

# Using Alloy in Process Modelling

Chris Wallace

Faculty of Computing, Engineering & Mathematics  
University of the West of England  
Coldharbour Lane  
Bristol BS16 1QY

**tel:** +44(0)117 965 62 61

**fax:** +44(0)117 344 31 55

**email:** [chris.wallace@uwe.ac.uk](mailto:chris.wallace@uwe.ac.uk)

## Abstract

In this paper we examine the uses that the analysable language Alloy can play in Process modelling. We explore its application to descriptions of the organisation and its rules, to the description of processes and in the meta-modelling of process models.

Keywords: Alloy, process modelling,

## 1 Organisational Modelling

The task of understanding organisations, their structures and processes, sufficiently well to engage in organisational design or redesign with confidence is daunting. Analysts must draw on many perspectives, their underlying philosophies and their enabling techniques. For example, qualitative understanding might be gained articulated with SSM. Precision may be added with the use of casual models, and feedback loops explored with systems dynamics. Questions of capacity and timing may require discreet event simulation or CPM. Interaction between actors might be described with UML Activity models [xUML] or Role Activity Diagrams [xMO]. Structural relationships might be represented with UML Class Diagrams. Even this sample from the armamentarium of techniques shows the complexity of the task facing organisational analysts and designers, ranging as they do over the spectrum from qualitative to quantitative, from essentially social to essentially cognitive. Add to this the higher order ability to select the appropriate technique, use it on just the right part of the total problem, know when its value is exhausted and how to utilise the understanding gained in action.

This paper seeks to address those problems in understanding which arise from complex structures and processes, whose semantics are difficult to understand without some form of cognitive assistance. We argue that formal specification with Alloy should be added to the already-overstuffed bag of techniques. Our focus here is on the 'essential' [xCD] or conceptual model of an organisation, expressed as far as possible without implementation bias. We present a brief introduction to the Alloy language and its tools, and proceed to illustrate its use in developing and checking both static and dynamic organisational models and in more generic patterns and meta-models. We aim to give a flavour of the process of model-making as well as the nature of the final product. The complete scripts of the models developed for this paper are available from the author's web site[xCW].

## 2. The Alloy System

### 2.1 Background

Alloy is a textual language developed at MIT by Daniel Jackson and his team[xMJ]. Alloy develops ideas from Z and from the many attempts to formalise object modelling, especially Syntropy[xCD] and Catalysis[xWI]. Alloy is designed to be not merely descriptive but also to be analysable. Analysis in Alloy

takes the form of discovery of instances of a model, or counter-examples of an assertion about the model. It achieves this by exhaustive search for conforming instances within a space bounded by user-defined limits on the cardinality of entity sets. The search is accomplished by the construction of a complete Boolean formula for the model space, converted to Conjunctive Normal Form (CNF), which is passed to an off-the-shelf SAT (satisfiability) solver to find an assignment of values to the Boolean variables which satisfies the CNF. Progress in the development of SAT solvers is a very active research field, fuelled by the widespread use of this technique in planning problems such as the optimisation of chip circuit layout. Search speed has been improved by smarter pre-processing in Alloy to exploit symmetry and by improved search algorithms in SAT solvers. The Alloy software also provides tools to display a discovered instance of a model, either as a hierarchy or graphically using AT&T's GraphViz. The Alloy Analyser and related paper can be found on the Alloy web site [xAS]

This approach can, in general, only reveal the presence of errors in a model. It cannot, except in special cases, prove that no errors exist. Naturally, the scope of instances to be explored is limited. However models with up to  $10^{30}$  binary states have been reported to be handled in acceptable time. Moreover, the 'Small Scope' hypothesis [xAA] postulates that most errors can be found in models of small scope. This is supported by the experience of the MIT team in using Alloy on a range of problems, where Alloy has shown its value in demonstrating flaws in some well-known systems.

## 2.2 The Alloy language

Alloy uses first order predicate logic over the domain of Relations and typed Sets. Sets contain atoms which are immutable, indivisible and uninterpreted. Relations are the familiar relations from relational algebra and calculus. As in the relational algebra, sets and scalars are merely special cases: sets are relations of arity one, scalars are sets of cardinality one. The set operators (*union*, *intersection*, *difference*) and predicates (*equality*, *inclusion*), Boolean operators (*and* || *or* => *implication*, and *not*), and integer arithmetic (*+*, *-*) are supplemented by the relational join.  $r1.r2$  denotes the inner join on matching values in the rightmost field of  $r1$  with the leftmost field of  $r2$  i.e.  $\langle a,c \rangle$  is in  $r1.r2$  iff there exist  $b$  such that  $\langle a,b \rangle$  in  $r1$  and  $\langle b,c \rangle$  in  $r2$ . The relational join encompasses the familiar dot notation of object navigation when  $r1$  is a scalar.

The following examples use a subset of the full Alloy language. The full language has a richer set of constructs, and shortcut forms to allow models to be written more succinctly.

