CISC/CMPE 422, CISC 835:
Formal Methods in Software Engineering

Syntax and Semantics of Alloy

Juergen Dingel
Oct 2019

Syntax: Expressions

Atomic expressions

none	keyword evaluating to the empty set
name	name of signature or a field
var	variable

Composite expressions

{*var* : *expr* | φ}	set comprehension where φ is formula
~*expr*	inverse
^*expr*	transitive closure
**expr*	reflexive, transitive closure
expr + *expr*	union
expr – *expr*	difference
expr & *expr*	intersection
expr -> *expr*	Cartesian (cross) product
expr.*expr*	relational composition

Syntax: Formulas

Atomic formulas

Let r, s denote expressions:

`r in s`	*subset: every element of r also is an element of s*
`r = s`	*set equality: r and s contain the same elements*

Composite formulas

Let f and g denote formulas:

`!f`	*negation: not f*
`f && g`	*conjunction: f and g*
`f || g`	*disjunction: f or g*
`f => g`	*implication: if f then g*
`f <=> g`	*equivalence: f if and only if g*
`all x : r	f`	*universal quantification: f true for all x in r*
`some x : r	f`	*existential quantification: f true for at least one x in r*
`one x : r	f`	*f true for exactly one x in r*
`lone x : r	f`	*f true for at most one x in r*
`no x : r	f`	*f true for no x in r*
`some r`	*r contains at least one element*
`one r`	*r contains exactly one element*
`lone r`	*r contains at most one element*
`no r`	*r contains no element*

Semantics

Let *Spec* be an Alloy specification (a.k.a., module or model)

- *Spec* consists of
  - Signatures
  - Constraints, i.e., predicate logic formula φ_{Spec} over signatures
- Questions:
  - What exactly are **satisfying instances *I*** of Spec?
    - $I = (D^I, F^I, P^I)$
    - How are D^I, F^I, and P^I defined?
    - What are the symbols in *F* and *P*?
  - Formal definition of when φ_{Spec} holds in *I*?
    - Satisfaction relation: $I \vDash \varphi_{Spec}$
    - Evaluation function: $eval^I(\varphi_{Spec})$
  - Is Alloy's analysis **sound** and **complete**?

Semantics (Cont'd)

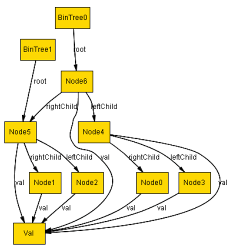
```
module BinTrees

sig Val { }

sig Node {
    leftChild : lone Node,
    rightChild : lone Node,
    val : Val
}

sig BinTree {
    root : lone Node
}

fun nodes[b : BinTree] : set Node {
    (b.root).*(leftChild + rightChild)
}

pred isLeaf[n : Node] {
    no n.leftChild && no n.rightChild
}

fact Facts {
    // no cycles
    all b : BinTree | no n : nodes[b] | n in n.^(leftChild + rightChild)
    // at most one parent
    all b : BinTree | all n : nodes[b] | lone n.~(leftChild + rightChild)
    // all nodes belong to at least one tree
    Node in (BinTree.root).*(leftChild + rightChild)
    // left child iff right child
    all b : BinTree | all n : nodes[b] | some n.leftChild iff some n.rightChil
    // children are different
    all b : BinTree | all n : nodes[b] | !isLeaf[n] => (n.leftChild != n.rightC
    // balanced
    all b : BinTree | # (b.root.leftChild).*(leftChild + rightChild) = # (b.roc
}
```

- Function symbols *F*
  - F_{Sig} = {Val, Node, BinTree} // signature names, all arity 0
  - F_{Attr} = {leftChild, rightChild, val, root} // attribute names, all arity ≥ 2
  - F_{Op} = {*, +, &, ~, …} // relational operators, all arity ≥ 1
- Predicate symbols *P* = {in, lone, some, …}
- Constraints in BinTrees compiled into predicate logic formula $\varphi_{BinTrees}$ over function symbols *F*, predicate symbols *P*, variables *V*

