Exhaustive testing:
A new tool based on Alloy

This is a slightly amended version of the talk slides given at ACCU Oxford on 2nd November 2017. All model excerpts are subject to the MIT license reproduced in the appendix. Slide notes are in italics; slide contents in normal font.

Contents

1 The challenge 3
2 Hello World 3
2.1 Introduction 3
2.2 A test case for the light switch 3
2.3 And the corresponding Alloy model 4
2.4 Mappings 5
2.5 Coverage 5
2.6 The Equivalent mapping in Ada 6
2.7 And the test case in Ada 6
3 Outline solution 7
3.1 Antecedents 7
3.1.1 Modelling languages 8
3.1.2 Design by contract 8
3.2 Antecedents (again) 9
3.3 Previous work in this area 9
3.4 Problems with existing tools 9
3.5 Why use Alloy? 9
3.6 Design goals 9
3.7 Outline solution revisited 10
3.8 Features 10
4 Command Line Parser 11
4.1 Introduction 11
4.2 The corresponding Alloy model 11
4.3 Testing a successful parse 13
4.4 Testing a failure with no arguments 14

4.5 Coverage 15
4.6 Equivalence classes for arguments 15
4.7 Command line parser – summary 16
5 Binary search 16
5.1 Introduction 16
5.2 The corresponding Alloy model 17
5.3 And a generated test 19
5.4 Coverage 19
6 Modelling constructors 20
6.1 Introduction 20
7 Future work 23
8 Appendix 23
8.1 License 23

1 The challenge

Given a sufficiently precise set of requirements:

- Generate an exhaustive test suite,
- In any imperative language

This shouldn't be too controversial. I've spent a lot of time thinking about different ways of expressing software designs, I've seen quite a few approaches used in practice. You'll have used a lot of them, natural language, UML, pseudocode. If you can model:

- *types and their invariants,*
- *operations and their pre/postconditions,*

Then you should be able to generate simulations that cover the full set of equivalence classes for your model.

2 Hello World

2.1 Introduction

A light switch can be modelled as:

- One bit of state,
- Two operations

This is (just about) the simplest example I could think of.
Let's work through from the test case to the Alloy model.

2.2 A test case for the light switch

One test case is all we need to cover both states.

```
using NUnit.Framework;
using ExampleApi.HelloWorld;

namespace ExampleTests.HelloWorld
{
  [TestFixture]
  public class TC001
  {
    [Test]
    public void TC001_0()
    {
      LightSwitch lightswitch_0 = new LightSwitch();

      lightswitch_0.TurnOn();

      Assert.AreEqual(SwitchState.On, lightswitch_0.SwitchState);

      lightswitch_0.TurnOff();

      Assert.AreEqual(SwitchState.Off, lightswitch_0.SwitchState);

    }
  }
}
```

2.3 And the corresponding Alloy model

To model this with Alloy we need types and operations.
Next, we need an initial state.
And then a simulation to generate the test case.

```
module models/HelloWorld

open models/ImperativeMachine2/Type
open models/ImperativeMachine2/Operation
open models/ImperativeMachine2/Value
open models/ImperativeMachine2/Reference

//=============================================================================================
//  Types
//=============================================================================================

abstract sig SwitchState extends Value {}  //Defines an enum with two values.
one sig On, Off extends SwitchState {}

sig LightSwitch extends Reference          //Defines a type with a single time-varying field
{                                          //field.
  switchState: SwitchState one -> Time
}

//=============================================================================================
//  Operations
//=============================================================================================

sig Noop extends Operation {}
{
  IsNoop[LightSwitch]
  instanceData.switchState.pre = instanceData.switchState.post
}

sig TurnOn extends Operation {}
{
  IsVoidCommand[LightSwitch]
  instanceData.switchState.pre  = Off
  instanceData.switchState.post = On
}

sig TurnOff extends Operation {}
{
  IsVoidCommand[LightSwitch]
  instanceData.switchState.pre  = On
  instanceData.switchState.post = Off
}

//=============================================================================================
//  Initial state
//=============================================================================================

fact
{
  all l: LightSwitch | l.switchState.first = Off
}

//=============================================================================================
// Simulations
//=============================================================================================

run TC001
//@ALLOY_SOLUTION_ITERATOR:EXPECTED_SOLUTION_COUNT=2;EXPECTED_DISTINCT_SOLUTION_COUNT=1
{
  #LightSwitch = 1
  #Operation = 2
```

```
}
for 1 LightSwitch, 2 Operation, 3 Time expect 1
```

2.4 Mappings

Mappings are used to convert the abstract (Alloy) model to something that makes sense in the target language. We need to define a template (StringTemplate is the library used for this), and we map through relevant things such as type names. The file is a standard java properties file (easy to parse).