## 3. Structural Modelling

### 3.1 A university example

At the University of Borsetshire, the processes of student enrolment, assessment, course transfer and completion, as well as the slower processes of course modification, take place against a background of modules, courses, students and the complex relationships that bind them into a system. Part of this system is concerned with Modules (units of teaching and examination). Modules are inter-related in four important ways:

Level : Modules are grouped by the Level of difficulty, and Levels are ordered

Pre-requisites : A pre-requisite B means B must be passed before A is taken

Co-Requisites : A co-requisite B means B must be taken at the same time as A

Excludes : A excludes B means that, A has been passed, B cant be taken later – but not vice-versa (because B is contained within A for example)

We can show the situation in a UML Class diagram [Fig 1]

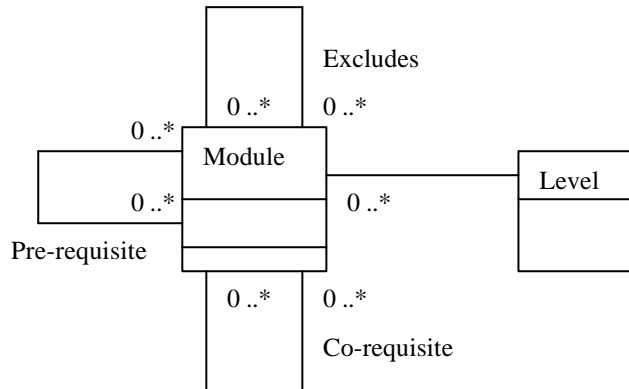


Fig 1. UML Class Diagram for Module.

One may observe that this diagram is not particularly informative. In particular, there is nothing to distinguish between the three recursive relationships despite the very different rules which pertain to each. Pre-requisites define an acyclic graph, co-requisites are an equivalence group (nearly) and excluded is non-symmetric. One can imagine that the meaning and interaction between these relationships can be the topic of hours of interesting discussion with users.

### 3.2 Alloy Model

We shall develop an Alloy model of this situation.

```
sig Level {}
```

Here we define a type to represent the level of difficulty of a Module. Later we will constrain Levels to be ordered. Note that Level actually denotes the *set* of Levels, not a type, although type checking uses a corresponding hidden type. When the Alloy Analyser executes the model, the cardinality of the Level set and all others must be defined. We can imagine Level as a set of unique surrogate keys.

```

sig Module {
  disj prereq, -- modules which must have been passed
  coreq,      -- modules which must be taken concurrently
  excludes -- modules which cannot be taken if this module taken
             : set Module, -- specifies a multiplicity of 0..*
  level : Level -- a multiplicity of 1
}

```

Here we have defined a second type to represent the units of teaching and assessment, together with three recursive (or homogeneous) relations  $\langle Module, Module \rangle$  and a relation with Level  $\langle Module, Level \rangle$ . We refer to these relations as `Module$prereq` or just as `prereq` if not ambiguous. The keyword `disj` specifies that the three relations `prereq`, `coreq` and `excludes` are disjoint. Hence for all Modules `m`, the sets `m.prereq`, `m.coreq` and `m.excludes` are also disjoint.

The module structure is further constrained by a number of rules or *facts*:

```
fact { -- module can't exclude itself
  all m : Module | m !in m.excludes
}
```

We read this as: for all `m` in set `Module`, `m` is not in the set of `m`'s excluded modules.

This fact is expressed in calculus form, but we could also express the same constraint in algebraic form, using some relational constants:

```
fact { -- module cant coreq itself
  no Module$coreq & iden[Module]
}
```

We read this as: the intersection of the relation `Module$coreq` with the identity relation  $\langle Module, Module \rangle$  is empty.

The `prereq` relation must be acyclic – no module can be a prerequisite of itself, directly or indirectly. Alloy's transitive closure operator,  $\wedge r$  (meaning  $r + r.r + r.r.r + r.r.r.r \dots$ ) is made analysable by the finite scope of Alloy models and this condition can be defined directly:

```
fact { -- no cycles in prerequisites
  no m : Module | m in m.^prereq
}
```

A pattern of constraints can be expressed as a generic function. For example, we can define the rules for any equivalence relation over the generic type `T` as:

```
fun Equivalence [T] (r : T->T) {
  iden[T] in r -- reflexive
  r.r in r     -- transitive
  ~r = r      -- symmetric  ~r denotes the inverse of the relation r
}
```

Here, Alloy implies conjunction between expressions within `{ .. }`.

