Automated Testing in Your Favorite Language

Jonathan Jacky

University of Washington, Seattle, USA
Modeled Computation, LLC

Acknowledgments

Foundations of Software Engineering at Microsoft Research:

Mike Barnett, Nikolaj Bjorner, Colin Campbell, Wolfgang Grieskamp, Yuri Gurevich, Lev Nachmanson, Wolfram Schulte, Nikolai Tillman, Margus Veanes

NModel:

Colin Campbell, Margus Veanes
recently Juhan Ernits, Ofer Rivlin

Behavior

We need to test *behavior*: ongoing activities that may exhibit history dependence and nondeterminism. For example: communication protocols, embedded controllers, user interfaces, ...

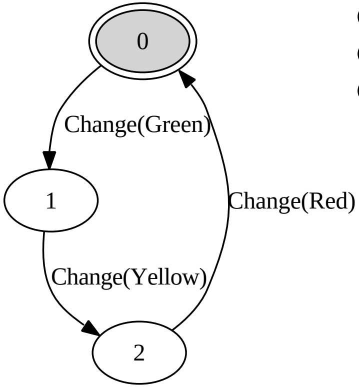
We represent behavior with *traces*: sequences of *actions* with arguments. Specify a system by describing which traces are allowed, and which are forbidden.

Allowed	Forbidden	Forbidden	Forbidden	Allowed	Allowed	Allowed
Push(0)	Push(0)	Pop(0)	Push(0)	Push(0)	Push(0)	Push(0)
Pop(0)	Pull(0)	Push(0)	Pop(1)	Push(1)	Pop(0)	Push(1)
				Push(2)	Push(1)	Push(2)
				Pop(2)	Pop(1)	Push(3)
				Pop(1)	Push(2)	Push(4)
				Pop(0)	Pop(2)	Push(5)
						etc...

Finite State Machines

Finite state machines can describe behaviors where the action arguments have a finite (small) number of values (no numbers, strings, ...).

Traffic light - a finite system, but with an infinite number of traces.

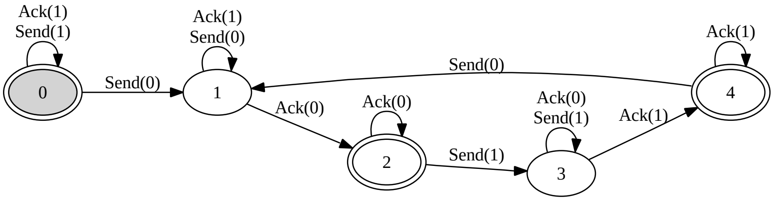


Change(Green)
Change(Yellow)
Change(Red)

Change(Green)
Change(Yellow)
Change(Red)
Change(Green)
Change(Yellow)
Change(Red)
Change(Green)
Change(Yellow)
Change(Red)
etc...

Finite State Machines

Alternating bit protocol - a finite system with nondeterminism.



Send(0)
Ack(0)
Send(1)
Ack(1)

Send(1)
Send(1)
Ack(1)
Send(0)
Ack(1)
Ack(1)
Send(0)
Ack(0)

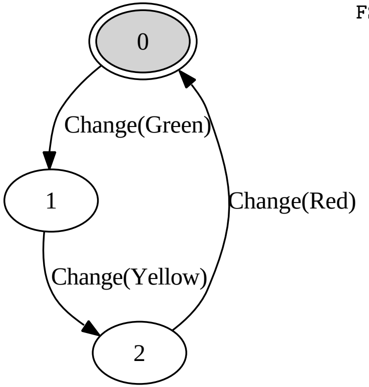
Send(1)
Send(1)
Ack(1)
Send(1)
Ack(1)
Send(1)

Send(1)
Ack(1)
Send(1)
Ack(1)
Ack(1)
Send(0)
Ack(0)

etc...

Finite State Machines

In NModel, FSMs are coded in text, as the graph of the machine.



```
FSM(Red, AcceptingStates(Red),
  Transitions(t(Red, Change(Green), Green),
   t(Green, Change(Yellow), Yellow),
   t(Yellow, Change(Red), Red)))
```

Model programs

Model programs can describe behaviors where the action arguments can have an “infinite” (very large) number of values.

