Changelist:

2015-02-04 – V3:

- Moved some redundant slides to end
- Worked through notes for first part of presentation.

2015-02-05 – V4:

- Checked source for models in first part – added in filenames for each model.

2015-02-08 – V5:

- Source code for second part models

2015-02-09 – V6:

- Figure and ground slides.

A question:

What tools do you use to describe requirements and designs?

We are going to spend some time this evening looking at a modelling language and tool called Alloy.

I've been exploring it for the past six months or so, and I think it's potentially very valuable as a tool for expressing requirements and designs in software.

During this talk I'm going to:

- show how it can be used,
- describe what I've learned by doing it,
- provide some examples of how it is used in the "real world".
- point you to some resources that will help you get started.

To get us thinking about the right sort of issues, let's start with a question...

Imagine you are working on a small feature (for some definition of small)...

What tools (if any) would you use to describe the requirements?
What tools (if any) would you use to describe the design?

What do these tools help you to achieve?
Where do they fall down?

Some possibilities...

- Natural language
- Diagrams – boxes and lines
- UML
- Pseudocode
- Cucumber

There are many options for expressing requirements and designs...
These are not mutually exclusive...
They have different dimensions of formality – some have very precisely defined syntax/semantics, others less so...

If you only know one tool, that is all you've got...

Average cost of defect fix

		Time Detected				
		Requirements	Architecture	Construction	System Test	Post-Release
Time Introduced	Requirements	1	3	5-10	10	10-100
	Architecture		1	10	15	25-100
	Construction			1	10	10-25

Data from Steve McConnell, Code Complete 2nd ed, p29.

Why do we use tools like this?

Usually because we want software that solves customer problems at a reasonable price.

Mistakes in requirements and design can be very expensive

They can make ongoing maintenance and development of your system much more expensive

If you identify mistakes late in your project, you may not have time to go back and rework it fully - you may just have to live with a hacked-in fix.

The data from Steve McConnell is very revealing – I think the numbers are scary.

Many mistakes stem from...

- Incompleteness
- Ambiguity
- Inconsistency

Leading to unforseen,undesirable consequences.

Where do mistakes come from?

People build consulting and writing careers on answering this question...

I'm going to do it in one slide...

Many mistakes stem from incompleteness, ambiguity and inconsistency.

Exposing these early on helps us build a common understanding of what our software is trying to do, and how it is trying to do it.

One solution is to increase formalism early on in the project.

- "Formalism" exists on a continuum
- Formalise where it is helpful
- Use it to reduce ambiguity, inconsistency etc.

It doesn't need to involve lots of paper + maths.

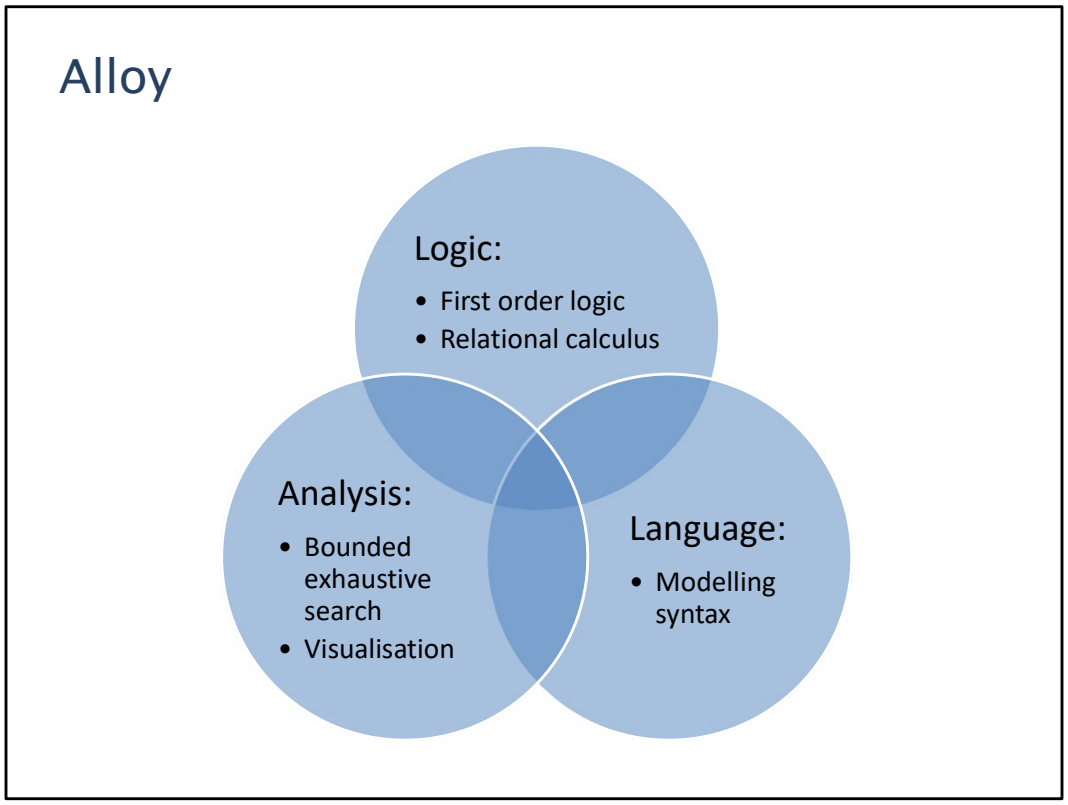
So it would seem logical that if we want to reduce or eliminate mistakes from these sources, we should try to find a way to reduce incompleteness, ambiguity, inconsistency.

This isn't an all or nothing decision...

You can increase the formalism where it is useful to do so,

And not where it isn't...