Module co-requisites define an equivalence relation if the module itself is included

```
fact {
  Equivalence (Module$coreq + iden[Module])
}
```

We use the `Ord` generic, which will be imported from a separate Alloy file, to specify an ordering on `Level` and associated helper functions. We will use this generic to establish two further rules:

```
fact { -- prereqs must be at a lower level
  all m: Module |
    all p: m.prereq | OrdLT(p.level, m.level)
}
fact { -- coreqs must be at the same level
  all m : Module |
    all p : m.coreq | p.level = m.level
}
```

Now define a simple predicate which should be true for some interesting instances:

```
fun showRich () {
  some m: Module | some m.coreq
  some m: Module | some m.prereq
  some m: Module | some m.excludes
}
```

The `run` statement defines the predicate to try and the scope – here 5 modules and 3 levels

```
run showRich for 5 but 3 Level
```

The Alloy Analyser compiles the model definition. When we execute the `run` statement, the Analyser finds a solution, and visualises it as a graph. For this model, the instance it finds is one in which `Module_5` has a pre-requisite `MKodule_4` with a co-requisite `Module_4` that is excludes `Module_5`. [Fig 2]

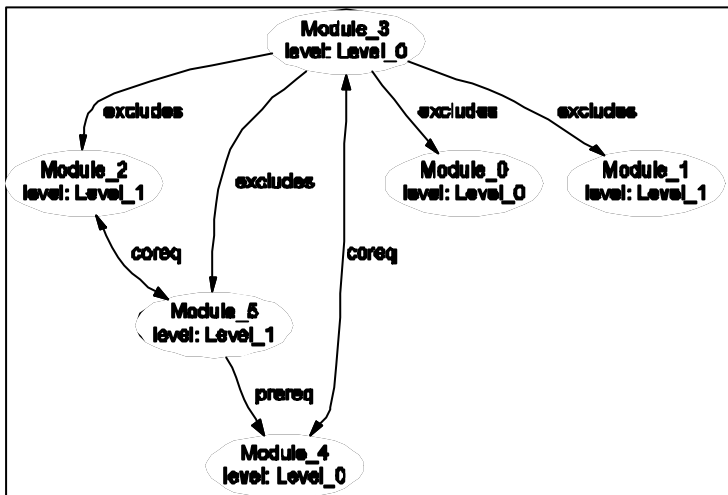


Fig 2. Instance of the model visualised by Alloy Analyser using GraphViz.

We attempt to fix this problem. We decide that an extra definition will help:

```

-- define the set of all modules which are required for module m
fun allreq(m : Module) : set Module {
  result = (m + m.coreq).*prereq    -- m.*r = m + m.^r
}

```

Here we have defined a formula with one argument that returns a set of Modules defined by the expression. We can then use this function to define an additional constraint:

```

fact { -- excluded modules must not be required, directly or indirectly
  all m : Module |
    no allreq(m) & m.excludes
}

```

Yet again we observe illegal configurations. After some thought we correct the function definition to:

```

-- define the set of all modules which are required for module m
fun allreq(m : Module) : set Module {
  result = m.*(prereq+coreq)
}

```

No incorrect examples are seen. However, we reason that the following should also be true and create an assertion:

```

assert reverse { all m : Module |
  -- module must not be excluded by modules required
  no allreq(m).excludes & m
}

```

```

check reverse for 6 but 2 Level

```

An instance is generated! We change the assertion to a fact to make this an invariant too. Further runs fail to show any incorrect examples.

### 3.3 Observations

Three observations may be made. First, much of the semantics remains unexpressed – the difference between passed modules and taken modules requires a model of the process a student undertakes as he passes through the University. Our model describes only those constraints between modules independent of their role in the student process. We will need to add this student process to fully describe these semantics.

Secondly, the greater part of the constraints of even the structural aspects cannot be expressed in diagrammatic form. The UML class diagram could be improved a little with additional adornments but

would still fail to fully depict the constraints. Undoubtedly, the Class diagram is useful, as one way to visualise the structure, but it is only a view, emphasising some features, such as the types involved in relations, and ignoring others. Research into richer graphical notation is on-going (e.g. Kent [xSK]), but it remains likely that some textual representation would continue to be required. This is clearly the case if the model is to be analysed. With UML, these additional constraints could be expressed in OCL [xOCL]. OCL has built-in data types which include numbers, sequences, sets and bags which make it more expressive than Alloy. However OCL has to handle a more complex meta-model which includes subclass and parametric polymorphism, operator overloading, multiple inheritance and introspection. OCL defines only the constraints additional to the Class diagram, has a more operational flavour and, crucially for our model, lacks a closure operation. Whilst there are some tools for parsing and simulating OCL, OCL is not analysable.

Thirdly, model building has a very different feel with Alloy. It feels more like programming, and offers the same rewards when a model works and frustrations when it does not. It seems the use of Alloy refutes Bertrand Meyer's slogan – “Bubbles don't crash”. Models typically crash either because the constraints are too restrictive, so that no instances are possible, or instances appear that are forbidden in the domain being modelled. Instance generation is extremely valuable in model development. As many have noted, for example Alexander[xCA], it is much easier to spot a misfit than it is to create a general theory of fitness. As the example illustrates, constraints can be surprisingly tricky to get right. This suggests a style of client-analyst cooperative analysis, the client suggesting or checking instances, the analyst developing the abstract model.