A model program consists of *state variables*, *action methods* and *enabling conditions*.

In NModel, model programs are coded in C#.

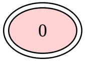
```
 // State
 internal static Sequence<int> stack = new Sequence<int>();

 // Push is always enabled
[Action] static void Push(int x) { stack = stack.AddFirst(x); }

 // Pop requires enabling condition
 static bool PopEnabled(int x) {
     return !stack.IsEmpty && stack.Head == x;
 }

 [Action] static void Pop(int x) { stack = stack.Tail; }
```

Exploration

Exploration generates an FSM from a model program, by starting at the initial state and executing some enabled actions, until ...

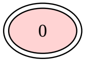
Exploration

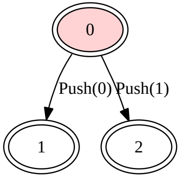
Exploration generates an FSM from a model program, by starting at the initial state and executing some enabled actions, until ...



Exploration

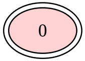
Exploration generates an FSM from a model program, by starting at the initial state and executing some enabled actions, until ...

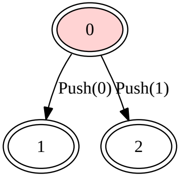


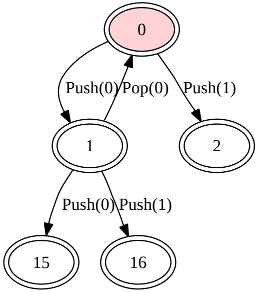


Exploration

Exploration generates an FSM from a model program, by starting at the initial state and executing some enabled actions, until ...

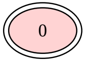


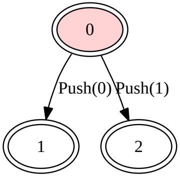


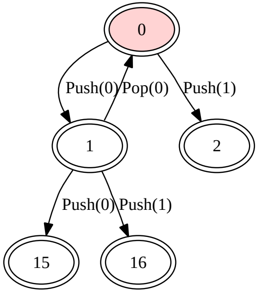


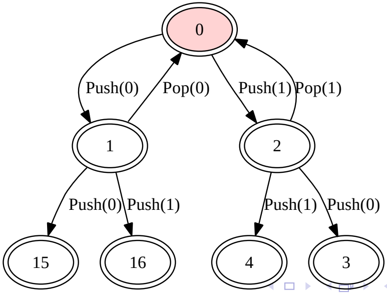
Exploration

Exploration generates an FSM from a model program, by starting at the initial state and executing some enabled actions, until ...







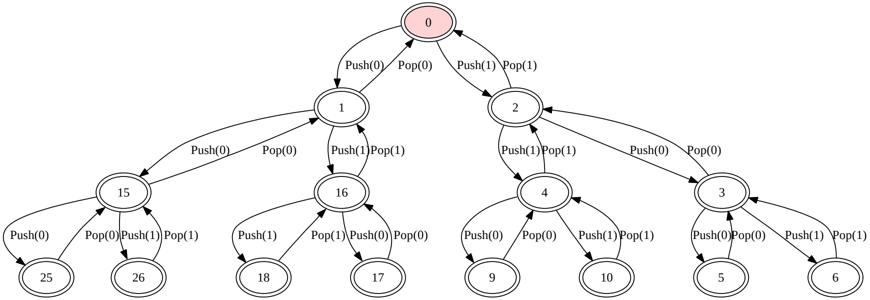


etc ...

Exploration

We must limit exploration of infinite programs. Finite *domains* limit the width of the graph. Here the *state filter* limits its depth.

```
static Set<int> Elements = new Set<int>(0, 1);
[Action] static void Push([Domain("Elements")] int x) ...

[StateFilter] static bool IsDepthLimited() { return Model.stack.Count < 4; }
```



Analysis

A *state invariant* should be true in every state. Define state invariants for safety analysis.