Make the decision on a case-by-case basis...



This gives the background to why I think Alloy is a valuable tool.

It provides a very efficient mechanism for expressing and exploring both structure and behaviour in software designs.

The language and tool comprises three facets - logic, language and analysis.

Logic:
- Everything is a relation
- Structure and behaviour are expressed using logical constraints

Language:
- ASCII only!
- Allows you to structure models
- Simple module system (generics)
- Basic polymorphism

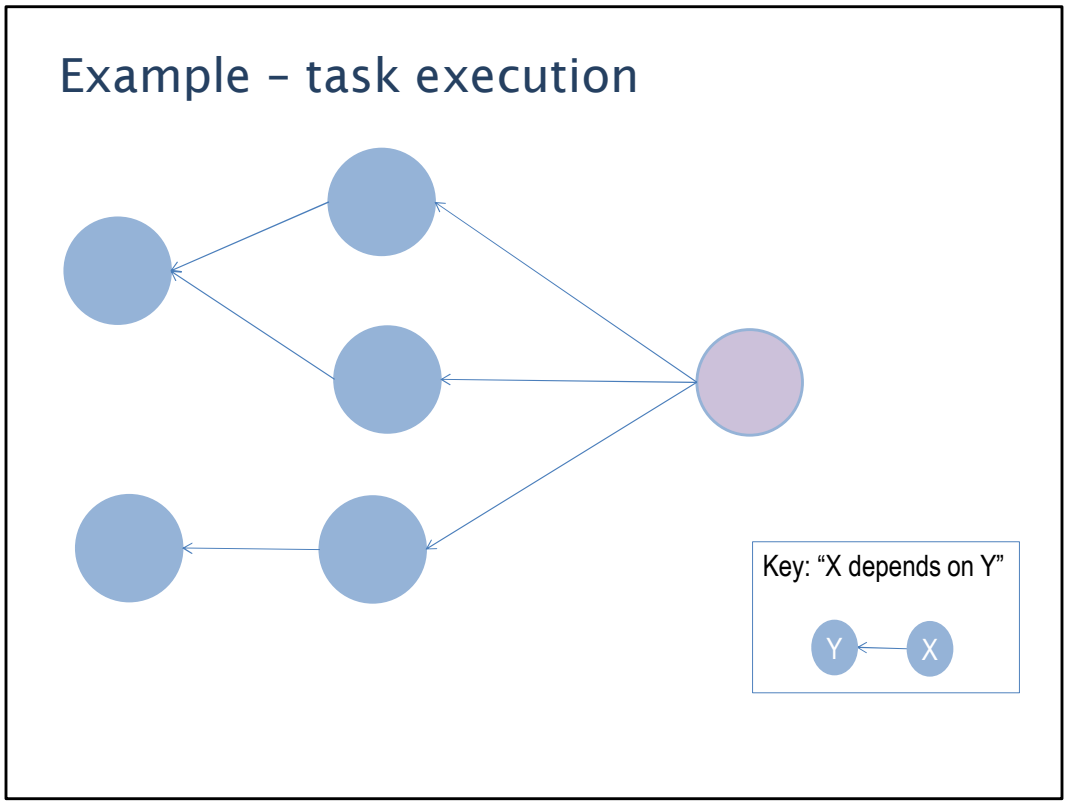
Analysis:
- Control of scope
- Simulation and checking
- Bounded exhaustive search of all examples / counterexamples within a given scope
- Automatic visualisation of examples / counterexamples

Development of Alloy

- Alloy's ancestors are Z and SMV:
  - Z – sets and relations
  - SMV – automated analysis
- Alloy was developed by Daniel Jackson of the Software Design Group at MIT.
- The language and tool is still actively developed.

Z – from Oxford

SMZ – Model checking for formally verifying finite-state concurrent systems (CMU).



On to our example....

Imagine we are creating an application to support execution of an arbitrary network of tasks.
For example, a build system or task scheduler of some sort.

We do a little bit of work at a whiteboard and come up with something like this picture.
We know the graph of tasks must be acyclic...
And that they might be a little more complex than just a simple tree of tasks.

What might a task look like in code?

```
class Task {
  private:
    Task [] predecessors_;

  public:
    Task [] getPredecessors()      { return predecessors_; }

    void addPredecessor(Task pred) { predecessors_~= pred; };
};
```

In a high-level language, this is how we might represent the nodes of a graph...

So far so good...

But we haven't got any mechanism for testing this abstraction – we'd have to write some unit tests to validate how it behaves.

Now the same abstraction in Alloy

```
sig Task {
  predecessors: set Task
}
```

Each type is a sig (signature). They define sets of atoms.
We define a relation, predecessors, that maps each Task to a set (possibly empty) of Tasks.

Now the same abstraction in Alloy

```
sig Task {
  predecessors: set Task
}

pred Show {
  some predecessors
}

run Show for 4
```

module Alloy/Models/ACCU_2015/Tasks_V1

We actually need a little more to make a working alloy model.

