Alloy: A Lightweight Object Modelling Notation

Daniel Jackson, ACM Transactions on Software Engineering, 2002

Presented By: Steven Stewart, 2012-January-23

Alloy: 2002 to present

Software is built on abstractions. Pick the right ones,
and programming will flow naturally from design...

- **Software Abstractions: Logic, Language, and Analysis** (Revised Ed.)

    Daniel Jackson (MIT Software Design Group)

Modelling

Almost all recent development methods factor out the structural aspect of a software system, usually called the ‘**object model**’

- **Lightweight models**
  - description (or specification) of basic structures
  - based on a small syntax
  - automation and analysis

What is Alloy? (2002)

A modelling notation that can express
a *useful range* of structural properties

- **Alloy** is a lightweight object modelling notation for describing...
  - basic structure
  - intricate constraints
  - operations (how structures change dynamically)

What is Alloy? (2012)

A modelling notation that can express
a *useful range* of structural properties

- **Alloy** is a "structural modelling language" based on first-order logic for expressing complex constraints and behaviours

  http://alloy.mit.edu/alloy/faq.html

- **Alloy** is a declarative specification language for expressing complex structural constraints and behaviour in a software system

  http://en.wikipedia.org/wiki/Alloy_(specification_language

What is Alloy? (2012)

Logic, Language, and Analysis

- **Logic**: all structures are represented as relations; structural properties are expressed with a few simple operators; states and executions both described using constraints

What is Alloy? (2012)

Logic, Language, and Analysis

- **Logic**: all structures are represented as relations; structural properties are expressed with a few simple operators; states and executions both described using constraints
- **Language:** syntax added to logic for structuring descriptions; a flexible type system; simple module system

What is Alloy? (2012)

Logic, Language, and Analysis

- **Logic**: all structures are represented as relations; structural properties are expressed with a few simple operators; states and executions both described using constraints
- **Language:** syntax added to logic for structuring descriptions; a flexible type system; simple module system
- **Analysis:** constraint solving; simulation (finding instances of states or executions); checking (finding counterexamples)

What is Alloy? (1997-2012)

Logic, Language, and Analysis

- **Logic:**
  - first-order logic + transitive closure (for describing reachability constraints

What is Alloy? (1997-2012)

Logic, Language, and Analysis

- **Logic:**
  - first-order logic + transitive closure (for describing reachability constraints
- **Language:**
  - declarative: describe the effect of a behaviour, not the mechanism

What is Alloy? (1997-2012)

Logic, Language, and Analysis

- **Logic:**
  - first-order logic + transitive closure (for describing reachability constraints
- **Language:**
  - declarative: describe the effect of a behaviour, not the mechanism
- **Analysis** (we'll get to this later)

What Alloy is not?

A modelling notation that can express
a *useful range* of structural properties

- **Alloy** is not for describing...
  - dynamic interactions between objects
  - syntactic structure in an implementation (i.e., class hierarchy and packages)

What Alloy is not?

A modelling notation that can express
a *useful range* of structural properties

- **Alloy** is not for describing...
  - dynamic interactions between objects
  - syntactic structure in an implementation (i.e., class hierarchy and packages)
- **DynAlloy** (Frias, Galeotti, and Pombo, 2005-present), an extension to Alloy to describe dynamic properties of systems using ‘actions’

Comparisons

...a large class of structural models can be described in Z without higher order features, and can thus be analyzed efficiently

- **Alloy** is based on **Z**, but simplifies the underlying semantics
- **Z** was not as amenable to analysis, so Alloy eliminates features that make analysis hard
- **OCL** deemed too complicated -- a consequence of trying to accommodate notions from object-oriented programming

Alloy: Language (2002)

Language syntax, type rules, and semantics -- simple and concise.

problem ::= *decl* formula*
decl ::= *var* : *typexpr*
typexpr ::=
  type
  | *type* -> *type*
  | *type* => *typexpr*

formula ::=	
expr in *expr*	subset
| ! *formula*	negation
| *formula* && *formula*	conjunction
| *formula* || *formula*	disjunction
| all *v* : *type* | *formula*	universal
| some *v* : *type* | *formula*	existential

expr ::=	
| *expr* + *expr*	union
| *expr* & *expr*	intersection
| *expr* - *expr*	difference
| *expr* . *expr*	navigation
| ~ *expr*	transpose
| + *expr*	closure
| {*v* : *t* | *formula*}	set former
| *Var*	