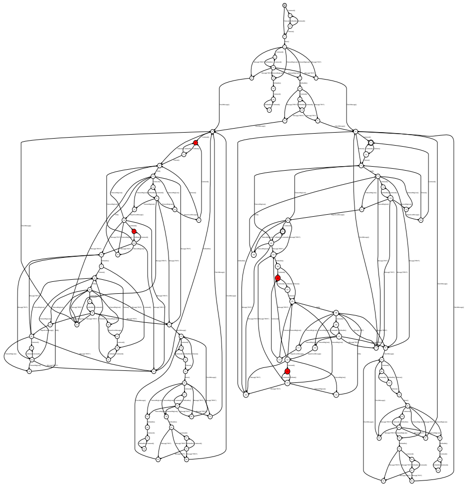
```
[StateInvariant]
static bool CalibrateInRange()
{
    return (!CalibrateEnabled() || buffer == InRange);
}
```

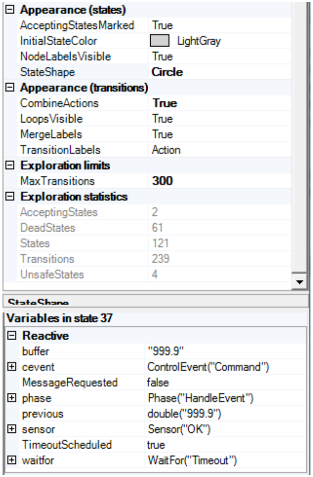
A model program should only stop in an *accepting state*. Define accepting state conditions for liveness analysis.

```
[AcceptingStateCondition]
static bool SafeCalibrateEnabled()
{
    return (CalibrateEnabled()
        && buffer == InRange
        && previous == double.Parse(InRange));
}
```

Safety Analysis

Exploration can search for *unsafe states* that violate the state invariant.





⊟ **Appearance (states)**	
AcceptingStatesMarked	True
InitialStateColor	▭ LightGray
NodeLabelsVisible	True
StateShape	**Circle**
⊟ **Appearance (transitions)**	
CombineActions	**True**
LoopsVisible	True
MergeLabels	True
TransitionLabels	Action
⊟ **Exploration limits**	
MaxTransitions	**300**
⊟ **Exploration statistics**	
AcceptingStates	2
DeadStates	61
States	121
Transitions	239
UnsafeStates	4

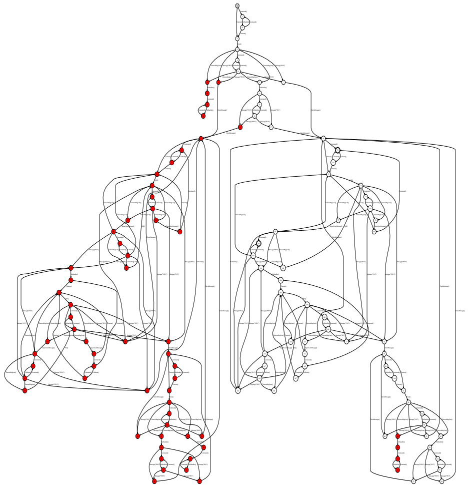
StateShape

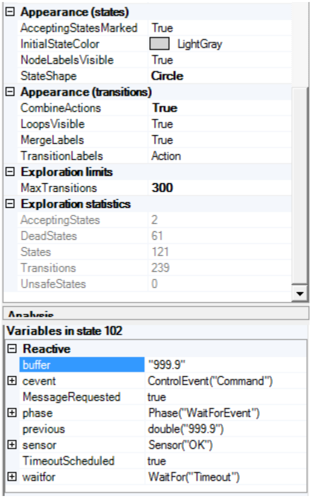
Variables in state 37

⊟ **Reactive**	
buffer	"999.9"
⊞ cevent	ControlEvent("Command")
MessageRequested	false
⊞ phase	Phase("HandleEvent")
previous	double("999.9")
⊞ sensor	Sensor("OK")
TimeoutScheduled	true
⊞ waitfor	WaitFor("Timeout")

Liveness Analysis

Exploration can search for *dead states* where there is no path to an accepting state.