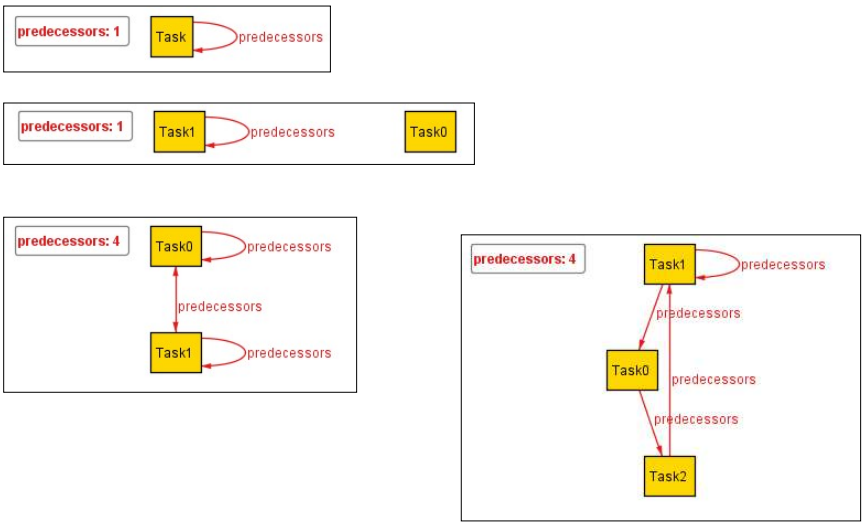
As well as the signature, we define a predicate, and a command which will find an instance of the predicate.

The run command is used to find instances of our model.

We've bounded the search to have at most four atoms

And we've said that we want to have a non-zero number of predecessors.

And when we execute (run)....



- Here is what you get...
- Some example graphs of tasks...
- Bounded search => 4 nodes at maximum.
- note we have orphans, cycles, etc.
- these can all be represented by our type structure.

Hence:

- Our system as currently specified is under-constrained – it can exist in states we don't want it to exist in.

We need to constrain our model...

```
sig Task {
  predecessors: set Task
}

fact {
  no task: Task | task in task.^predecessors
}

pred Show {
  some predecessors
}

run Show for 4
```

module Alloy/Models/ACCU_2015/Tasks_V2

Let's add our first constraint - we want to have a task graph which is acyclic.

We can represent this as a fact.

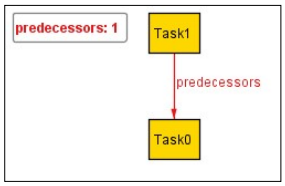
Facts are global.

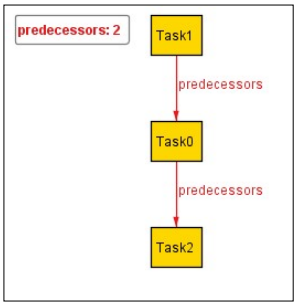
We've written a logical formula that says that none of our tasks should be reachable from itself.

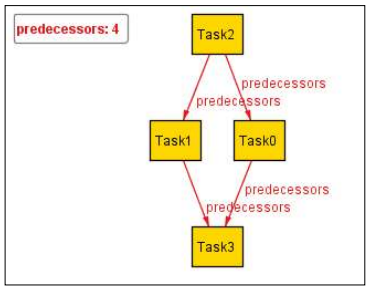
We navigate along relations using the familiar dot notation....

And the caret denotes the transitive closure operation.

Running some more simulations, here is what we get...







Now our examples are starting to look sensible...

What about connectivity?

```
open util/graph[Task]

sig Task {
  predecessors: set Task
}

fact {
  no task: Task | task in task.^predecessors
}

...

assert GraphIsWeaklyConnected {
  weaklyConnected[predecessors]
}

check GraphIsWeaklyConnected for 4
```

module Alloy/Models/ACCU_2015/Tasks_V3

Think back to our first examples – we saw a case where there was an orphan.

We want to check that there are no orphans produced by our model.

When we use the run command, Alloy searches for instances of a model.

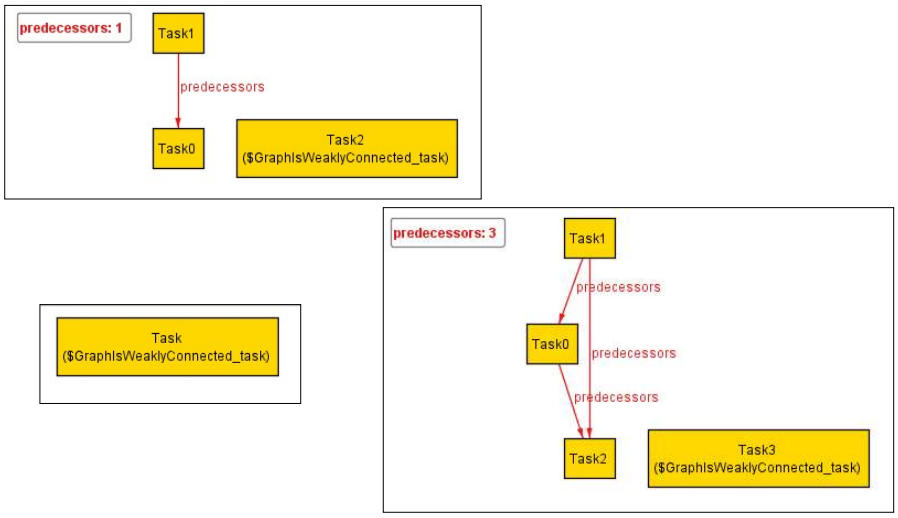
When we use the check command, Alloy searches for a counterexample of a model.

To eliminate orphan nodes, we can say that the graph is weakly connected.

This implies that all nodes are reachable from all other nodes, but may require us to traverse the wrong way along an edge.

To make life easier, we use a library function to structure the predicate.

Some counterexamples...



And when we search for counterexamples, we see that some exist...

So our model is still under-constrained.