It is a lot easier to push "implementation detail" to a mapping file – it simplifies the Alloy model.

```
# Code generation template
templateName = templates/CSharpTemplate.stg

# CSharp namespace definition:
namespaceName = ExampleTests

# Includes at the head of each file:
includes.1 = NUnit.Framework
includes.2 = ExampleApi.HelloWorld

# Names of accessors (<type>.<field>)
accessorNames.LightSwitch.switchState = SwitchState

# Names of assertions (<type>.<field>)
nullAssertNames.LightSwitch.switchState     = Assert.IsNull
equalityAssertNames.LightSwitch.switchState = Assert.AreEqual

# Type name aliases
typeNames.On  = SwitchState.On
typeNames.Off = SwitchState.Off
```

2.5 Coverage

All of the states are covered; Not hard.

c:\Data\SoftwareEngineering\FormalMethods\Alloy\Presentations\ACCU_Oct2017\SlideShow\examples\CSharp\ExampleApi\HelloWorld\LightSwitch.cs

#	Line	Line coverage
	1	using System;
	2	using System.Collections.Generic;
	3	using System.Linq;
	4	using System.Text;
	5	using System.Threading.Tasks;
	6	
	7	namespace ExampleApi.HelloWorld
	8	{
	9	public enum SwitchState
	10	{
	11	On,
	12	Off
	13	}
	14	
	15	public class LightSwitch
	16	{
5	17	public SwitchState SwitchState { get; private set; }
	18	
1	19	public LightSwitch()
	20	{
1	21	SwitchState = SwitchState.Off;
1	22	}
	23	
	24	public void TurnOn()
	25	{
1	26	SwitchState = SwitchState.On;
1	27	}
	28	
	29	public void TurnOff()
	30	{
1	31	SwitchState = SwitchState.Off;
1	32	}
	33	}
	34	}

2.6 The Equivalent mapping in Ada

This is the mapping to Ada:

```
# Code generation template
templateName = templates/AdaTemplate.stg

# Namespace definition - not used.

# The following packages are "withed" and "used" at the top of the specification:
includes.1 = Hello_World
includes.2 = Utils

# Names of accessors (<type>.<field>)
accessorNames.LightSwitch.switchState      = Get_Switch_State

# Names of assertions (<type>.<field>)
nullAssertNames.LightSwitch.switchState     = Assert_IsNull
equalityAssertNames.LightSwitch.switchState = Assert_Equal

# Type name aliases
typeNames.LightSwitch = Light_Switch
typeNames.SwitchState = Switch_State
```

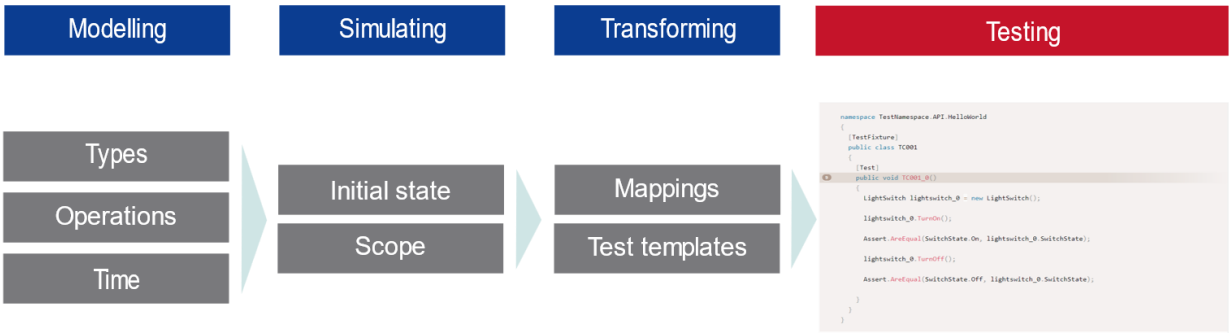
2.7 And the test case in Ada

This is simple to do with a good templating engine…

```
with AUnit.Test_Caller;
with Hello_World; use Hello_World;
with Utils; use Utils;

package body HelloWorld_TC001 is

   package Caller is new AUnit.Test_Caller (HelloWorld_TC001.Test_Fixture_Type);

   function Suite return Access_Test_Suite is
      Return_Value : constant Access_Test_Suite := new Test_Suite;
   begin
      Return_Value.Add_Test (Caller.Create ("TC001_0", HelloWorld_TC001_0'Access));

      return Return_Value;
   end Suite;

   procedure HelloWorld_TC001_0 (T : in out Test_Fixture_Type) is

      lightswitch_0 : Light_Switch;

   begin

      lightswitch_0.TurnOn;

      Assert_Equal (On, lightswitch_0.Get_Switch_State);

      lightswitch_0.TurnOff;

      Assert_Equal (Off, lightswitch_0.Get_Switch_State);

   end HelloWorld_TC001_0;


end HelloWorld_TC001;
```

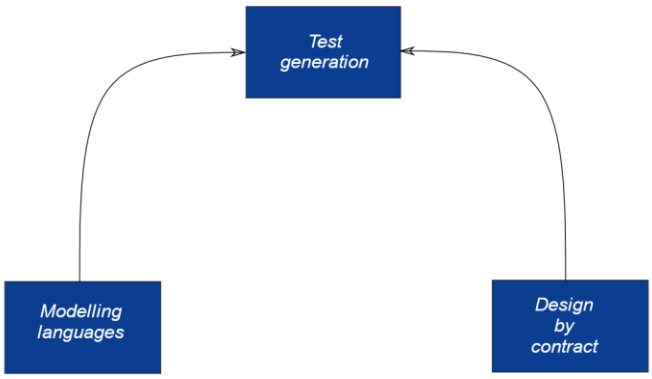
3 Outline solution

This describes the basic solution – model in Alloy, simulate in Alloy, use templates + mappings to transform to executable code in the target language.