⊟ **Appearance (states)**	
AcceptingStatesMarked	True
InitialStateColor	⬜ LightGray
NodeLabelsVisible	True
StateShape	**Circle**
⊟ **Appearance (transitions)**	
CombineActions	**True**
LoopsVisible	True
MergeLabels	True
TransitionLabels	Action
⊟ **Exploration limits**	
MaxTransitions	**300**
⊟ **Exploration statistics**	
AcceptingStates	2
DeadStates	61
States	121
Transitions	239
UnsafeStates	0

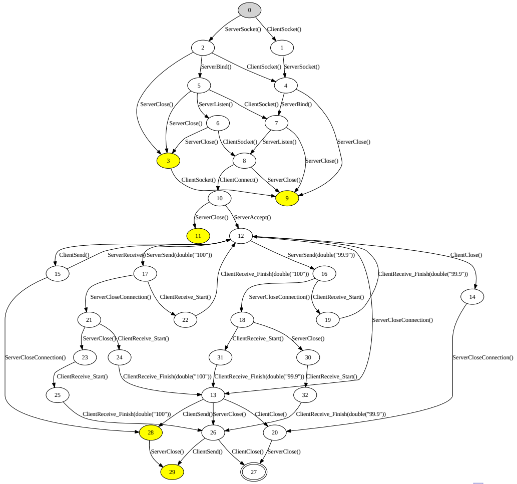
Analysis

Variables in state 102

⊟ **Reactive**	
buffer	"999.9"
⊞ cevent	ControlEvent("Command")
MessageRequested	true
⊞ phase	Phase("WaitForEvent")
previous	double("999.9")
⊞ sensor	Sensor("OK")
TimeoutScheduled	true
⊞ waitfor	WaitFor("Timeout")

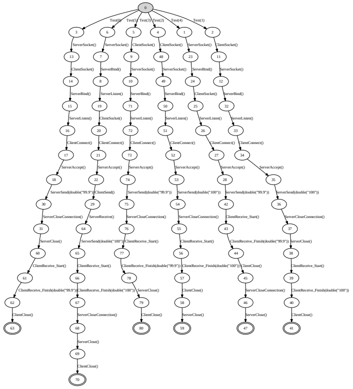
Offline Test Generation

Generate an FSM from a model program for a simple client/server using sockets. There are redundant paths due to different interleavings of startup and shutdown actions in client and server.



Offline Test Generation

Generate a test suite by traversing all the paths in the client/server FSM. Several similar test runs are needed to cover the redundant paths.



Test generation should include *scenario control* to exclude irrelevant cases.

Composition

Composition combines two or more programs to form a new program, the *product*.

$$M_1 \times M_2 = P$$

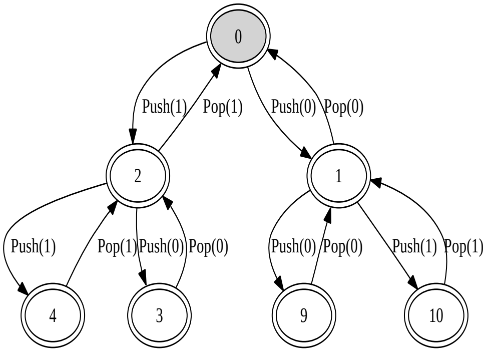
Usually we combine a *contract model program* in C# with a *scenario machine*, an FSM.

$$Contract \times Scenario = Product$$

Composition can be used for scenario control, validation, program structuring . . .

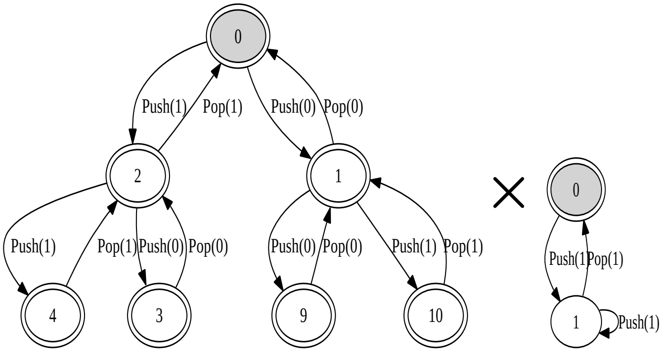
Composition

Composition synchronizes shared actions.