Var ::=	
| *var*	variable
| *Var* [*var*]	application

$$\frac{E \vdash a : S,\ E \vdash b : S}{E \vdash a \text{ in } b}$$

$$\frac{E, v : T \vdash F}{E \vdash \text{all } v : T \mid F}$$

$$\frac{E \vdash a : S \rightarrow T,\ E \vdash b : S \rightarrow T}{E \vdash a + b : S \rightarrow T}$$

$$\frac{E \vdash a : S \rightarrow T,\ E \vdash b : S \rightarrow U}{E \vdash a \,.\, b : U \rightarrow T}$$

$$\frac{E \vdash a : S \rightarrow T}{E \vdash {\sim}a : T \rightarrow S}$$

$$\frac{E \vdash a : T \rightarrow T}{E \vdash +a : T \rightarrow T}$$

$$\frac{E, v : T \vdash F}{E \vdash \{v : T \mid F\} : T}$$

$$\frac{E \vdash a : T \Rightarrow t,\ E \vdash v : T}{E \vdash a[v] : t}$$

$M : formula \rightarrow env \rightarrow boolean$
$X : expr \rightarrow env \rightarrow value$
$env = (var + type) \rightarrow value$
$value = \mathbb{P}\,(atom \times atom) + (atom \rightarrow value)$

$M[a \text{ in } b]\ e = X[a]\ e \subseteq X[b]\ e$
$M[!\,F]\ e = \neg\ M[F]\ e$
$M[F \,\&\&\, G]\ e = M[F]\ e \wedge M[G]\ e$
$M[F \,||\, G]\ e = M[F]\ e \vee M[G]\ e$
$M[\text{all } v : t \mid F]\ e = \bigwedge \{M[F](e \oplus v \mapsto x) \mid (x, unit) \in e(t)\}$
$M[\text{some } v : t \mid F]\ e = \bigvee \{M[F](e \oplus v \mapsto x) \mid (x, unit) \in e(t)\}$

$X[a + b]\ e = X[a]\ e \cup X[b]\ e$
$X[a \,\&\, b]\ e = X[a]\ e \cap X[b]\ e$
$X[a - b]\ e = X[a]\ e \setminus X[b]\ e$
$X[a \,.\, b]\ e = \{(x,z) \mid \exists y.\ (y,z) \in X[a]\ e \wedge (y,x) \in X[b]\ e\}$
$X[{\sim}a]\ e = \{(x,y) \mid (y,x) \in X[a]\ e\}$
$X[+a]\ e = \text{the smallest } r \text{ such that } r\,;r \subseteq r \wedge X[a]\ e \subseteq r$
$X[\{v : t \mid F\}]\ e = \{(x, unit) \in e(t) \mid M[F](e \oplus v \mapsto x)\}$
$X[v]\ e = e(v)$
$X[a[v]]\ e = (e(a))\ (e(v))$

Alloy: Relations

- In fact, beneath the surface, everything in Alloy is a relation

Alloy: Relations

- In fact, beneath the surface, everything in Alloy is a relation
- Scalars are simply unary relations with a single tuple
  - e.g., RootDir = {<d0>}

Alloy: Relations

- In fact, beneath the surface, everything in Alloy is a relation
- Scalars are simply unary relations with a single tuple
  - e.g., RootDir = {<d0>}
- We can express structure with relations, using three different styles of logic supplemented with set, relational, and logical operators

Alloy: Language (2012)

- Supports three styles of expressing logic
  - predicate calculus, navigation expression, relational calculus

Alloy: Language (2011)

- Supports three styles of expressing logic
  - **<u>predicate calculus</u>**, navigation expression, relational calculus

two kinds of expressions:

relation names (used as predicates) and *tuples* formed from quantified variables

"names in an address book are mapped to at most one address"

all n: Name, d, d': Address | n->d **in** address **and** n->d' **in** address **implies** d = d'

Alloy: Language (2011)