So let's constrain further:

```
open util/graph[Task]

sig Task {
  predecessors: set Task
}

fact {
  no task: Task | task in task.^predecessors
  one task: Task | task.*predecessors = Task
}

...

assert GraphIsWeaklyConnected {
  weaklyConnected[predecessors]
}

check GraphIsWeaklyConnected for 4
```

module Alloy/Models/ACCU_2015/Tasks_V4

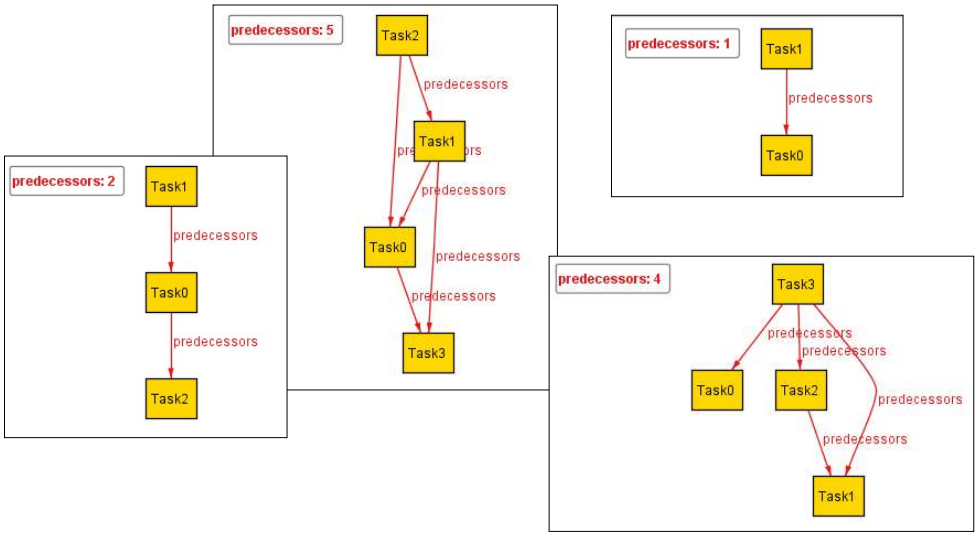
Let's add an extra constraint to the model.

We want all nodes to be strongly connected to a single root node – only traverse the right way down the arrows.

This implies that our graph will have no orphans – so should eliminate the problem shown by our GraphIsWeaklyConnected assertion.

Somewhat arbitrarily, we express this as a logical formula....

No counterexamples...
And the models are sensible...



These are starting to look good...

The simulations look sensible....

And there are no counterexamples to our assertion.

So we've got a pretty good handle on the structure of our model.

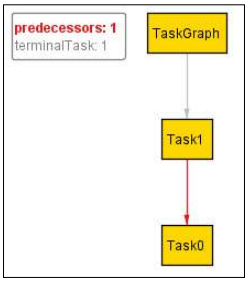
Finally, let's refactor a bit...

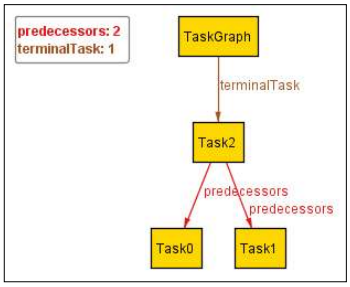
```
open util/graph[Task]

sig Task {
  predecessors: set Task
}

one sig TaskGraph {
  terminalTask: one Task
}
{
  dag[predecessors]
  rootedAt[predecessors, terminalTask]
}

pred Show { some predecessors }
run Show for 4

assert GraphIsWeaklyConnected { weaklyConnected[predecessors] }
check GraphIsWeaklyConnected for 4
```

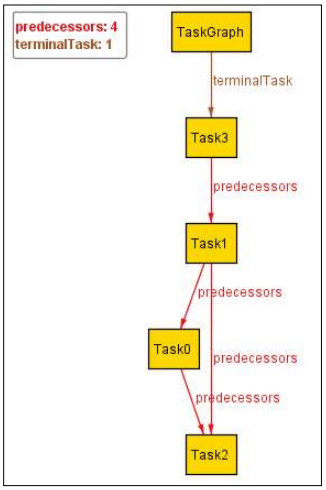
Introduce the TaskGraph signature to act as a container for the graph – it provides the "entry point" to our task graph.

We rework the constraints ever so slightly, to use library functions where possible.

The models look sensible...







And we repeat our simulation / check commands....

So what have we learned?

- Our basic abstraction was underconstrained
- By applying constraints we can see how the "state space" of our abstraction has been reduced.
- We've identified an invariant on our task graph, and this can be mapped back to our code.

How unconstrained? – plot the picture....

Extending our original example...

```
class Task {
  private:
   Task [] predecessors_;
  public:
    Task [] getPredecessors()      { return predecessors_; }
    void addPredecessor(Task pred) { predecessors_~= pred; };
};

class TaskGraph {
  private:
    Task terminalTask_;
    invariant() { //Code to check constraints... }
  public:
    Task getTerminalTask() { return terminalTask_; }
};
```

This is in D – where we have runtime contracts checking as part of the language...

So it is easy to add an invariant to a class.

The signatures map to types, and the constraints (in this case) map to invariants related to some of the types.

Next – behaviour....

- So far we have focused on structure.
- We can also model how the system changes over time.
- We'll carry on developing our model of task execution.

We've seen how to model structure.

We can also model behaviour...

Let's consider execution of an arbitrary set of tasks.

To start with consider just one task....

Here's a model for something that changes with time:

```
open util/ordering[Time]

abstract sig CompletionCode {}
one sig Pending, Completed extends CompletionCode {}

sig Time {}

sig Task {
  status: set CompletionCode->Time
}
{
  all t: Time | #status.t = 1
}

pred Show { some Task }
run Show for 4 but 1 Task
```

module Alloy/Models/ACCU_2015/TaskExecution_1

There are other ways of doing this – but I think this one works well for our problem.

