

## Using Alloy in Process Modelling

C. 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.

## 1. Organisational Modelling

Process models such as RAD and UML Activity models are useful in describing specific sequences of action and inter-action in an organisation's workflow. However such descriptions depend implicitly on definition of the actors and artefacts involved and their structural relationships. For example, in a university, any description of the student processes will refer to notions of student, module, course and assessment, and are governed by the rules that constrain the inter-relationships between them. We envisage a process such as 'Change course' as a disturbance in this network of entities, creating or destroying instances, changing their attributes and state, making and breaking relationships between entities. But through all this, something remains stable, some truths persist, and must persist. The description of this fabric is a pre-condition to making sense of any process description.

Modelling languages for these entities and relationships are some of the oldest used in systems development. [First place of course goes to the flow models of which RAD and Activity diagrams are the latest manifestations.] Numerous diagrammatic forms exist, culminating in the Class diagrams of UML. Valuable though such forms have proved to be, structural complexity is frequently beyond the expressive powers of diagrams and many alternative textual languages have been developed. This paper will examine one such language, Alloy.

## 2. The Alloy System

Alloy is a textual language developed at MIT by Daniel Jackson and his team[1]. Alloy develops ideas from Z and from the many attempts to formalise object modelling. 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, conversion to Conjunctive Normal Form (CNF) which is passed to an off-the-shelf SAT 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 to exploit symmetry and by improved search algorithms. 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.

This approach can, in general, only reveal the presence of errors in a model. However the experience of the MIT team in using Alloy on a range of problems has shown its value in demonstrating flaws in some well-known systems.

### 3. The Alloy language

Alloy uses first order predicate logic over the domain of relations. 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*, in *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  $r1$  or  $r2$  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.

### 4. A university example

At the University of the West of England, the processes of student enrolment, assessment, course transfer and completion, and 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. Here is an example of part of that structure modelled in Alloy.

```
sig Level {}
```

Here we define an entity set, Level to represent the three levels (years) on a Course. Later we will constrain them to be ordered.

Note that Level is a Set, not a type, although type checking uses a corresponding hidden type. We can imagine Level as a set of unique surrogate keys. In the generated instances these will be manifest as Level-0, Level-1 etc.

```
sig Module {
  disj prereq, -- modules which must have been passed
  coreq, -- modules to be taken concurrently
  excluded -- modules which are incompatible
  : set Module, -- specifies a multiplicity of 0..*
  level : Level -- a multiplicity of 1
}
```

Here we have defined a second entity set Module 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 for example Module\$prereq or just as prereq if not ambiguous. The keyword disj specifies that for all Modules, the three sets prereq, coreq and excluded are 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.excluded
}
```

We read this as: for all  $m$  in 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.

Recurrent patterns can be expressed as a generic function. Here we define the condition for an acyclic graph, using Alloy's transitive closure operator.  $\hat{r}$  is  $r + r.r + r.r.r + r.r.r.r$  etc, made realisable by the finite scope of Alloy models.

```
fun Acyclic [t] (r : t->t) { no x:t | x in x.^r } -- a->b is the relational crossproduct
```

We can use this generic to require that pre-requisites form a DAG

```
fact { -- no cycles in prerequisites
  Acyclic(Module$prereq)
}
```

Now define a second generic for equivalence:

```
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
}
```

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 file, to specify an ordering on Level. This introduces a relation  $\langle \text{Level}, \text{Level} \rangle$  constrained to represent an ordering, with associated helper functions. We will use this 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 instances:

```
fun show () {
  some m: Module | some m.coreq
  some m: Module | some m.prereq
  some m: Module | some m.excluded
}
```

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

```
run show for 5 but 3 Level
```

The Alloy software compiles this script and executes it. It finds a solution, but the instance is one in which one module's co-requisite module has a pre-requisite that is excluded. We attempt to fix this problem with another constraint, re-compile and re-execute. Yet another instance confounds our expectations.

Modelling 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 is wrong. Perhaps Bertrand Meyer's slogan 'Bubbles don't crash' is negated.

## 5. Alloy processes

The Alloy language allows us to go beyond the representation of static structure. We can also model evolving systems through the use of states. In this approach, we regard the system as progressing through a sequence of states, which are modelled simply as an entity set constrained by order.

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.

Each Cell is modelled as an entity. The matrix is defined by relationships between Cells. Those to the Cells to the right and below are necessary, but other 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
}
```

The top-leftmost Cell is modelled as a special case by defining a subset of Cell. Extension allows new sets to be defined, which may have additional relations, but these are still subsets, not subtypes. The only types are the base types. Sub-typing as used in an Object-oriented language introduces much complexity and subsets are in any case frequently more useful.

```
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 full model can be found at [3]

The game proceeds in generations. At each generation, the population of living Cells changes.

```
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 in one Generation to a Cell in the next:

```
fun Trans(this, next: Gen, c: Cell) {
  let livers = LiveNeighbours(this, c) |
  (c !in this.live && #livers = 3) => -- dead cells with 3 live neighbours becomes live
  -- # r denotes the cardinality of r
  c in next.live
  else (
  -- live cells with 2 or 3 live neighbours stays alive
  (c in this.live && (#livers = 2 || #livers = 3)) =>
  c in next.live 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.

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.

Process modelling using this declarative global state is very different from the imperative style on which RAD and Activity models are based. Alloy has proved successful in modelling a range of distributed algorithms but it is 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.

More prosaically we can describe operations in a business process as state transitions, with pre-and post conditions, but allow a non-deterministic sequence of operations. We can still check that desirable states are reachable by letting Alloy search for a sequence with the defined end state. Thus in our University model, we could define:

```
sig Student {
  taking, passed : set Module
  level : Level
}

sig Action {}
static disj sig Enrol, Take, Pass, Fail, Graduate extends Action {}
```

Operations can then be defined, such as:

```
fun take(s, s' : Student, m : Module) {
  ( m !in (s.passed + s.taking) && s.level = m.level && m !in (s.passed + s.taking).excluded
  ) =>
  (s'.taking = s.taking + m && s'.passed = s.passed && s'.level = s.level )
}
```

Then we can define a trace by the selection of an action at each step, evaluation of any guards, followed by the respective operation to change state. The full model is at [3].

## 6. Meta-modelling

The debate over the precise semantics of UML models continues, but the language in which these semantics are expressed ( OCL + the UML class diagrams) is itself problematic. Researchers, for example Eshuis [2]in his PhD thesis must adopt their own definitions to proceed. This thesis maps Activity Models, or at least the temporal aspects to a hyper-graph that can be translated into a form suitable for model checking with NuSMV. This approach to checking activity models and state-charts looks very promising. However even in this careful work, the author's attempt to define the structure of a hyper-graph, when translated by this author to an Alloy model, shows that getting constraints right without the help of model checking is very hard. As Jackson found with the UML meta-model, constraints are too weak and allow clearly invalid instances to be generated.

## 7. Reflections

Developing declarative models is hard for experts. It is harder for our students, but at least Alloy allows the modeller to bridge the gap between the general and the particular. This dialectic seems crucial in the development of a real understanding of the meaning of a model, perhaps even of the meaning of modelling itself.

## References

1. Jackson, D, Shlyakhter, I and Sridharan, M. *A Micromodularity Mechanism* Proc. ACM SIGSOFT Conf. Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC '01) Vienna, September 2001
2. Eshuis, H *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD Thesis University of Twente 2002
3. <http://www.cems.uwe.ac.uk/~cjwallac/alloy>