## 4. Dynamic Models

### 4.1 The Problem of Life

The Alloy language allows us to go beyond the representation of static structure. We can also model evolving systems through the use of states. Alloy has no in-built notion of state or operations, but these can be easily simulated. In fact this lack is an advantage since it allows the modeller to choose a meta-model appropriate to the task.

We can illustrate this technique with a model of the familiar Game of Life invented by the mathematician John Conway. It describes a matrix of cells and the rule by which each cell lives or dies from one generation (state) to the next. This model is a minor reworking of an example written by Bill Thies and Manu Sridharan at MIT. Fig 3 shows the UML Class diagram and the simple State diagram for a cell.

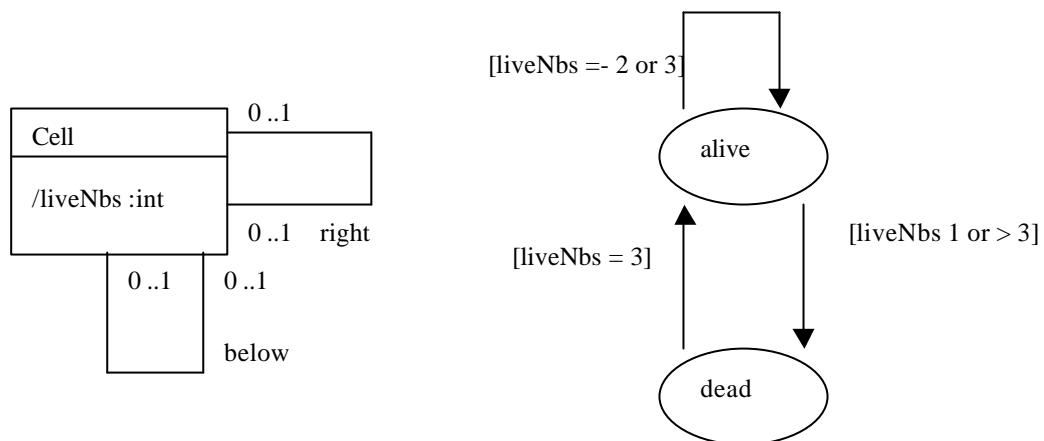


Fig 3. Class and State Chart for Life

### 4.2 Alloy model

Cell is modelled as a type. The matrix is defined by relationships between Cells. Those to the Cells to the right and below are essential, but other derived relationships are useful in expressing the model.

```
sig Cell {
```

```

right, below : option Cell, -- option specifies a 0/1 multiplicity
above, left : option Cell,
neighbours: set Cell
}

```

The derived relationships are constrained by facts:

```

fact { all c: Cell {
  c.above = c.~below
  c.left = c.~right
  c.neighbours =
    c.above.left + c.above + c.above.right +
    c.left + c.right +
    c.below.left + c.below + c.below.right }
}

```

Note that for a cell *c* at the top edge, the join of *c* with the relation *above* merely yields an empty set of type *Cell*. Likewise *c.~below*, which is equivalent to *below.c*, yields the empty set of type *Cell* so the invariant holds for these cells as well as for interior cells.

The top-leftmost *Cell* is modelled as a special case by defining a subset of *Cell*. *Extension* in Alloy allows additional sets to be defined, which may have additional relations, but these are just subsets, not subtypes. The only types are the *base* types. Subtyping as used in an Object-oriented language introduces much complexity and subsets are in any case frequently more useful, since subsets need not be mutually exclusive and can be defined intensively.

```

static sig Root extends Cell {} -- static denotes a cardinality of 1

```

As in the university example, it is no trivial problem to establish the minimum constraints to define a square matrix of *Cells*. We will omit these rules here.

The game proceeds in generations. At each generation, some *Cells* are alive. A instance of *Gen* represents the state of the system at a step.

```

sig Gen { live : set Cell }

```

The rules depend on the number of a *Cell*'s living neighbours, so we define a helper function:

```

fun LiveNeighbours(g: Gen, c: Cell) : set Cell {
  result = c.neighbours & g.live
}

```

Now define the transition rule for a *Cell* from one *Generation* to the next:

```

fun Trans(this, next: Gen, c: Cell) {
  let livers = LiveNeighbours(this, c) | -- let defines a synonym – it's not assignment!
  (c !in this.live && #livers = 3) -- a dead cell with 3 live neighbours becomes live
  -- # r denotes the cardinality of r
  => c in next.live
  else (
    (c in this.live && (#livers = 2 || #livers = 3))
    => c in next.live -- a live cells with 2 or 3 live neighbours stays alive
  else
    c !in next.live
  )
}

```

Now constrain the *Generations* to be ordered, and require the change between *Generations* for all *Cells* to be constrained by our rule. The first *Generation* is left unconstrained.

```

fact Life {
  all g : Gen - Ord[Gen].last |
  let next = OrdNext(g) |
  all c: Cell |
  Trans(g,next,c)
}

```

Let's see if we can find a final *Generation* that has living cells:

```

fun StillAlive () {
  some Ord[Gen].last.live
}

run StillAlive for 9

```

Alloy discovers a final configuration comprising four live cells in a square.