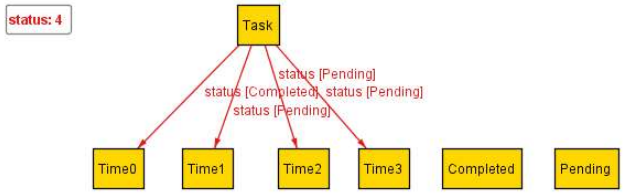
CompletionCode defines an enum

We have a Time signature which is ordered using library functionality...

And we have a Task which has a relation, status, which can take different values.

We apply the constraint that each time instance has exactly one status associated with it.

Some of our models have incorrect changes in status:

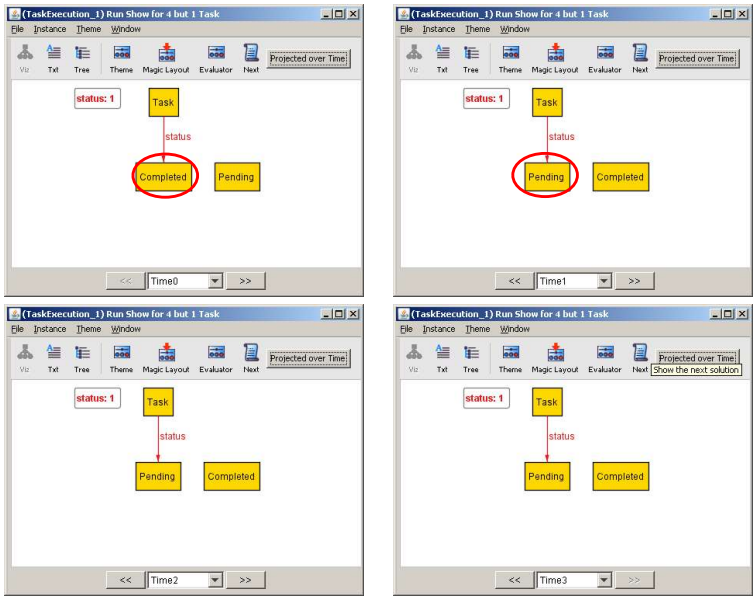


What we want is to have tasks which:

- start out as pending (always),
- may complete during the life of the model,
- never revert back to pending once completed.

Hence when you step through the models, you can see problems in some cases:

It's easier to see if you project on the time dimension:



We don't want to have models that start with status completed...

And move to pending....

Set the initial state:

```
open util/ordering[Time]

abstract sig CompletionCode {}
one sig Pending, Completed extends CompletionCode {}

sig Time {}

sig Task {
  status: set CompletionCode->Time
}
{
  all t: Time | #status.t = 1
}

fact traces {
  all task: Task | {
     task.status.first = Pending
  }
}
...
```

module Alloy/Models/ACCU_2015/TaskExecution_2

What we are doing is building up a state machine.

We define a fact, Traces, that describes how the system can change over time.

First we add an initial condition – at time t= 0. we are always pending.

Add predicates / fact to describe state machine...

```
open util/ordering[Time]

...

fact traces
{
  all task: Task | {
    task.status.first = Pending
    all t: Time - last | let t'= t.next |
      exec[task, t, t'] || noop [task, t, t']
  }
}

pred exec (task: Task, t, t': Time) { }
pred noop (task: Task, t, t': Time) { }

pred show { some Task }
run show for 4 but 1 Task
```

module Alloy/Models/ACCU_2015/TaskExecution_3

The fact traces now says that:

- initially we are in state pending,
- for all pairs of time atoms, t, t.next, we can either have exec or noop.
- the predicates currently have empty bodies – so won't do anything

Put appropriate constraints in the operations predicates:

```
open util/ordering[Time]

...

fact traces {
  all task: Task | {
     task.status.first = Pending
    all t: Time - last | let t'= t.next |
      exec[task, t, t'] || noop [task, t, t']
  }
}

pred exec (task: Task, t, t': Time) {
  task.status.t  = Pending
  task.status.t' = Completed
}

pred noop (task: Task, t, t': Time) {
  task.status.t = task.status.t'
}
```

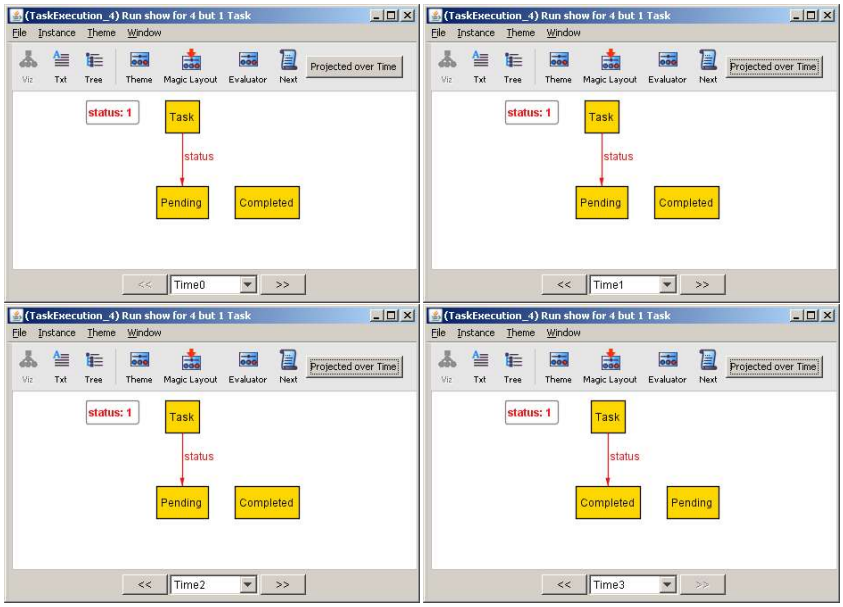
module Alloy/Models/ACCU_2015/TaskExecution_4