3.1 Antecedents

Test generation is feasible because of two things, modelling languages and design by contract.



3.1.1 Modelling languages

There are a lot of different modelling languages out there, each with their own characteristics. The important thing is that they allow us to precisely describe certain facets of the design of a piece of software. You don't have to describe everything, just the facets you are interested in. Some of these languages have been around for a long time. CSP and Z were both first described in 1977. More modern languages include sophisticated toolchains to allow simulations to be executed, or proof conditions to be checked.

Languages such as Alloy, Z, CSP give you...

- Precise descriptions of:
  - Structure
  - Behaviour
- Tooling to support:
  - Type checking
  - Visualisation
  - Proof checking

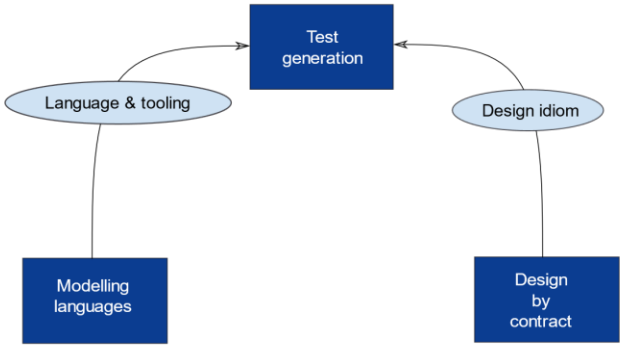
3.1.2 Design by contract

Modelling languages give you a notation to describe things like structure and behaviour. But you also need an idiom that you can use to structure your ideas. DbC is a software correctness methodology that uses preconditions, postconditions on operations, and invariants on types to describe software behaviour. Tony Hoare originally described the use of preconditions and postconditions in 1969. DbC was developed by Bertrand Meyer (and is trademarked by him). Not enough languages support it natively (Eiffel, D, Ada). However there are various libraries or extensions that to a reasonable job of providing these facilities in other languages.

The most important idea you've never heard of....

- Trademarked by Bertrand Meyer
- Native support in:
  - Eiffel
  - Ada
  - D
- Tooling to allow use in:
  - Java (Cofoja, JML)
  - C# (Code Contracts for .Net)
  - C/C++ (Perfect Developer)

3.2 Antecedents (again)



3.3 Previous work in this area

IntelliTest (in VS) analyses paths through your code & tries to generate an exhaustive test suite. Weakness? It documents what your code does already. JMLUnit generates test cases based on DBC-style tests - good. TestEra - generates test cases for Java based on DBC-style specifications, and uses Alloy under the hood.

- Microsoft Digger / Pex (IntelliTest)
- JML/JMLUnit
- TestEra
- QuickCheck

3.4 Problems with existing tools

- Microsoft Digger / Pex (IntelliTest)
  - Tests document existing behaviour
- JML/JMLUnit
  - Annotations on Java code
- TestEra
  - Annotations on Java code
- QuickCheck
  - Haskell

3.5 Why use Alloy?

- Language supports many modelling idioms
- Analyser allows whole of state space to be checked
- Mature product - over 10 years old
- Mature API
- Already been used for test generators