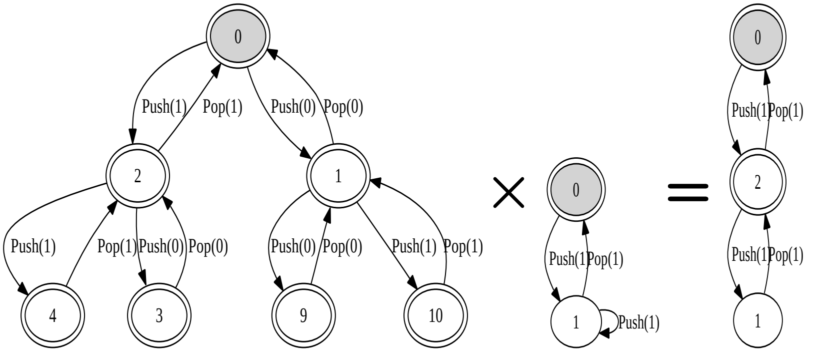
Composition

Composition synchronizes shared actions.



Composition

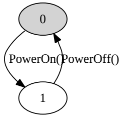
Composition synchronizes shared actions.



This usually has the effect of restricting behavior.

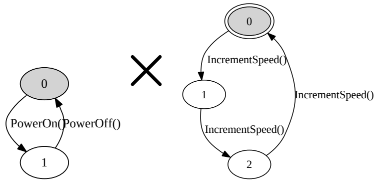
Composition

Composition interleaves unshared actions.



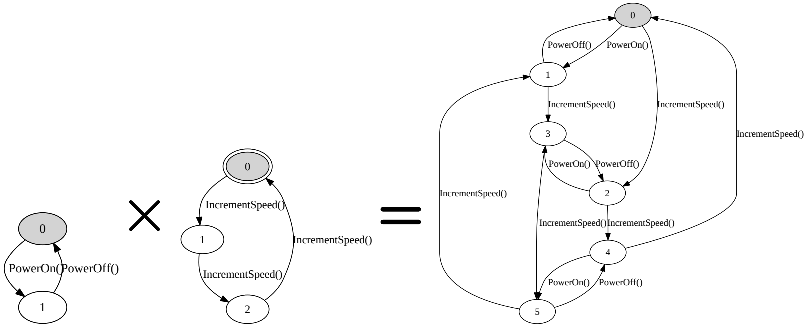
Composition

Composition interleaves unshared actions.



Composition

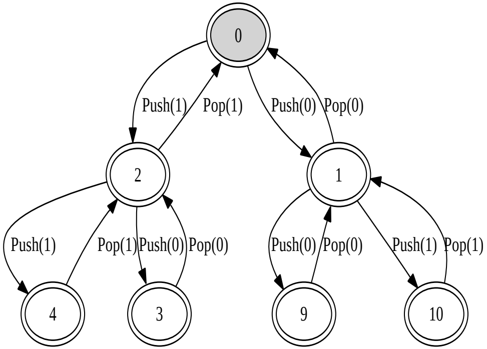
Composition interleaves unshared actions.



This usually has the effect of adding behavior.

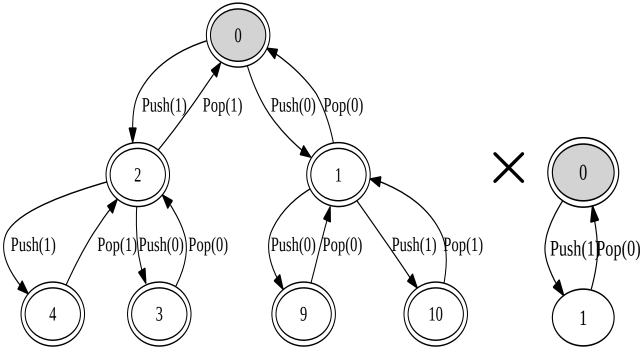
Validation

Composition with a scenario can help validate a model program.



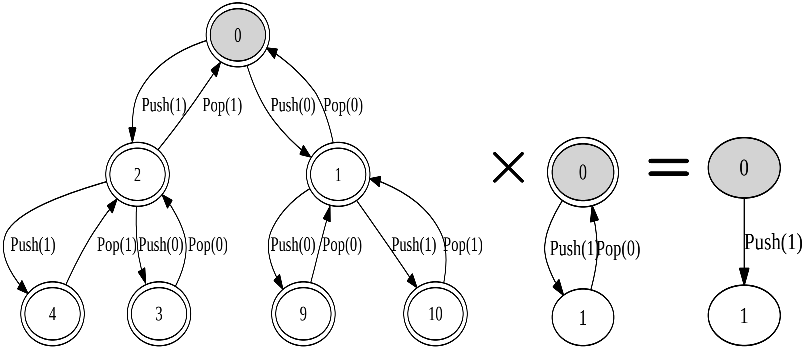
Validation

Composition with a scenario can help validate a model program.