Complete model for a single, time-evolving task...

```
open util/ordering[Time]

abstract sig CompletionCode {}
one sig Pending, Completed extends CompletionCode {}

sig Time {}

sig Task {
  status: set CompletionCode->Time
}
{
  all t: Time | #status.t = 1
}

fact traces {
  all task: Task | {
     task.status.first = Pending
    all t: Time - last | let t'= t.next |
      exec[task, t, t'] || noop [task, t, t']
  }
}

pred exec (task: Task, t, t': Time) {
  task.status.t  = Pending
  task.status.t' = Completed
}

pred noop (task: Task, t, t': Time) {
  task.status.t = task.status.t'
}

pred show { some Task }
run show for 4 but 1 Task
```

module Alloy/Models/ACCU_2015/TaskExecution_4

Projecting over the Time signature shows us the ordering looks sensible....



Now integrate both parts of the model...

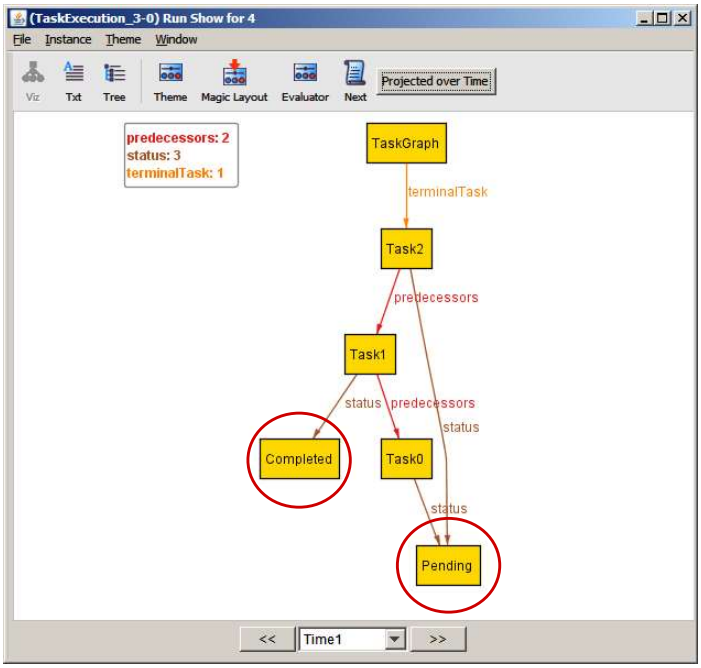
```
open util/graph[Task]
open util/ordering[Time]

abstract sig CompletionCode {}
one sig Pending, Completed extends CompletionCode {}

sig Time {}

sig Task {
  predecessors: set Task,
  status: set CompletionCode->Time
}
{
  all t: Time | #status.t = 1
}

...
```

module Alloy/Models/ACCU_2015/Tasks_3_0

Note how the graph of task structure doesn't change with time – so doesn't have a time element in its signature

However status does change with time.

Our first attempt is underconstrained…

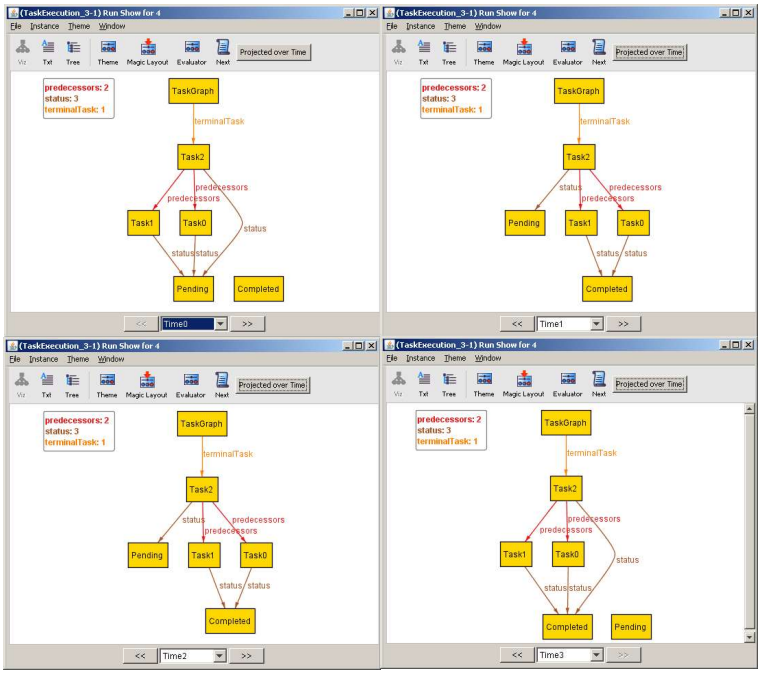


- We've not told the model about precedence…

Fixing this is straightforward...

```
pred exec (task: Task, t, t': Time) {
  task.status.t  = Pending
  task.status.t' = Completed

  no task.predecessors || predecessorsCompleted[task, t.prev]
}

pred noop (task: Task, t, t': Time) {
  task.status.t = task.status.t'
}

pred predecessorsCompleted (task: Task, t: Time){
  all predecessorTask: task.predecessors | predecessorTask.status.t =
Completed
}
```

And now the model looks sensible...



module Alloy/Models/ACCU_2015/Tasks_3_1

Implementing this in code gives...

```
void Execute()
  in { //Preconditions...
    assert (getStatus() == CompletionCode.Pending);
    foreach(pred; getPredecessors())
      assert (pred.getStatus() == CompletionCode.Completed);
  }
  out { //Postconditions...
    assert (getStatus() == CompletionCode.Completed);
  }
  body { //Work...
    ...
    status_ = CompletionCode.Completed;
  }
```

With care, you can get a really clean mapping between the abstract model and the implementation...