3.6 Design goals

- Use "standard" Alloy

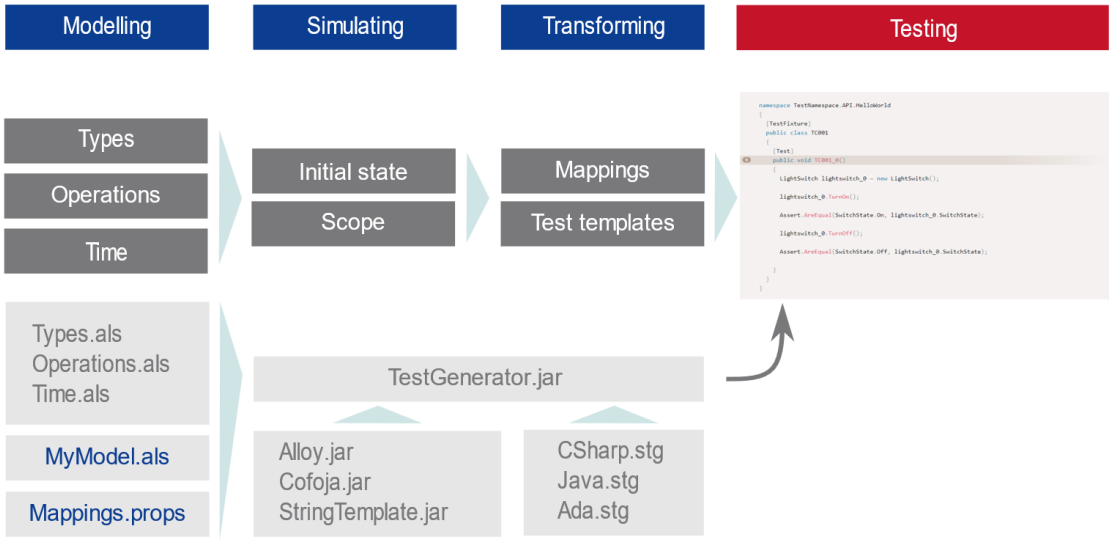
  Pros - Simpler tool maintenance

Cons - Models more verbose

- Models exist separately from target code
  Pros - Use at any point in the development cycle
  Cons - Models more difficult to build?
- Capable of targeting multiple languages
  Pros - Flexibility
- Simple to develop

3.7 Outline solution revisited

MyModel.als and Mappings.props – what you configure; .stg files – language-specific templates. TestGenerator.jar – the test generator application; Cofoja.jar – Java contracts library; StringTemplate.jar – Java string templates library.



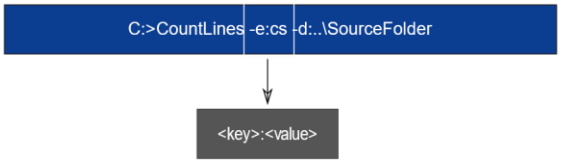
3.8 Features

- Modelling concepts:
  - Types
  - Operations
  - Time
- Generation of assertions:
  - Postconditions
  - Return values
- Simplifications:
  - Single-threaded models
  - No handling of exceptions

4 Command Line Parser

4.1 Introduction

A slightly more complex example… parsing command line arguments. Two valid keys, -e and -d; both are required.



4.2 The corresponding Alloy model

```
module models/CommandLineParser

open models/ImperativeMachine2/Type
open models/ImperativeMachine2/Operation
open models/ImperativeMachine2/Reference
open models/ImperativeMachine2/List
open models/ImperativeMachine2/Value


//=============================================================================================
//  The parse result:
//=============================================================================================

abstract sig ParseResult extends Value{}
one sig Success, Failure extends ParseResult{}

//=============================================================================================
//  The array of values passed to the parser
//=============================================================================================

one sig ArgumentList extends List{}
{
  data.elems = Argument
}

//=============================================================================================
//  An argument passed to the parser (<key><delim><value>)
//=============================================================================================

abstract sig Argument extends Reference{}
sig DirectoryArgument, ExtensionArgument, BadArgument extends Argument{}
```

```
//=============================================================================================
//  The converted argument.
//=============================================================================================

abstract sig ConvertedArgument extends Reference{}
one sig DirectoryArgumentConverted, ExtensionArgumentConverted extends ConvertedArgument{}

//=============================================================================================
//  The argpack type, has three fields to store converted values.
//=============================================================================================

one sig Argpack extends Reference
{
  directory: DirectoryArgumentConverted -> Time,
  extension: ExtensionArgumentConverted -> Time
}


//=============================================================================================
//  The parse operation takes a single ArgumentList as parameter, returns ParseResult:
//=============================================================================================

sig Parse extends Operation {
  arguments: one ArgumentList
}{
  IsCommandWithReturn[Argpack, ParseResult]

  ArgumentsAreValid[arguments] => {
    returns = Success

    one instanceData.directory.post :> DirectoryArgumentConverted
    one instanceData.extension.post :> ExtensionArgumentConverted
  }
  else {
    returns = Failure

    no instanceData.directory.post
    no instanceData.extension.post
  }
}

//=============================================================================================
//  Each of DirectoryArgument, ExtensionArgument must appear exactly once.
//  Order in list is unimportant.
//=============================================================================================

pred ArgumentsAreValid[arguments: ArgumentList]
{
  one arguments.data.DirectoryArgument
  one arguments.data.ExtensionArgument
}

//=============================================================================================
//  Initialisation
//=============================================================================================

fact
{
  all a: Argpack| one t: first | Init[a, t]
}

pred Init[a: Argpack, t: Time]
{
  no a.directory.t
  no a.extension.t

}

//=============================================================================================
```

```
//  Simulations - successful parse:
//=============================================================================================

run TC001_Success
//@ALLOY_SOLUTION_ITERATOR:EXPECTED_SOLUTION_COUNT=2;EXPECTED_DISTINCT_SOLUTION_COUNT=2
{
  one o: Operation | o.returns = Success
}
for 1 Argpack, 2 Argument, 1 Operation, 2 Time

//=============================================================================================
//  Simulations - failed parse:
//=============================================================================================

run TC020_NoArgument_Failure
//@ALLOY_SOLUTION_ITERATOR:EXPECTED_SOLUTION_COUNT=1;EXPECTED_DISTINCT_SOLUTION_COUNT=1
{
  one o: Operation | o.returns = Failure
  #ArgumentList.data = 0
}
for 1 Argpack, 2 Argument, 1 Operation, 2 Time

run TC021_OneArgument_Failure
//@ALLOY_SOLUTION_ITERATOR:EXPECTED_SOLUTION_COUNT=3;EXPECTED_DISTINCT_SOLUTION_COUNT=3
{
  one o: Operation | o.returns = Failure
  #ArgumentList.data = 1
  #Argument = 1
}
for 1 Argpack, 1 Argument, 1 Operation, 2 Time

run TC022_TwoArgument_Failure
//@ALLOY_SOLUTION_ITERATOR:EXPECTED_SOLUTION_COUNT=7;EXPECTED_DISTINCT_SOLUTION_COUNT=7
{
  one o: Operation | o.returns = Failure
  #ArgumentList.data = 2
  #Argument = 2
}
for 1 Argpack, 2 Argument, 1 Operation, 2 Time

run TC023_ThreeArgument_Failure
//@ALLOY_SOLUTION_ITERATOR:EXPECTED_SOLUTION_COUNT=21;EXPECTED_DISTINCT_SOLUTION_COUNT=21
{
  one o: Operation | o.returns = Failure
  #ArgumentList.data = 3
  #Argument = 3
}
for 1 Argpack, 3 Argument, 1 Operation, 2 Time
```