Validation

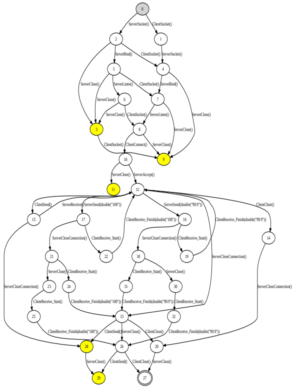
Composition with a scenario can help validate a model program.



The product shows whether the model program can execute the complete scenario. Does the product reach an accepting state?

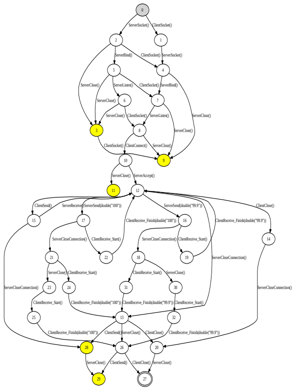
Scenario Control

Compose the contract model program with a scenario machine to eliminate redundant startup and shutdown paths.

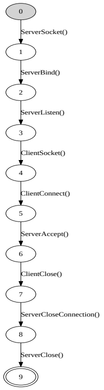


Scenario Control

Compose the contract model program with a scenario machine to eliminate redundant startup and shutdown paths.

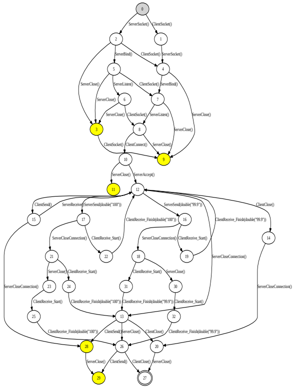


×

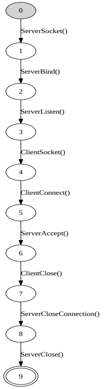


Scenario Control

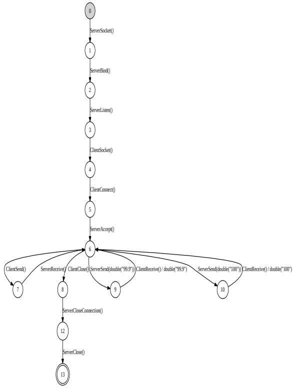
Compose the contract model program with a scenario machine to eliminate redundant startup and shutdown paths.



×



=



The product is a connected graph so all paths can be covered by a single test run.

Test Harness

Executing tests requires a harness (or adapter) to connect the model program to the implementation. In NModel a test harness is called a *stepper*.

```
// Implementation
Server s = new Server();
Client c = new Client();

public CompoundTerm DoAction(CompoundTerm action) {
    switch (action.Name) {
        // ...
      case("ServerSend"):
        s.Send((double)((Literal) action.Arguments[0]).Value);
        return null;

        // Actions that return values split into _Start and  _Finish
        case("ClientReceive_Start"):
        return CompoundTerm.Create("ClientReceive_Finish", c.Receive());
    }
}
```

Test Execution

The NModel test runner can execute a test script that was generated offline.

```
> ct /r:Stepper.dll /iut:ClientServerImpl.Stepper.Create /testSuite:Scenar
 ioTest.txt

TestResult(0, Verdict("Failure"),
"Action 'ClientReceive_Finish(double(\"99\"))' not enabled in the model",
 Unexpected return value of finish action, expected:
  ClientReceive_Finish(double "99.9"))
    Trace(
        Test(0),
        ServerSocket(),

        ... etc. ...

        ServerSend(double("100")),
        ClientReceive_Start(),
        ClientReceive_Finish(double("100")),
        ServerSend(double("99.9")),
        ClientReceive_Start(),
        ClientReceive_Finish(double("99"))
    )
```

On-the-fly Testing

On-the-fly testing overcomes some disadvantages of offline test generation.

