`[/testSuite:<string>]*`
: File name of a file containing a sequence of actions sequences to be used as the test suite.

`[/fsm:<string>]*`
: File name of a file containing the term representation `fsm.ToTerm()` of an `fsm` (object of type FSM). Multiple FSMs are composed into a product.

`[/startTestAction:<string>]*`
: Name of start action of a test case. This value is used only if a testSuite is provided. The default 'Test' action sybmol is considered as an internal test action symbol. If another action symbol is provided it is not considered as being internal by default. Default value: 'Test'

`<model>*`

Fully qualified names of factory methods returning an object that is a subclass of `ModelProgram`. Multiple models are composed into a product. No factory method arguments are needed if the `/mp:` option is used instead.

`@<file>`

Read response file for more options.

Chapter 3

Further Reading

The NModel framework is more fully explained in the book, *Model-Based Software Testing and Analysis with C#*, by Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte, Cambridge University Press, 2008.

Several cross-references here actually refer to that book.

3.1 Types for model programs

See section 10.2 in the book.

3.2 Modeling objects

See chapter 15 in the book.

3.3 State grouping

See section 11.2.5 in the book.