4.3 Testing a successful parse

This is the set of test cases corresponding to TCC_001_Success in 4.2:

```
using System;
using System.Collections.Generic;
using NUnit.Framework;
using ExampleApi.CommandLineParser;

namespace ExampleTests.CommandLineParser
{
  [TestFixture]
  public class TC001_Success
  {
    [Test]
    public void TC001_Success_0()
    {
      Argpack argpack_0 = new Argpack();
```

```
      IList<String> argumentlist_0 = Factory.CreateArgumentList();
      String directoryargument_0 = Factory.CreateDirectoryArg();
      argumentlist_0.Add(directoryargument_0);
      String extensionargument_0 = Factory.CreateExtensionArg();
      argumentlist_0.Add(extensionargument_0);
      String directoryargumentconverted_0_time_1 = Factory.CreateDirectoryArgConverted();
      String extensionargumentconverted_0_time_1 = Factory.CreateExtensionArgConverted();

      Assert.AreEqual(ParseResult.Success, argpack_0.Parse(argumentlist_0));

      Assert.AreEqual(directoryargumentconverted_0_time_1, argpack_0.Directory);
      Assert.AreEqual(extensionargumentconverted_0_time_1, argpack_0.Extension);

    }
    [Test]
    public void TC001_Success_1()
    {
      Argpack argpack_0 = new Argpack();
      IList<String> argumentlist_0 = Factory.CreateArgumentList();
      String extensionargument_0 = Factory.CreateExtensionArg();
      argumentlist_0.Add(extensionargument_0);
      String directoryargument_0 = Factory.CreateDirectoryArg();
      argumentlist_0.Add(directoryargument_0);
      String directoryargumentconverted_0_time_1 = Factory.CreateDirectoryArgConverted();
      String extensionargumentconverted_0_time_1 = Factory.CreateExtensionArgConverted();

      Assert.AreEqual(ParseResult.Success, argpack_0.Parse(argumentlist_0));

      Assert.AreEqual(directoryargumentconverted_0_time_1, argpack_0.Directory);
      Assert.AreEqual(extensionargumentconverted_0_time_1, argpack_0.Extension);

    }
  }
}
```

4.4 Testing a failure with no arguments

And here's the "no argument" failure:

```
using System;
using System.Collections.Generic;
using NUnit.Framework;
using ExampleApi.CommandLineParser;

namespace ExampleTests.CommandLineParser
{
  [TestFixture]
  public class TC020_NoArgument_Failure
  {
    [Test]
    public void TC020_NoArgument_Failure_0()
    {
      Argpack argpack_0 = new Argpack();
      IList<String> argumentlist_0 = Factory.CreateArgumentList();

      Assert.AreEqual(ParseResult.Failure, argpack_0.Parse(argumentlist_0));

      Assert.IsNull(argpack_0.Directory);
      Assert.IsNull(argpack_0.Extension);

    }
  }
}
```