### 4.3 Observations

The exploration of dynamic population properties is simplified in Alloy by the reification of state, allowing relationships between states to be expressed straightforwardly. For example we can explore the Life system for stable configurations by constraining successive generations to have the same set of live cells.

This approach works well if there are many simple, tightly coupled processes. Here, process interaction is handled by shared state and all processes advance synchronously. Alloy has proved successful in modelling a range of distributed algorithms and it would be interesting to explore its application to organisational processes. Modelling change through global state changes constrained by transition rules shifts attention from a view of organisational processes based on a defined sequence of actions and communications, towards a more organic, perhaps we can say more *fluxist* perspective. We see a business as a collaborating society of actors and artefacts, constrained by networks of relationships and obeying local rules, with certain global behaviour, designed or otherwise, being the overall outcome. Cellular automata, such as the Game of Life, have long been considered to be demonstration of emergence, showing that stable, entity-like structures can emerge from simple low-level distributed processes.

## 5 Single Process models

### 5.1 Problem

Many organisational processes are complex with weak interactions between processes. Traditionally state machines have been used to model these situations. UML State Diagrams, Activity Diagrams and RAD models can be viewed as variants of this approach. These are widely used to visualise interaction between processes and the sequential constraints on actions. Tools for simulating state machines are available [xSMV]

However, as in our University case, actions can also be constrained by the structure in which they take place. It is often not possible to ignore the inter-dependence between structural and temporal constraints. Valuable though the diagrams are for visualisation, and for user interaction, understanding more complex interactions requires an approach which allows both types of constraint to be addressed in the same language.

In the University case study, we would like to describe the progress of a student as she enrolls in a course, takes and sometimes passes modules, advances to successive stages and finally graduates. This progress is constrained by University rules. A student:

- Must pass two modules at each stage before proceeding to the following stage
- Cannot take a module already passed
- Cannot take a module which has been excluded by a module already passed
- Cannot take a module unless all its pre-requisites have been passed
- Can only take modules which are at the same level as the current stage
- Must take co-requisite modules at the same time

A student may be suspended at any time and cannot progress until reinstated. A student who passed two modules at the final stage graduates. A student who has never been suspended and never failed a module graduates with Honours. Otherwise the graduate is an Ordinary graduate.

A simplified UML State diagram is shown in Fig 4.

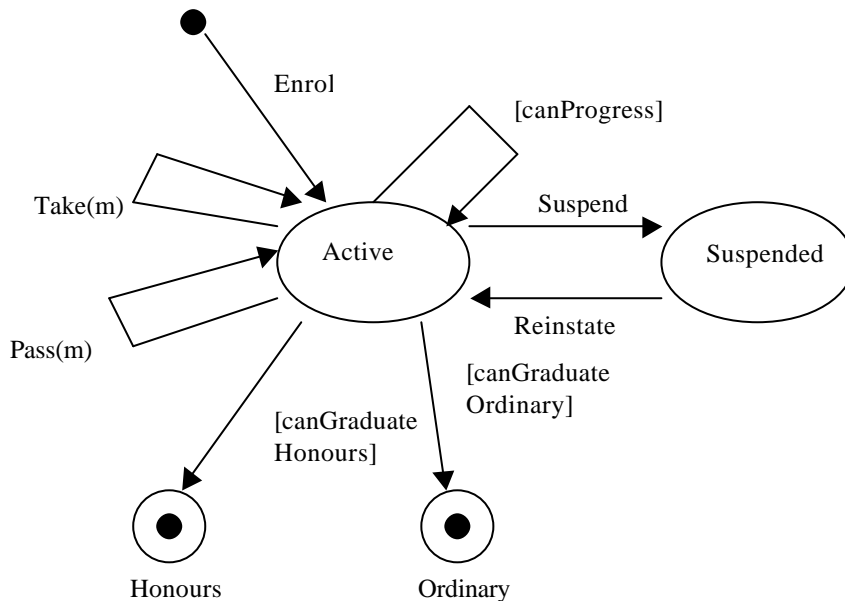


Fig 4. Simplified State diagram for the Student Process

## 5.2 Alloy Model

One approach to modelling this in Alloy is focus on a single student, since there is no interaction between students [for the purposes of this model at least].

We define a type that defines the set of relations which, taken together, define a student's state.

```

sig StudState {
  taking, passed : set Module,
  level : option Level,
  event : option Event
}
  
```

In addition, we need to distinguish between the 'states' in the state-machine. Here we refer to these as 'Modes' to avoid confusion. We model these as subsets of the Student State.

```

disj sig Initial, Active, Suspended, Final extends StudState {}
disj sig Ordinary, Honours extends Final {}
fact {Final in Ordinary + Honours }
  
```

In this approach, we choose to reify Events too, so that we can refer to the Event history itself.