- Supports three styles of expressing logic
  - predicate calculus, **navigation expression**, relational calculus

    expressions denote sets that are formed by navigating from quantified variables along relations

    "names in an address book are mapped to at most one address"

all n: Name | **lone** n.address

Alloy: Language (2011)

- Supports three styles of expressing logic
  - predicate calculus, navigation expression, **relational calculus**

  expressions denote relations,
  and there are no quantifiers

  "names in an address book are mapped to at most one address"

no ~address.address - **iden**

Alloy: Language (2011)

- **Set operators:**
  - union, intersection, difference, subset, equality

Alloy: Language (2011)

- **Set operators:**
  - union, intersection, difference, subset, equality
- **Relational operators:**
  - product, join, transpose, transitive-closure, reflexive closure

Alloy: Language (2011)

- **Set operators:**
  - union, intersection, difference, subset, equality
- **Relational operators:**
  - product, join, transpose, transitive-closure, reflexive closure
- **Logical operators:**
  - negation, conjunction, disjunction, implication, bi-implication

Alloy: Analysis

- The relational specification is translated into a boolean formula, which is handed over to a backend SAT-solver
- If a model has at least one *instance* satisfying all constraints, then the model is said to be *consistent*
- To check an assertion, we look for a model (or instance) of its negation in order to produce a counter-example
- The **Alloy Analyzer** enables the ability to check models within a finite scope -- failure to find a model within that scope does not prove that the formula is inconsistent
- An exhaustive scope of 10 gives more coverage of a model than hand-written test cases ever could! (D. Jackson: “small scope hypothesis”)

Alloy: Analysis

> Program testing can be used to show the presence of bugs,
> but never to show their absence
>
> *Edsger W. Dijkstra*

- **Small-scope hypothesis** (from *Software Abstractions*)
  - Most bugs in code elude testing

Alloy: Analysis

Program testing can be used to show the presence of bugs,
but never to show their absence

Edsger W. Dijkstra

- **Small-scope hypothesis** (from *Software Abstractions*)
  - Most bugs in code elude testing
  - Instance finding has more extensive coverage than traditional testing

Alloy: Analysis

Program testing can be used to show the presence of bugs,
but never to show their absence

Edsger W. Dijkstra

- **Small-scope hypothesis** (from *Software Abstractions*)
  - Most bugs in code elude testing
  - Instance finding has more extensive coverage than traditional testing
  - Most bugs have small counterexamples -- if you examine all small cases, you're likely to find a counterexample

Alloy: Analysis

Program testing can be used to show the presence of bugs,
but never to show their absence

Edsger W. Dijkstra

- **Small-scope hypothesis** (from *Software Abstractions*)
  - Most bugs in code elude testing
  - Instance finding has more extensive coverage than traditional testing
  - Most bugs have small counterexamples -- if you examine all small cases, you're likely to find a counterexample
- **Summary:** covering of ALL cases (potentially billions) in a small scope will uncover most flaws!

The Alloy Style of Modelling

Alloy is amenable to fully automatic semantic analysis that can provide checking of consequences and consistency

- In this style of modelling, a model can be developed incrementally, and explored at each step using the analyzer

Alloy: Analysis

Alloy is amenable to fully automatic semantic analysis that can provide checking of consequences and consistency

- **Simulation**
  - View instances of your model. Correct model. Fix them. Try again.
- **Checking**
  - Check assertions against the specification.
  - Find counterexamples. Correct model (fix bugs). Save yourself from future hassles!

The clock is ticking...

- If we have enough time, I'll show you a sample Alloy specification for a file system -- somewhat similar to the one in the paper, but simpler...
- Until then, we will move on to "Experience and Evaluation"

Experience and Evaluation

- 2000 - 2002
  - analysis of a resource discovery system
  - design of an air traffic control system component
  - reformulation of some essential properties of Microsoft COM's query interface
  - translation from OCL to Alloy of UML core metamodel, which was shown to be consistent using the Alloy Analyzer