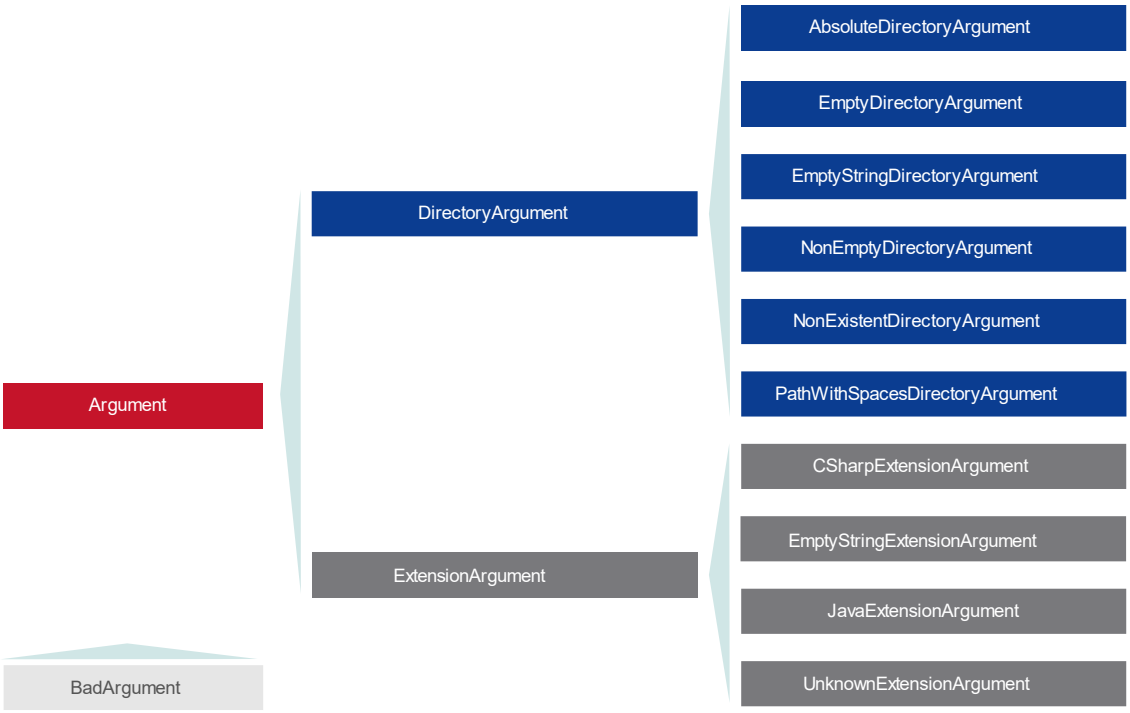
4.5 Coverage

Not all states covered; Sometimes not all equivalence classes covered by the model, sometimes need to inspect CIL to identify why the gaps are there (compiler-generated code)

```
 6  namespace ExampleApi.CommandLineParser
 7  {
 8      public class Argpack
 9      {
10          private String directory_;
11          private String extension_;
12
34 13          public String Directory { get { return directory_; } }
34 14          public String Extension { get { return extension_; } }
15
16          public ParseResult Parse(IList<String> arguments)
17          {
18              //Error handling:
59 19              if (arguments.Count()           != 2) return ParseResult.Failure;
12 20              if (arguments.Distinct().Count() != 2) return ParseResult.Failure;
21
28 22              foreach (String argument in arguments)
23              {
10 24                  String[] parameter = argument.Trim().Split(':');
25
10 26                  if(parameter.Count() != 2)
27                  {
 4 28                      HandleParseFailure();
 4 29                      return ParseResult.Failure;
30                  }
31
 6 32                  Contract.Assert(parameter.Count() == 2);
33
 6 34                  String key   = parameter[0];
 6 35                  String value = parameter[1];
36
 6 37                  ParseResult result = ParseResult.Failure;
38
 6 39                  switch (key)
40                  {
 6 41                      case "-d": result = Convert(value, out directory_); break;
 6 42                      case "-e": result = Convert(value, out extension_); break;
 0 43                      default:    result = ParseResult.Failure;     break;
44                  }
45
 6 46                  if(result == ParseResult.Failure)
47                  {
 0 48                      HandleParseFailure();
 0 49                      return result;
50                  }
51              }
52
 2 53              return ParseResult.Success;
 4 54          }
55
56          //Make sure output values are null & return parse result:
57          private void HandleParseFailure()
58          {
 4 59              directory_ = null;
 4 60              extension_ = null;
 4 61          }
62
63          //Stub for a conversion function:
64          private ParseResult Convert(String value, out String result )
65          {
 6 66              result = value;
 6 67              return ParseResult.Success;
68          }
69      }
```

4.6 Equivalence classes for arguments

Can use Alloy's type system to build classification hierarchies that model the problem domain very precisely…



4.7 Command line parser – summary

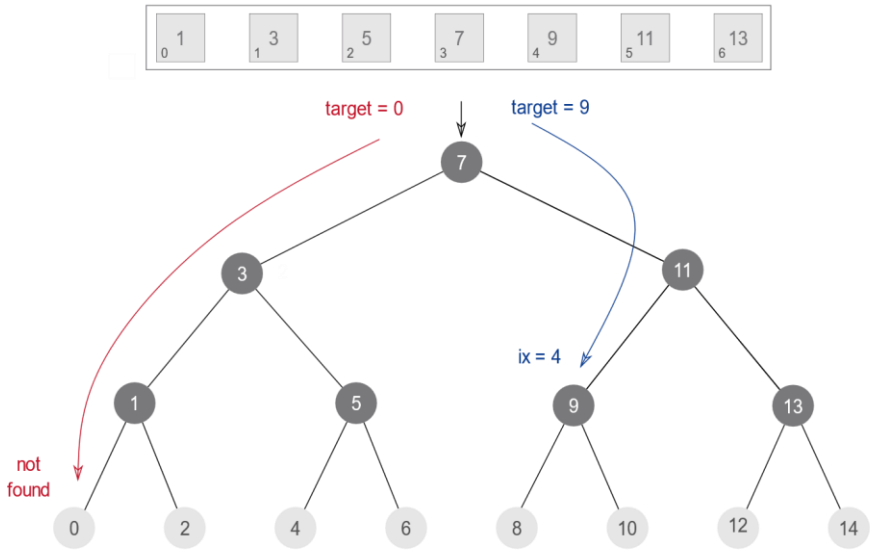
One of the most interesting things about doing this type of modeling is that it forces you to be really precise…