- An instance is a type-consistent assignment of values to function symbols in F_{Sig} and F_{Attr}

- A satisfying instance is an instance s.t. $\varphi_{BinTrees}$ holds with symbols in F_{Op} and *P* having their standard meaning

For instance




```
Eval> BinTree
  {BinTree$0, BinTree$1}
Eval> Node
  {Node$0, Node$1, Node$2, Node$3, Node$4, Node$5,
  Node$6}
Eval> Val
  {Val$0}
Eval> root
  {BinTree$0->Node$6, BinTree$1->Node$5}
Eval> rightChild
  {Node$4->Node$0, Node$5->Node$1, Node$6->Node$5}
Eval> leftChild
  {Node$4->Node$3, Node$5->Node$2, Node$6->Node$4}
Eval> val
  {Node$0->Val$0, Node$1->Val$0, Node$2->Val$0,
  Node$3->Val$0, Node$4->Val$0, Node$5->Val$0,
  Node$6->Val$0}
```


Semantics (Cont'd)

Will focus on **kernel/core language** of Alloy containing only an **adequate** set of operators and connectives, leaving out operators and connectives that can be defined in terms of adequate ones:

For instance,

- some x : expr | φ = !all x : expr | !φ
- one x : expr | φ = (some x : expr | φ) && (all y : expr | φ => y=x)
- no x : expr | φ = !some x : expr | φ
- lone x : expr | φ = (no x : expr | φ) || (one x : expr | φ)

and

- some expr = some x: expr | x=x
- no expr = no x: expr | x=x

and

- sig S {a : lone T} = sig S {a : set T} and fact S {all s : S | lone s.a}
- sig S {a : T} = sig S {a : set T} and fact S {all s : S | one s.a}

Syntax of Alloy Kernel

Specifications <Spec>

<Spec> ::= <SigList> <FactList>

<SigList> ::= <Sig> | <Sig> <SigList>

<Sig> ::= **sig** *s* **{}** | **sig** *s* **{**<AttrList>**}**

<AttrList> ::= <Attr> | <Attr> <AttrList>

<Attr> ::= *a* **: set** <Type>

<Type> ::= *s* | *s* **->** <Type>

<FactList> ::= <Fact> | <Fact><FactList>

<Fact> ::= <φ>

where $s \in F_{Sig}$ and $a \in F_{Attr}$

Syntax of Alloy Kernel (Cont'd)

Expressions <Expr>

<Expr> ::= *name* | *var* | **none** | <Expr> <BinOp> <Expr> | <UnOp> <Expr>

<BinOp> ::= **+** | **&** | **-** | **.** | **->**

<UnOp> ::= **~** | **^**

where $name \in F_{Sig} \cup F_{Attr}$ and $var \in V$

Formulas <φ>

<φ> ::= <Expr> **in** <Expr> |
!<φ> |
<φ> **&&** <φ> |
all *var* **:** <Expr> **|** <φ>

where $var \in V$

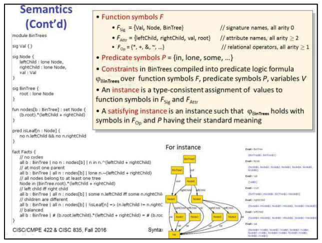
Symbols

So, a specification *Spec* will contain the following symbols:

- Function symbols $F = F_{Sig} \cup F_{Attr} \cup F_{Op}$ where
  - F_{Sig} = set of signature names in *Spec*, all with arity 0
  - F_{Attr} = set of all attribute names in Spec
    where for $a \in F_{Attr}$, arity(a) = k+1, if *a*: **set** s_1 -> … -> **set** s_k
  - F_{Op} = <BinOp> ∪ <UnOp> = {**+**, **&**, **-**, **.**, **->**, **~**, **^**} with expected arities
- Predicate symbols *P*
  - *P* = {**in**} with arity(**in**) = 2