We can use functions to define pre-conditions of transitions e.g.

```

fun canTake (m : Module, s: StudState) {
  m !in (s.passed + s.taking) -- cannot take a module already passed or taken
  m !in s.(passed + taking).excludes -- cannot take a module if excluded
  m.prereq in s.passed -- must have passed all the prereq
  m.level = s.level -- must be at the same level
  #s.taking < 2 -- can not take more than 2
}
  
```

and post-conditions e.g.

```

fun doTake (m : Module, s, s' : StudState) {
  s'.taking = s.taking + m
  s'.passed = s.passed
  s'.level = s.level
}
  
```

Note that we need to explicitly constrain the unchanged parts of a Student's state.

We define a rule which constrains the initial state, the final state and the relationship between successive states. At each step, some event occurs:

```
fact StudentProcess {
  doInitial(Ord[StudState].first)
  Ord[StudState].last in Final
  all s : StudState - Ord[StudState].last |
    let s' = OrdNext(s) |
      some e : Event |
        studtrans(s,s',e)
        s'.event = e
}
```

There are multiple transitions possible, and each is defined by the conditions that must obtain in the pre- and post states. Some are triggered by Events, others are merely guarded transitions.

```
fun studTrans (s,s' : StudState, e: Event) {
  --Initial ---- Enrol --> Active
  s in Initial && e in Enrol &&
  doEnrol(s,s') && s' in Active
  --Active --- Take(m) --> Active
  || s in Active && e in Take && canTake(e.m,s) &&
  doTake(e.m,s,s') && s' in Active
  etc
}
```

The declarative nature of Alloy now allows us to simply compose the full model by textual inclusion of the structural model defined above. We can check that a given state is reachable by:

```
fun showHonours { some Honours }

run showHonours for 12 but 2 Level, 6 Module
```

The Alloy Analyser will now attempt to find an instance of the full model, and the result, if any, visualised.

We believe that when the student finishes there should be no excluded modules:

```
assert noexcluded{
  -- no excluded modules taken when finished
  let s = Ord[StudState].last |
    no (s.taking + s.passed) & (s.taking + s.passed).excludes
}

check noexcluded for 12 but 2 Level, 4 Module
```

Alloy finds an example where an excluded module is taken *before* the module which excludes it. This example helps to understand that although exclusion is symmetric as far as pre- and co-requisites, it is asymmetric dynamically. The semantics of this relation cannot be understood without considering both aspects.

## 5.4 Observations

Reification of both states and events allows us to express more complex constraints than a simple state machine would allow. For example, the transition to Honours depends not on the current state alone but on previous states, and on the event history. In Alloy we can express this directly as:

```
fun canGraduateHonours (s : StudState) {
  -- can only finish with Honours if never Suspended and no module Failed
  canGraduate(s)
  no OrdPrevs(s) & Suspended
  no OrdPrevs(s).event & Fail
}
```

This avoids the need to introduce artificial states which a state machine description would require, without compromising our ability to analyze the model.

One constraint has not been represented here: constraints on co-requisites require modules to be taken several at a time, since any ordering of single takings would violate the constraint.

The full Alloy model seems somewhat verbose but much of the bookkeeping, such as managing the frame conditions and the event tracking could be added by a veneer supporting a selected meta-model.

## 6 Multiple Processes

### 6.1 Problem

Several styles of modelling multiple Processes are possible. Which style is appropriate will depend upon the problem. One approach is to model each Process separately. Consider a separate Tutor process, modelled in a similar style to Student. These two processes are independent. Suppose under some conditions the tutor may transit to BadMood state. We may then add a precondition for suspension:

```
--Active -- Suspend --> Suspended
|| s in Active && e in Suspend && && some BadMood &&
doNothing(s,s') && s' in Suspended
```

Since Student and Tutor states effectively interleave freely, this transition is possible if the Tutor ever enters BadMood. However, if the Tutor process can only transit to BadMood when a student is Suspended, no instances of the model which included Suspension could exist. Handling shared events is possible with some difficulty.

Another approach is model the global state of the system, as we did with the Game of Life. Suppose we want to model Modules as processes too. We will assume they have a simple state, either available or not available. The events of Module are restricted to the following:

Toggle(m)	which toggles the state of module m
Take(m)	a shared event with the Student Process : module becomes unavailable

### 6.2 Alloy Model

We represent the full System State as:

```
sig SysState {
  taking, passed : set Module,
  level : option Level,
  event : option Event,
  available : set Module,
  smode : Smode
}
```

In this formulation, the Mode of the Student has been reified, rather than modeled as subsets. Because Alloy is 1<sup>st</sup>-order and hence lacks reflection, one cannot simply express the frame condition that the Student doesn't change Mode from state to state, necessary now since some events (those for Module alone) must leave the Student unchanged. However, to misquote Brian Randel, any problem can be solved with an extra reification.