- Clarity - Explicit modelling of types, operations, expected results
- Conciseness - Compact model represents lots of behavior, without specifying implementation details
- Completeness - All states within model covered automatically

5 Binary search

5.1 Introduction

Another example… binary search



5.2 The corresponding Alloy model

The slightly odd design of orderings relations was because of a quirk with the Alloy solver – there is a better way…

```
module models/BinarySearch

open models/ImperativeMachine2/Type
open models/ImperativeMachine2/Operation
open models/ImperativeMachine2/Value
open models/ImperativeMachine2/Reference
open models/ImperativeMachine2/Boolean

//=============================================================================================
//  Primitives to represent keys and values, together with orderings:
//=============================================================================================

abstract sig K extends Value {}
abstract sig V extends Value {}

one sig K0, K1, K2 extends K {}
one sig V0, V1, V2, V3, V4, V5, V6 extends V {}

one sig Orderings
{
  AllKeys:   seq K,
  AllValues: seq V
}
{
  AllKeys.elems = K
  !AllKeys.hasDups

  AllValues.elems = V
  !AllValues.hasDups

  //Defines ordering of elements of keys
  AllKeys[0] = K0
  AllKeys[1] = K1
  AllKeys[2] = K2

  //Defines ordering of elements of values
  AllValues[0] = V0
  AllValues[1] = V1
  AllValues[2] = V2
  AllValues[3] = V3
```

```
  AllValues[4] = V4
  AllValues[5] = V5
  AllValues[6] = V6
}

//=============================================================================================
//  Types
//=============================================================================================

one sig ReturnValue extends Value
{
  success: one Boolean,
  kFound: lone K
}

one sig Table extends Reference
{
  keys : seq K,
  vals : seq V
}
{
  //All slots used:
  #keys = #vals

  //All keys in:
  keys = Orderings.AllKeys

  //Maps K0 -> V1, K1 -> V3, K2 -> V5:
  all ix: keys.inds | vals[ix] = Orderings.AllValues[plus[mul[ix,2],1]]
}

//=============================================================================================
//  Operations
//=============================================================================================

sig GetKey extends Operation
{
  v: one V
}
{
  IsCommandWithNullableReturn[Table, ReturnValue]

  instanceData.KeyExists[v] =>
  {
    returns.success = True
    returns.kFound = instanceData.GetKey[v]
  }
  else
  {
    returns.success = False
    no returns.kFound
  }
}

pred KeyExists[t: Table, v: V]
{
          some ix: t.vals.inds | t.vals[ix] = v
}

fun GetKey[t: Table, v: V] : one K
{
  let ix = t.vals.idxOf[v] | t.keys[ix]
}

//=============================================================================================
//  Simulations
//=============================================================================================

run TC001
```

```
//@ALLOY_SOLUTION_ITERATOR:EXPECTED_SOLUTION_COUNT=4;EXPECTED_DISTINCT_SOLUTION_COUNT=4
{
  ReturnValue.success = False
}
for 7 seq, 1 Operation, 2 Time,  exactly 7 V, exactly 3 K expect 1

run TC002
//@ALLOY_SOLUTION_ITERATOR:EXPECTED_SOLUTION_COUNT=3;EXPECTED_DISTINCT_SOLUTION_COUNT=3
{
  ReturnValue.success = True
}
for 7 seq, 1 Operation, 2 Time, exactly 7 V, exactly 3 K expect 1
```

5.3 And a generated test

```
using NUnit.Framework;
using System;
using ExampleApi.BinarySearch;
using ExampleTests.TestHelpers;

namespace ExampleTests.BinarySearch
{
  [TestFixture]
  public class TC002
  {
    [Test]
    public void TC002_0()
    {
      Table table_0 = BinarySearchHelper.CreateTable();

      ReturnValue returnvalue_0 = table_0.GetKey(1);
      Assert.AreEqual(true, returnvalue_0.Success);
      Assert.AreEqual(0, returnvalue_0.KFound);


    }
    [Test]
    public void TC002_1()
    {
      Table table_0 = BinarySearchHelper.CreateTable();

      ReturnValue returnvalue_0 = table_0.GetKey(3);
      Assert.AreEqual(true, returnvalue_0.Success);
      Assert.AreEqual(1, returnvalue_0.KFound);


    }
    [Test]
    public void TC002_2()
    {
      Table table_0 = BinarySearchHelper.CreateTable();

      ReturnValue returnvalue_0 = table_0.GetKey(5);
      Assert.AreEqual(true, returnvalue_0.Success);
      Assert.AreEqual(2, returnvalue_0.KFound);


    }
  }
}
```

