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

Juergen Dingel

Fall 2019

Lecture:

- Specification vs implementation
- The power and utility of formal specifications
- Intro to Alloy

What is a specification?

- American Heritage Dictionary:

> *"A detailed, exact statement of particulars, especially a statement prescribing materials, dimensions, and quality of work for something to be built, installed, or manufactured"*

- For software, e.g.,
  - **input/output behaviour** of a system, component, or method
  - a **class invariant**
  - the **description of interactions** necessary for the execution of a protocol
  - **structure and relationships** between objects

Desirable features of specifications

- As precise and detailed as necessary
- As abstract and unconstraining as possible
- Declarative rather than operational
  - what? vs how?
- Correct
- Consistent

Example: Specification of the Internet Protocol (IP)

DARPA. Internet Protocol Specification (RFC 791). Sept 1981.
Available at https://tools.ietf.org/html/rfc791

V.G. Cerf. In praise of under-specification? CACM 60(8):7-7. Aug 2017.
Available at https://dl.acm.org/citation.cfm?id=3110531

Specifications vs implementations

- Specifications
  - *"as abstract as possible, as concrete as necessary"*
  - "declarative" rather than "operational"

  => may not be executable
- Implementation
  - executable
- The promise of Prolog

```
member(X, [X|_]).
member(X, [Y|Ys]) :- X=/=Y, member(X, Ys).
```

Specification languages

1. Non-formal: Natural language

- Pros
  - expressive
  - no/little training required
- Cons
  - often imprecise

> *"Aircraft that are non-friendly and have an unknown mission or the potential to enter restricted air-space within 5 minutes shall …"*

  - limited opportunity for (automated) analysis due to its complexity (e.g., implicit context knowledge)

The (sometimes hidden) complexity of natural language

- E.g., informal descriptions of requirements may implicitly assume **context knowledge**:

Shoes must be worn!

Dogs must be carried!

What is the problem?

[M. Jackson. *Software Specifications and Requirements: a lexicon of practice, principles and prejudices*. Addison-Wesley, 1995.]

The (sometimes hidden) complexity of natural language

- Informal descriptions of requirements may implicitly assume **context knowledge**:

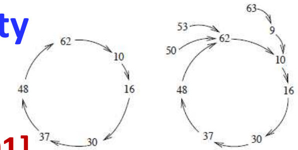
Shoes must be worn! → $\forall x : Person.\ \exists y : Shoes.\ Owns(x,y) \wedge Wears(x,y)$

Dogs must be carried! → $\forall x : Person.\ \exists y : Dog.\ Owns(x,y) \wedge Carries(x,y)$ ☹

→ $\forall x : Person.\ \forall y : Dog.\ Owns(x,y) \rightarrow Carries(x,y)$ ☺

Analyzing informal models
in a meaningful way typically impossible

The (sometimes hidden) complexity of software



Chord: Distributed hash table [Chord01]

[Chord01] Stoica, Morris, Karger, Kaashoek, Balakrishnan. *"Chord: A scalable peer-to-peer lookup service for Internet applications"*. SIGCOMM. 2001.

- *"3 features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance"*
- Papers present properties, invariants and proofs
- 4th most-cited paper in CS for years (CiteSeer)
- 2011 SIGCOMM Test-of-Time Award

> *"Unfortunately, the claim of correctness is not true. The original specification […] does not have eventual reachability, and not one of the seven properties claimed to be invariants […] is actually an invariant."*
>
> *"For complex protocols such as Chord, there is every reason to use lightweight modeling as a design and documentation tool"*
>
> P. Zave. Various papers on **http://www.research.att.com/~pamela/chord.html**

But, once a problem is formalized amazing things are possible

- Impress your friends by solving every Sudoku puzzle

```
 , , | , , | ,1, ,
4, , | , , | , , ,
 ,2, | , , | , , ,
------------------
 , , | ,5, |4, , ,
 , ,8| , , |3, , ,
 , ,1| ,9, | , , ,
------------------
3, , |4, , |2, , ,
 ,5, |1, , | , , ,
 , , |8, ,6| , , ,
```

www.spass-prover.org

```
6,9,3|7,8,4|5,1,2,
4,8,7|5,1,2|9,3,6,
1,2,5|9,6,3|8,7,4,
------------------
9,3,2|6,5,1|4,8,7,
5,6,8|2,4,7|3,9,1,
7,4,1|3,9,8|6,2,5,
------------------
3,1,9|4,7,5|2,6,8,
8,5,6|1,2,9|7,4,3,
2,7,4|8,3,6|1,5,9,
```

?