A satisfying instance $I = (D^I, F^I, P^I)$ of *Spec*

- interprets symbols in F_{Op} and *P* as expected
- assigns values to symbols in $F_{Sig} \cup F_{Attr}$ s.t.
  - types of attributes are respected
  - all formulas in <FactList> hold



Domain D^I

$\mathcal{D}_{s_i}$ = *infinite set of unique atoms for interpretation of signature* s_i

$\mathcal{D}^{\mathcal{I}}_{s_i} = \{(d) \mid d \in \mathcal{D}_{s_i}\}$

for all signature names s_i in $\mathcal{F}_{Sig}$.

$$\mathcal{D}^{\mathcal{I}} = \{d \subseteq \mathcal{D}^{\mathcal{I}}_{s_i} \mid s_i \in \mathcal{F}_{Sig}\} \cup \{d \subseteq \mathcal{D}^{\mathcal{I}}_{s_1} \times \ldots \times \mathcal{D}^{\mathcal{I}}_{s_n} \mid \exists a \in \mathcal{F}_{Attr}.type(a) = (s_1, \ldots, s_n)\}$$

Example: Assume that *List* and *Node* are signature names in $\mathcal{F}_{Sig}$ and that $\mathcal{F}_{Attr}$ contains attribute names *head* and *next* with arity two and types

$$type(head) = (List, Node)$$
$$type(next) = (Node, Node)$$

and that φ_{Facts} and φ_P are formulas using these names. Also assume that the atoms in $\mathcal{D}_{Node}$ and $\mathcal{D}_{List}$ are denoted by $N0, N1, N2, \ldots$ and $L0, L1, L2, \ldots$, respectively. The semantic domain $\mathcal{D}^{\mathcal{I}}$ will, e.g., contain the following elements:

$\{(N0), (N1), (N2)\}$	possible interpretation of signature *Node*
$\{(N0)\}$	possible interpretation of signature *Node*
$\emptyset$	possible interpretation of signature *Node*
$\{(L0)\}$	possible interpretation of signature *List*
$\{(L0), (L1), (L2), (L3)\}$	possible interpretation of signature *List*
$\{((N0), (N1)), ((N1), (N2))\}$	possible interpretation of attribute *next*
$\{((L0), (N0))\}$	possible interpretation of attribute *head*
$\emptyset$	possible interpretation of attribute *head*

The semantic domain $\mathcal{D}^{\mathcal{I}}$ will, e.g., not contain the following elements:

$\{((N0), (N1), (N2)), ((N1), (N2), (N3))\}$	no attribute of matching type
$\{((N0), (L0)), ((N1), (L1))\}$	no attribute of matching type

Interpretation of function symbols F^I

As expected

$\mathcal{F} = \mathcal{F}_{Op} \cup \mathcal{F}_{Name}$

Case 1: $f \in \mathcal{F}_{Op}$, **i.e.,** $f \in$ {none, ~, ^, +, &, -, ., ->}
The interpretation of all these symbols is *fixed*:

- $\texttt{none}^{\mathcal{I}} = \emptyset \in \mathcal{D}^{\mathcal{I}}$
- $\sim^{\mathcal{I}}$ = the unary function that reverses the input relation $d \in \mathcal{D}^{\mathcal{I}}$ i.e., $(d_k, d_{k-1}, \ldots, d_2, d_1) \in \sim^{\mathcal{I}}(d)$ iff $(d_1, d_2, \ldots, d_{k-1}, d_k) \in d$
- $\hat{}^{\mathcal{I}}$ = the unary function that builds the *transitive closure* of binary relations d for all $s \in \mathcal{F}_{Sig}$ with $type(d) = (s, s)$
- $+^{\mathcal{I}}$ = the binary function that returns the *union* of its input
- $\&^{\mathcal{I}}$ = the binary function that returns the *intersection* of its input
- $-^{\mathcal{I}}$ = the binary function that returns the *difference* of its input
- $.^{\mathcal{I}}$ = the binary function that returns the *relational composition* of its input
- $\texttt{->}^{\mathcal{I}}$ = the binary function that returns the *cartesian product* of its input