5.4 Coverage

Spot the deliberate error in the implementation…

```
 9      public class Table
10      {
11          IList<Int32> values_ = new List<Int32>();
12
13          public Table(IList<Int32> values)
14          {
15              values_ = values;
16          }
17
18          public ReturnValue GetKey(int v)
19          {
20              Int32 upper = values_.Count - 1;
21              Int32 lower = 0;
22
23              while (upper >= lower)
24              {
25                  int ix = (lower + upper) / 2;
26
27                  int v_ix = values_[ix];
28
29                  if (v < v_ix) //guess is above target
30                  {
31                      upper = ix - 1;
32                  }
33                  else if (v > v_ix) //guess is below target
34                  {
35                      lower = ix + 1;
36                  }
37                  else //guess is same as target
38                  {
39                      return new ReturnValue (true, ix);
40                  }
41              }
42
43              return new ReturnValue (false, null);
44          }
45      }
```

6 Modelling constructors

6.1 Introduction

Need to be able to model constructors explicitly… slightly different approach to structuring some aspects of the model…

```
module models/BinarySearchWithConstructor

open models/ImperativeMachine2/Type
open models/ImperativeMachine2/Operation
open models/ImperativeMachine2/Value
open models/ImperativeMachine2/Boolean
open models/ImperativeMachine2/List
open models/ImperativeMachine2/Reference

//=============================================================================================
//  Primitives to represent an ordered set of values:
//=============================================================================================
abstract sig K extends Value {}
abstract sig V extends Value {}

one sig K0, K1, K2 extends K {}
one sig V0, V1, V2, V3, V4, V5, V6 extends V {}

one sig Orderings
{
  AllValues: seq V,
  AllKeys:   seq K
```

```
}
{
  AllKeys =
    0 -> K0 +
    1 -> K1 +
    2 -> K2

  AllValues =
    0 -> V0 +
    1 -> V1 +
    2 -> V2 +
    3 -> V3 +
    4 -> V4 +
    5 -> V5 +
    6 -> V6
}

//=============================================================================================
//  Type returned by GetKey.
//=============================================================================================

one sig ReturnValue extends Value
{
  success: one Boolean,
  kFound: lone K
}

//=============================================================================================
//  Represents an array of data we can use to initialise the lookup table via its ctr.
//=============================================================================================

abstract sig M extends Value
{
  k: one K,
  v: one V
}

one sig M0 extends M{} { k = K0 && v = V1 }
one sig M1 extends M{} { k = K1 && v = V3 }
one sig M2 extends M{} { k = K2 && v = V5 }

one sig TableData extends List{}
{
  data =
    0 -> M0 +
    1 -> M1 +
    2 -> M2
}

//=============================================================================================
//  The lookup table, containing a sequence of values.
//=============================================================================================

one sig Table extends Reference
{
  mappings: seq M
}

//=============================================================================================
//  Constructor for Table. Takes an instance of TableData as a parameter.
//=============================================================================================

sig Constructor extends Operation
{
  tableData: one TableData
}
{
  IsConstructor[Table]
  instanceData.mappings = tableData.data
```

```
}

//=============================================================================================
//  GetKey operation takes a single V as parameter, returns a ReturnValue object.
//=============================================================================================

sig GetKey extends Operation
{
  v: one V
}
{
  IsCommandWithReturn[Table, ReturnValue]

  instanceData.KeyExists[v] =>
  {
    returns.success = True
    returns.kFound = instanceData.GetKey[v]
  }
  else
  {
    returns.success = False
    no returns.kFound
  }

}

//=============================================================================================
//  KeyExists predicate is true if v exists in table t.
//=============================================================================================
pred KeyExists[t: Table, vToFind: V]
{
  some m: t.mappings.elems | m.v = vToFind
}

//=============================================================================================
//  GetKey function returns the index of v in table t.
//=============================================================================================
fun GetKey[t: Table, vToFind: V] : K
{
  let mapping = {m: t.mappings.elems | m.v = vToFind} |
  mapping.k
}

//=============================================================================================
//  Apply constraints to give correct behaviour with constructors
//=============================================================================================
fact
{
    UseConstructors
}

//=============================================================================================
//  Simulations
//=============================================================================================

run TC001
//@ALLOY_SOLUTION_ITERATOR:EXPECTED_SOLUTION_COUNT=8;EXPECTED_DISTINCT_SOLUTION_COUNT=4
{
  ReturnValue.success = False
}
for 7 seq, 2 Operation, 3 Time expect 1

run TC002
//@ALLOY_SOLUTION_ITERATOR:EXPECTED_SOLUTION_COUNT=6;EXPECTED_DISTINCT_SOLUTION_COUNT=3
{
  ReturnValue.success = True
}
for 7 seq, 2 Operation, 3 Time expect 1
```

7 Future work

Limitation is the number of (spare) hours I've got… Building version 1 has taught me a lot, now need to focus on simplifying what I've got, and making it easier to use…

- V&V
- Documentation
- Model validator utility
- Command generator utility
- Testing of exception contracts
- Testing of performance constraints

8 Appendix

8.1 License