What value do we get from this?

- An efficient design / visualisation tool
- A powerful combination of elements:
  - ASCII models
  - Simulation of examples / counterexamples
  - Visualisation
- Can easily build partial models
  - Hence focus on areas of interest

We could keep on developing the model further, for example to add constraints around concurrent execution.

However hopefully you have got the basic idea of how it works.

So what?

*

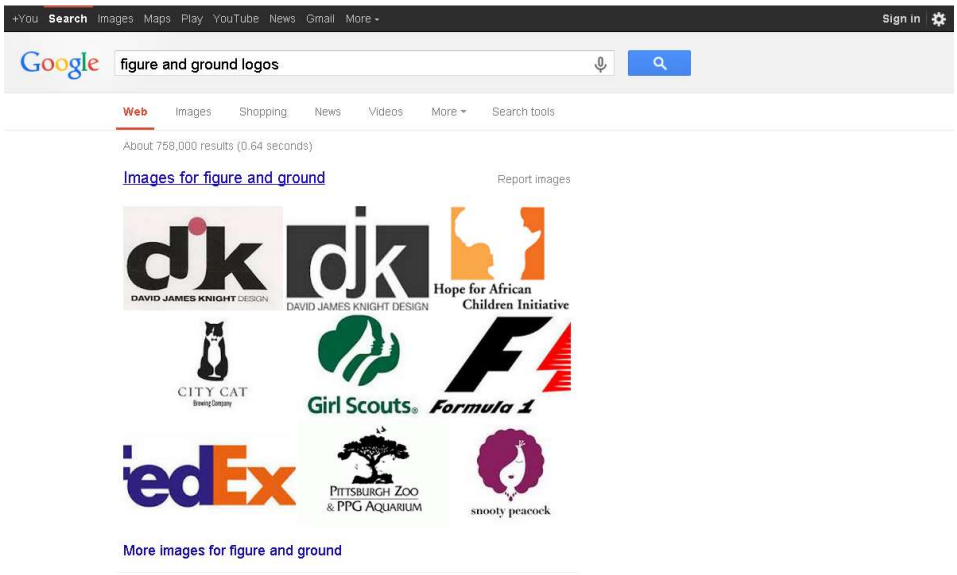
**Mark of FedEx*

Do you see the arrow?

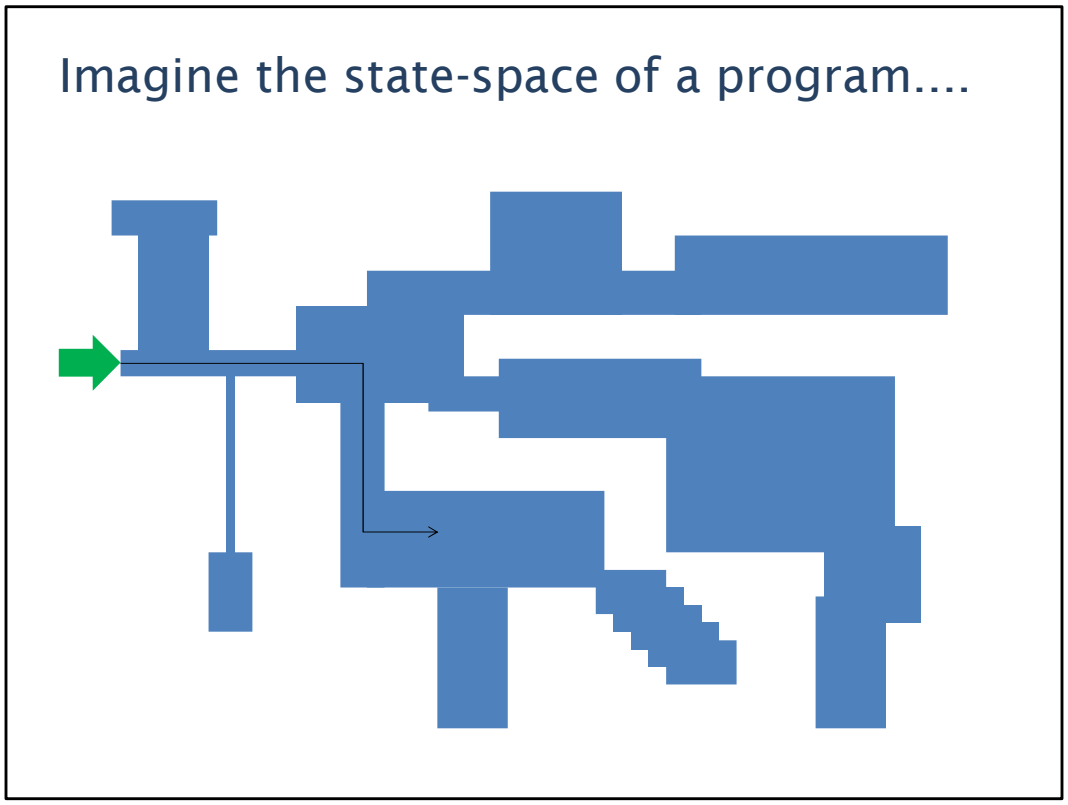
This is a great piece of design...
It was produced in 1994 by Lindon Leader of Leader Creative.

It uses the notion of figure and ground:
- Letters form the figure
- Everything else is the ground
- The ground contains an arrow.

Once you start looking...



There are loads...



- Why is this relevant to software?