The sequence of SysStates is defined as:

```
fact UnivProcess {
  doStudInitial(Ord[SysState].first)
  doModInitial(Ord[SysState].first)
  Ord[SysState].last.smode in Final
  all s : SysState - Ord[SysState].last |
    let s' = OrdNext(s) |
```

```

        some e : Event {
            studTrans(s,s',e)
            all m : Module | modTrans(m,s,s',e)
            s'.event = e
        }
    }

```

and the transition for a module defined as:

```

fun modTrans(mod : Module, s,s' : SysState, e : Event) {
    e in Take && e.m = mod => mod !in s'.available
    else
        e in Toggle && e.m = mod && mod in s.available =>
            mod !in s'.available
    else
        e in Toggle && e.m = mod && mod !in s.available =>
            mod in s'.available
    else
        mod in s.available => mod in s'.available
    else
        mod !in s.available => mod !in s'.available
}

```

The Student transitions can be modified to depend upon the module state, for example by adding

```

    m in s.available

```

to the pre-condition for canTake(m) in Student.

If we need to take this further, and model multiple students then we can alter SysState to contain ternary relations such as:

```

    taken, passed : Student ->+ Module,
    smode : Student -> Smode,

```

so that each Student has its own state in each SysState. studTrans would be modified accordingly.

## 6.3 Observations

The state space expands greatly as multiple processes are added. Given the current state of SAT solvers, the analyst must choose the area to be modelled with care, since even a model of this size requires significant running time to solve.

## 7 Implicit Sequencing

### 7.1 The Problem

Organisational processes are often, at heart, concerned with the construction of a complex aggregation of parts. For example Kawalek and Bustard [xKB ] use a RAD model to define the process of writing an appraisal, from proposal, through market analysis, legal opinion and report writing. Sequential models such as RAD can over-constrain these processes and must be explicit in handling parallel paths.

An approach which may be appropriate is to use the sequencing implicit in constraints in a object model. Consider a simple model of a house. A house is composed of a base, four walls and a roof. Each wall has a frame and cladding. We can show this as a UML class model[ Fig 5].

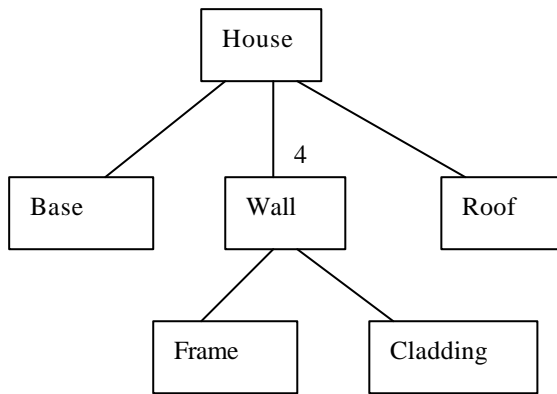


Fig 5. Class diagram of complete House

This model of the completed house would need to be weakened to include houses under construction as well. [Fig 6] However, this model is now too weak: houses with only a roof, or a base and a roof but no walls. Feasible partial houses are a subset of these instances. The House must have a Base before any Walls, and all Walls before a Roof. However the Walls may have all the frames made and then clad or any interleaving of these activities.

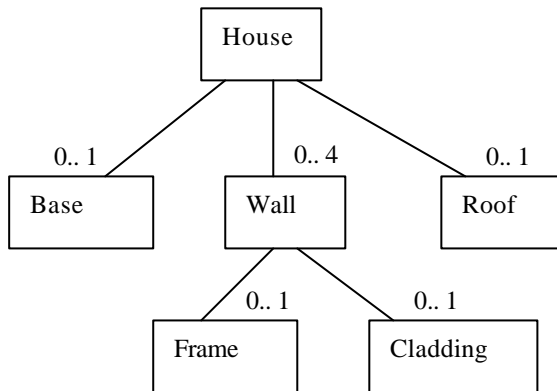


Fig 6. Class diagram of incomplete House

## 7.2 Alloy Model

In Alloy we can represent the House as:

```

sig House {
  base : option Base,
  walls : set Wall,
  roof : option Roof
}

sig Wall {
  frame : option Frame,
  cladding : option Cladding
}
  
```

Additional constraints describe the dependencies between parts:

```

fact { all h : House |
  (some h.walls => one h.base) -- if it has any walls it must have a base
  (some h.roof => #(h.walls & CompleteWall) = 4) -- if it has a roof it must have 4 complete walls
}

sig CompleteWall extends Wall{}
  
```

```
fact { CompeteWall : { w : Wall | one w.frame & one w.cladding } }  
fact { all w : Wall | some w.cladding => one w.frame }
```

With some additional rules to ensure that houses are proper aggregates, the Alloy Analyzer can be used to instantiate only feasible partial houses, or to check that the rules allow a complete house to be built at all.