Case 2: $f \in \mathcal{F}_{Name} = \mathcal{F}_{Sig} \cup \mathcal{F}_{Attr}$

If f is a signature name, i.e., $f \in \mathcal{F}_{Sig}$, then the interpretation $f^{\mathcal{I}}$ of f will be given by a subset of $\mathcal{D}^{\mathcal{I}}_f$, i.e., the set of atoms associated with f:

$$f^{\mathcal{I}} \subseteq \mathcal{D}^{\mathcal{I}}_f \in \mathcal{D}^{\mathcal{I}}, \qquad \text{for all } f \in \mathcal{F}_{Sig}$$

If f is an attribute name, i.e., $f \in \mathcal{F}_{Attr}$ with $type(f) = (s_1, \ldots, s_k)$, then the interpretation $f^{\mathcal{I}}$ of f is given by a relation respecting the type of f, i.e.,

$$f^{\mathcal{I}} \subseteq (\mathcal{D}^{\mathcal{I}}_{s_1} \times \ldots \times \mathcal{D}^{\mathcal{I}}_{s_k}) \in \mathcal{D}^{\mathcal{I}}, \qquad \text{for all } f \in \mathcal{F}_{Attr} \text{ with } type(f) = (s_1, \ldots, s_k)$$

Interpretation of predicate symbols P^I

Definition of interpretation $\mathcal{P}^{\mathcal{I}}$ of predicate symbols The interpretation of the predicate symbols $\mathcal{P}$ = {in} is also fixed. The predicate symbol in is interpreted as subset check, i.e.,

$$\texttt{in}^{\mathcal{I}} : \mathcal{D}^{\mathcal{I}} \times \mathcal{D}^{\mathcal{I}} \to \mathbb{B}$$

such that for all $d_1, d_2 \in \mathcal{D}^{\mathcal{I}}$

$$\texttt{in}^{\mathcal{I}}(d_1, d_2) = \begin{cases} true, & \text{if } d_1 \subseteq d_2 \\ false, & \text{otherwise} \end{cases}$$

When exactly does φ hold in *I*?

Definition of a satisfaction relation Just like in Predicate Logic, the satisfaction relation $\models$ is intended to relate an instance $\mathcal{I}$ and a formula φ as defined above

$$\mathcal{I} \models \varphi$$

iff φ holds in $\mathcal{I}$. Let $\mathcal{V}$ denote the variables in φ. The definition of the satisfaction relation is similar to that for Predicate Logic in Section 4.2.

$$\mathcal{I} \models \varphi \text{ iff } \mathcal{I}, \emptyset \models \varphi$$

where $\mathcal{I}, l \models \varphi$ for mappings (we also called them environments) $l : \mathcal{V} \to \mathcal{D}^{\mathcal{I}}$ is defined inductively by

$$\begin{array}{ll} \mathcal{I}, l \models p(e_1, \ldots, e_n) & \text{iff } p^{\mathcal{I}}(evalE_l^{\mathcal{I}}(e_1), \ldots, evalE_l^{\mathcal{I}}(e_n)) = true \quad \text{for all } p \in \mathcal{P} \\ \mathcal{I}, l \models \texttt{!}\varphi & \text{iff it is not the case that } \mathcal{I}, l \models \varphi \\ \mathcal{I}, l \models \varphi \texttt{ \&\& } \psi & \text{iff } \mathcal{I}, l \models \varphi \text{ and } \mathcal{I}, l \models \psi \\ \mathcal{I}, l \models \texttt{all } x \texttt{:} e \texttt{|} \varphi & \text{iff } \mathcal{I}, l[x \mapsto d] \models \varphi \text{ for all } d \in evalE_l^{\mathcal{I}}(e) \end{array}$$

In the above, $evalE_l^{\mathcal{I}} : expr \to \mathcal{D}^{\mathcal{I}}$ is the evaluation function for expressions:

$$\begin{array}{rcl} evalE_l^{\mathcal{I}}(var) & = & l(var) \text{ for all } var \in \mathcal{V} \\ evalE_l^{\mathcal{I}}(name) & = & name^{\mathcal{I}} \text{ for all } name \in \mathcal{F}_{Name} \\ evalE_l^{\mathcal{I}}(\mathsf{op}(e_1, \ldots, e_n)) & = & \mathsf{op}^{\mathcal{I}}(evalE_l^{\mathcal{I}}(e_1), \ldots, evalE_l^{\mathcal{I}}(e_n)) \text{ for all } \mathsf{op} \in \mathcal{F}_{Op} \end{array}$$

Look familiar?