Experience and Evaluation

- 1997 - present
  - addition of quantifiers, higher arity, relations, polymorphism, subtyping, and signatures
  - **Alloy4** uses a model finder called **Kodkod**, demonstrating significant improvements in performance and scalability
- Alloy has been used to model... name servers, network configuration protocols, access control, telephony, scheduling, document structuring, key management, cryptography, instant messaging, railway switching, filesystem synchronization, semantic web

http://alloy.mit.edu/alloy/faq.html

Experience and Evaluation

- Over 600 citations (Google Scholar)
- Over 100 case studies
- A dozen languages translated to Alloy
- Select tools built on Alloy4 (Alloy + Kodkod)
  - Forge, Squander, Alloy4Eclipse, DynAlloy, TACO, Equals Checker, Nitpick, Margrave

http://alloy.mit.edu/alloy/applications.html

Concluding Remarks

A modelling notation that can express
a *useful range* of structural properties

- Alloy emerged from a series of observations
  - a large class of structural models can be described in **Z** without higher-order features, and can thus be analyzed 'efficiently'

Concluding Remarks

A modelling notation that can express
a *useful range* of structural properties

- Alloy emerged from a series of observations
  - a large class of structural models can be described in **Z** without higher-order features, and can thus be analyzed 'efficiently'
- Alloy combines familiar and well-tested ideas from existing notations

Concluding Remarks

A modelling notation that can express
a *useful range* of structural properties

- Alloy emerged from a series of observations
  - a large class of structural models can be described in **Z** without higher-order features, and can thus be analyzed 'efficiently'
- Alloy combines familiar and well-tested ideas from existing notations
- Alloy's kernel language represents an attempt to capture the "essence of object modelling"

Concluding Remarks

A modelling notation that can express
a *useful range* of structural properties

- Alloy emerged from a series of observations
  - a large class of structural models can be described in **Z** without higher-order features, and can thus be analyzed 'efficiently'
- Alloy combines familiar and well-tested ideas from existing notations
- Alloy's kernel language represents an attempt to capture the "essence of object modelling"
- The ability to experiment with a model and check properties *changes the very nature of modelling*

Alloy Specification (2002)

- The model is divided into *paragraphs*...
- *domain paragraph*: declares sets of atoms (e.g., file system objects, directory entries, names)
- *state paragraph*: declares state components, which are static sets representing fixed classifications of objects (e.g., *File* and *Dir*)
- *definition paragraph*: used to define relations in terms of other state components (e.g., the *parent* of *o* follows the *entries* relation backward)
- *invariants*: these are facts about the model (e.g., any two distinct entries have different names)

Alloy Specification (2012)

- *Signatures* -- introduces a set of **atoms**
- *Facts* -- constraints that are assumed always to hold
- *Assertions* -- constraints that are expected to follow from the facts of the model; the analyzer checks assertions to detect design flaws
- *Predicates* -- constraints that you don't want to record as facts; (e.g., you might want to analyze a model with a particular constraint included, and then excluded)
- *Functions* -- a named expression intended for reuse

Example Alloy Specification

```
abstract sig FSObject {}
sig File, Dir extends FSObject {}
```

```
sig FileSystem {
   root: Dir,
   live: set FSObject,
   contents: Dir lone-> FSObject,
   parent: FSObject ->lone Dir
} {
   no root.parent
   live in root.*contents
   parent = ~contents
   contents in live->live
}
```

- A file system is composed of file system objects (*FSObject*), which are files (*File*) and directories (*Dir*)

- A *FileSystem* has a root directory, a *live* set of files and directories, a relation describing the *contents* of directories, and a *parent* relation that describes parent/sub-directory relationships

Facts about our model:
no root has a parent
FSObjects are reachable from root
parent is the inverse of *contents* relation
contents only defined on live FSObjects

Example Alloy Specification

```
abstract sig FSObject {}
sig File, Dir extends FSObject {}
```

```
sig FileSystem {
    root: Dir,
    live: set FSObject,
    contents: Dir lone-> FSObject,
    parent: FSObject ->lone Dir
} {
    no root.parent
    live in root.*contents
    parent = ~contents
    contents in live->live
}
```

We expect that a file system will have only one root directory, which we can check via an assertion and a check command.

```
// File system has one root
assert oneRoot {
    all fs: FileSystem { #fs.root = 1 }
}
check oneRoot for 5
```

If one exists, the analyzer will find a counterexample within the specified scope (5, here), which advises us to revise our model.