### 7.3 Application to the Student Process

We can use this approach to model a student's career as a composition of parts, each Stage requiring the completion of the prior Stage, modules taken before being passed or failed. The work required to develop the full set of invariants, in place of the mixture of invariants and pre- and post-conditions in the state model identified an additional weakness in the expression of the co-requisites semantics, but we lost some of the semantics of suspension.

### 7.4 Observations

In this approach to process modelling, we exploit the duality between a process and its product. The House model describes not only the partial stages of completion, but it also implicitly describes the activities which create the structure. This declarative formulation minimally constrains the sequence of activities. Simulation with Alloy allows intermediate states to be examined, although we cannot step through the construction process itself. Nevertheless, this approach may be useful because it encourages a more declarative style of process description.

Here too the Alloy model presents challenges in writing and visualising. We anticipate that Alloy would be supplemented with a veneer which would ease the formulation of the dependency style suggested here, perhaps even generated from a diagrammatic Class model.

## 8 Generalisations

Patterns such as those described in Martin Fowler's excellent collection of business data models [xMF] must also be carefully defined. Fowler used Class and Sequence diagrams but often the most interesting part of the model, such as hierarchical relationships, could only be expressed informally. Moreover, it is inevitable that without analytical support, even the most competent modeller will sometimes omit constraints. Yet if the pattern is to be used in practice, the more subtle, less easily visualised constraints form just as important a part of the full model as those which are given prominence in the diagram.

Meta-models, which we can regard as more generalised patterns defining a style of modelling, are ubiquitous. For example, in a recent book on process modelling [xBXN], 6 of the 20 chapters contain descriptions of proposed meta-models. Making meta-models precise is hard work, as the debate over the precise semantics of UML models illustrates. Jackson found that even with OCL definitions, constraints were too weak, allowing instances of the meta-model which are clearly unintended.

More generally still, Alloy has a role in the education of analysts. A deep understanding of fundamental constructs such as sets, relations, functions, the meaning of entity and state models is more attainable in the Alloy environment. After seeing the chaos which ensues when a frame condition is omitted in a state model, one gains an almost visceral understanding of the concept. Experimentation and instantiation is invaluable in developing a deep understanding of the meaning of a model, as a description of a space of possibilities. The dialectic between the general and the particular surely lies at the heart of the modelling task. Developing declarative models is hard, especially for those more used to imperative programming. A solid model is the result of numerous drafts, puzzles posed and sometimes solved, fresh approaches tried.

## 9 Conclusion

Alloy and its analyser appear to be a valuable addition to the process analyst's armamentarium. The ability to handle both structural and temporal aspects in the same model is particularly useful in expressing

problem semantics. As with any modelling technique however, the analyst needs to develop a keen sense of where the precision of an Alloy model would be of benefit within the totality of analytical and design activities.

## Acknowledgements

The author would like to thank Michael Jackson, Daniel Jackson and the members of Process Modelling Group at the University of the West of England for their help and support.

## References

- [xCA] C. Alexander, Notes on the Synthesis of Form, Harvard Univ. Press , 1964
- [xAA] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "Small Scope Hypothesis". (Submitted for publication.) September 2002.
- [xBKN] D. Bustard, P. Kawalek and M. Norris (ed), System Modeling for Business process Improvement ,Artech House 2000
- [xCD] S. Cook and J. Daniels, Designing Object Systems: Object-oriented Modeling with Syntropy, Prentice-Hall 1994
- [xWI] D.F.D'Souza and A.C.Wills , Objects, Components and Frameworks with UML: The Catalysis Approach, Addison Wesley Longman, 1998
- [xMF] M. Fowler, Analysis Patterns, Addison-Wesley 1997
- [xUML] M. Fowler with K. Scott, UML Distilled ,Addison-Wesley 1997
- [xDJ] D. Jackson, D. I. Shlyakhter and M. Sridharan, A Micromodularity Mechanism Proc. ACM SIGSOFT Conf. Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC '01) Vienna, September 2001
- [xKG] P. Kawelak and R.M. Greenword, The Organisation, the Process and the Model in D. Bustard, P. Kawalek and M. Norris (ed), System Modeling for Business process Improvement ,Artech House 2000 pp61-80
- [XSK] S. Kent, Constraint Diagrams: Visualising Invariants in Object Oriented Models IN: Proceedings of OOPSLA97, ACM Press 1997
- [xMO] M A Ould, Business Processes - Modelling and Analysis for Re-engineering and Improvement , John Wiley & Sons, Chichester, 1995.
- [xOCL] J. Warmer and A. Kleppe, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, 1998
- [xAS] <http://sdg.lcs.mit.edu/alloy>
- [xCW] <http://www.cems.uwe.ac.uk/~cjwallac/alloy>