Soundness of Alloy's analysis

Soundness Alloy's consistency analysis can be said to be sound iff for every Alloy specification *Spec* with run predicate P, the instance $\mathcal{I}$ produced by Alloy's analysis in response to the command

```
run P() for n
```

for some scope n does indeed make all the constraints in *Spec* and P true, i.e.,

$$\mathcal{I} \models \varphi_{Spec} \wedge \varphi_P$$

Is Alloy's analysis sound?

Completeness of Alloy's analysis

Completeness Alloy's consistency analysis can be said to be complete iff for every instance $\mathcal{I}$ with

$$\mathcal{I} \models \varphi_{Spec} \wedge \varphi_P$$

for some specification *Spec* and predicate P Alloy's analysis in response to the command

```
run P() for maxSize
```

where *maxSize* is the size of the signature interpretation in $\mathcal{I}$ with the most elements, i.e.,

$$maxSize = max\{| \, s^{\mathcal{I}} \, | \mid s \in \mathcal{F}_{Sig}\}$$

will eventually (using the "Next instance" command) produce an isomorphic instance of $\mathcal{I}$. Similarly for assertion analysis.

Is Alloy's analysis complete?

Alloy Summary

- Language
  - Predicate Logic + Relational Calculus + Support for reuse, modularity
  - Declarative, property-oriented
- Analysis
  - 2 types of check
    - 1) consistency
    - 2) assertions

    } Bounded SAT solving
  - Tradeoff: expressiveness (of analysis) for decidability
  - `easy to use'
- Good match for
  - Object models, i.e., descriptions of collections of objects and their relationships and operations
- Unlikely to be a good match for
  - capturing constraints on, e.g., numerical data, performance, usability

WU1

COMMUNICATIONS OF THE ACM

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARCH | PRAC

Home / Magazine Archive / August 2017 (Vol. 60, No. 8) / In Praise of Under-Specification? / Full Text

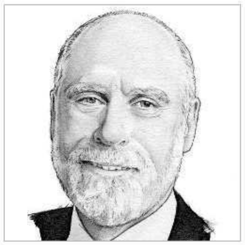
Capturing the Problem in Implementation-independent Terms: Specifications

CERF'S UP

In Praise of Under-Specification?

By Vinton G. Cerf
Communications of the ACM, August 2017, Vol. 60 No. 8, Page 7
10.1145/3110531
Comments (4)

VIEW AS: SHARE:



Eric Schmidt, executive chairman of Alphabet (Google's parent company), recently drew my attention to the notion of "under-specification." He reminded me that the Internet had benefited strongly from this concept. Several specific examples came to mind. The Internet Protocol (IP) specification does not contain any information about routing. It specifies what packets look like as they emerge from or arrive at the hosts at the edge of the Internet, but routing is entirely outside of that specification[a] partly because it was not entirely clear what procedures would be used for Internet routing at the time the specification was developed and, indeed, a number of them have been developed over time. There is nothing in the specification that describes the underlying transmission technology nor is there anything in the specification that speaks to how the packet's payload (a string of bits) is to be interpreted. These matters are open to instantiation independent of the specification of packet formats.

Slide 17

WU1 Windows User, 11/2/2018

Formal Specification

- Capture problem as abstractly as possible and as precisely as necessary
  - Specifications vs implementations
  - Declarative vs operational
  - Enable automatic analysis
    - Key tradeoff: expressiveness vs complexity
- Gain deeper understanding of problem and possible solutions (e.g., A4)