- Generates test cases as the test run executes
- No FSM is generated, needn't finitize
- Handles nondeterminism economically
- Test runs can be indefinitely long, nonrepeating
- Can select among enabled actions randomly, or use an optional programmable strategy to increase coverage

Summary

Modeling and analysis

- A model can serve as a test case generator and oracle.
- Models are effective for testing behavior: ongoing activity that may exhibit history-dependence and nondeterminism.
- Represent behavior with traces: sequences of actions with arguments.
- Describe finite behaviors with FSMs, infinite behaviors with model programs.
- Exploration generates an FSM from a model program for visualization, safety and liveness analyses, or offline test generation.
- Composition combines programs by synchronizing shared actions and interleaving unshared actions.
- To validate a model program, compose it with FSMs that represent forbidden and allowed scenarios.

Summary

Testing

- Automated testing requires scenario control to limit the generated tests.
- To achieve scenario control, compose the contract model program with a scenario machine.
- The test harness translates between the abstract model program and the implementation.
- Offline test generation depends on exploration, with all its limitations.
- On-the-fly testing can generate indefinitely long, nonrepeating test runs for nondeterministic systems.
- On-the-fly testing can use a programmable strategy to increase coverage.

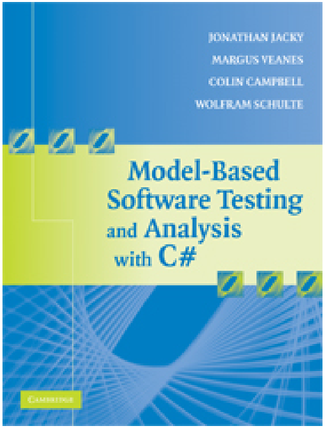
Observations

Industrial experience seems to be limited to *post hoc* modeling by test engineers, to test implementations that are already built.

Modeling and analysis in this style might help during earlier project stages, to assist in evaluating, improving, and documenting designs, and to plan for tests.

Resources

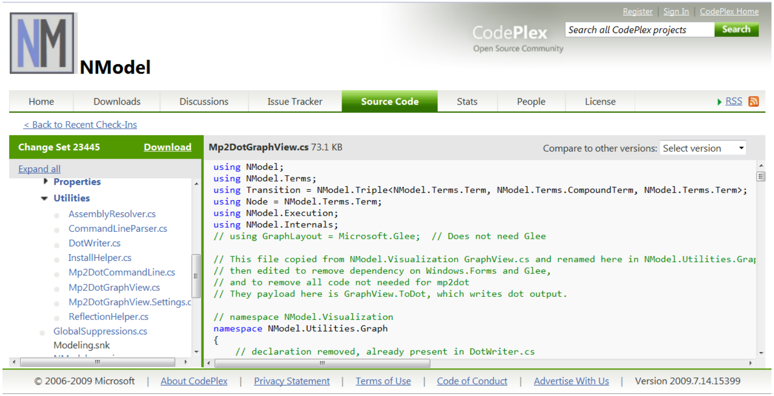
Model-Based Software Testing and Analysis with C#
Jonathan Jacky, Margus Veanes, Colin Campbell, Wolfram Schulte



Cambridge University Press, 2008

Resources

NModel is a model-based testing and analysis framework for C#.



```
using NModel;
using NModel.Terms;
using Transition = NModel.Triple<NModel.Terms.Term, NModel.Terms.CompoundTerm, NModel.Terms.Term>;
using Node = NModel.Terms.Term;
using NModel.Execution;
using NModel.Internals;
// using GraphLayout = Microsoft.Glee;  // Does not need Glee

// This file copied from NModel.Visualization GraphView.cs and renamed here in NModel.Utilities.Grap
// then edited to remove dependency on Windows.Forms and Glee,
// and to remove all code not needed for mp2dot
// They payload here is GraphView.ToDot, which writes dot output.

// namespace NModel.Visualization
namespace NModel.Utilities.Graph
{
   // declaration removed, already present in DotWriter.cs
```

© 2006-2009 Microsoft | About CodePlex | Privacy Statement | Terms of Use | Code of Conduct | Advertise With Us | Version 2009.7.14.15399

http://nmodel.codeplex.com/