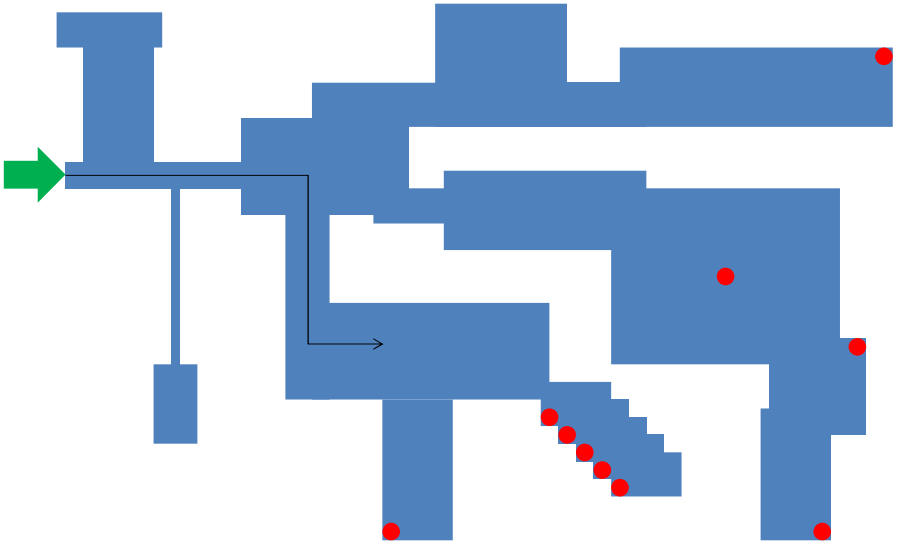
- Imagine the state-space of a program – just a figurative representation.

- The blue-coloured areas are states your program can exist in (not necessarily states we want or know about).

- When you start up the program, you are at the green arrow

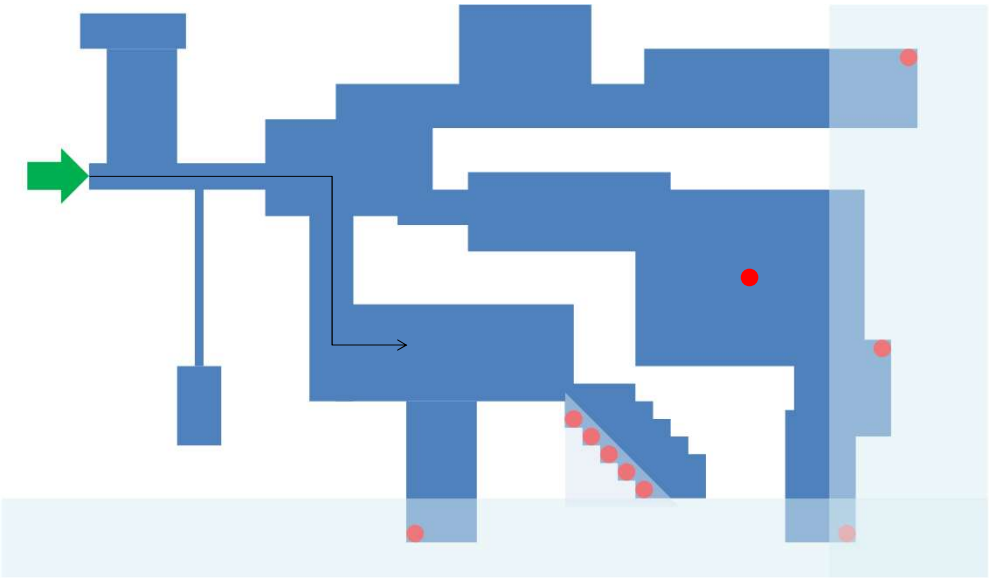
- As you use the program, you navigate through the different states.

We could get adverse behaviour...



- Red dots represent states where crashes occur.
- Remember playing battleships?

Maybe careful use of constraints will help...



- When you are writing code it is very easy to just think about what you want it to do – specific behaviours.
- This can occur with TDD or any excessively incremental approach....
- In that case we are focusing on the “figure”
- Using a model based on constraints forces you to think about the “ground”....
- This provides a very efficient way of reducing the state space of your system....
- And we can use it to help design both the structure of our types and their behaviour (contracts).
- You can define equivalence classes this way as well.

This process is facilitated by the use of a declarative language + simulation tools.

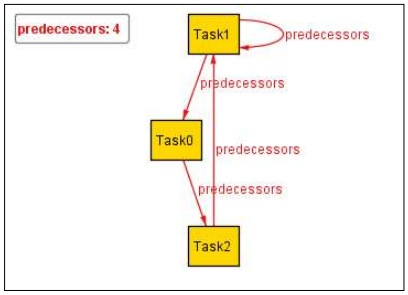
Remember Sapir-Whorf – “thoughts and behaviour are determined/influenced by language” =>

- Imperative languages are good for “figure”
- Declarative / constraint based languages are good for “ground”.

Think about the first simulation...

```
class Task {
  private:
    Task [] predecessors_;

  public:
    Task [] getPredecessors()      { return predecessors_; }
    void addPredecessor(Task pred) { predecessors_~= pred; };
};
```

```
sig Task {
  predecessors: set Task
}
```



For me, this is the really important slide in this deck.

It highlights the difference between an apparently obvious piece of code and what it does in reality.

Finding these problems early gives you a much better chance of producing high-quality, maintainable code.

In summary:

- The language/tool are very powerful
- Plenty of real-world examples / case studies
- I've been surprised by how using it has changed my view of the designs I've done.
- Using it incrementally has been straightforward
- Can develop models + code concurrently

Getting started

Tool and materials:

- http://alloy.mit.edu/alloy/
- “Software Abstractions” (Jackson)
- Tutorials (excellent)

Questions?