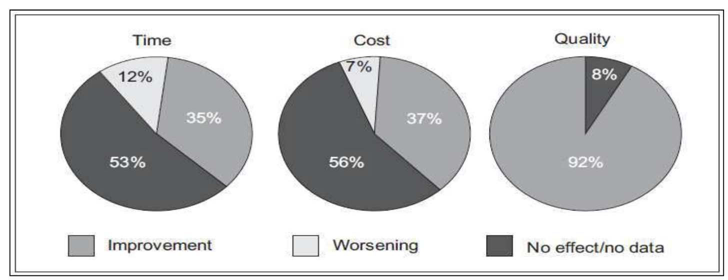
Note: This is not a toy!
More than 10^{38} possibilities, i.e., size of state space > 10^{38}

Number of cells in human body: 10^{13}
Number of atoms in universe: 10^{80}

Wow!

But, once a problem is formalized amazing things are possible (cont'd)

- Survey of 62 int'l FM projects
  - **Domains:** Real-time, distributed & parallel, transaction processing, high-data volume, control, services

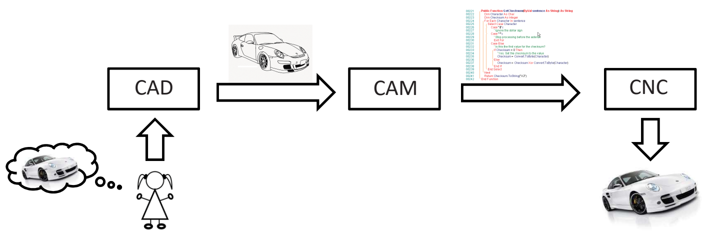


[Radio Technical Commission for Aeronautics (RTCA). DO-333: Formal Methods Supplement to DO-178C and DO-278A.
[Woodcock et al. Formal Methods: Practice and Experience. ACM Computing Surveys 41(4). 2009]

But, once a problem is formalized amazing things are possible (cont'd)

Mechanical design from about 1972: CAD/CAM

1. Create drawings w/ computer (CAD)
2. From drawing, computer automatically generates program to drive milling and CNC machines (CAM)

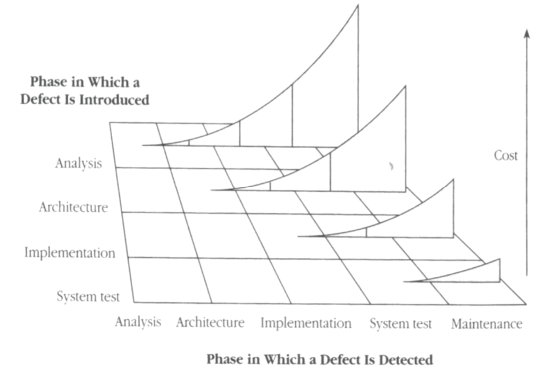


=> much better analysis capabilities and productivity

=> CAD/CAM has revolutionized manufacturing

Analysis of specifications

- Analysis of specifications for correctness, consistency, (un)desirable properties can pay off



Specification languages

1. Non-formal
- Natural language

2. Semi-formal
- UML

3. Formal
- precisely defined semantics
- mechanisms for abstraction, analysis, modularity, reuse
- Used for
  - safety-critical systems, but
  - not necessarily (e.g., state machines)
- Examples:
  - 1) Propositional and Predicate logic, 2) Alloy,
  - Z, B, VDM, …

Formal specification languages

1. Propositional and/or predicate logic

add: $(String \times Object \times \wp(String \times Object)) \rightarrow \wp(String \times Object)$ *such that*

$$\forall d : \mathcal{P}(String \times Object).\, \forall d' : \mathcal{P}(String \times Object).\, \forall key : String.\, \forall val : Object.$$
$$d' = add(key, val, d) \quad \leftrightarrow$$
$$\big((\neg\, \exists v : Object. \langle key, v\rangle \in d) \quad \rightarrow \quad d' = d \cup \{\langle key, val\rangle\}\big) \wedge$$
$$\big(\exists v : Object. \langle key, v\rangle \in d. \quad \rightarrow \quad d' = d - \langle key, v\rangle \cup \langle key, val\rangle\big)$$

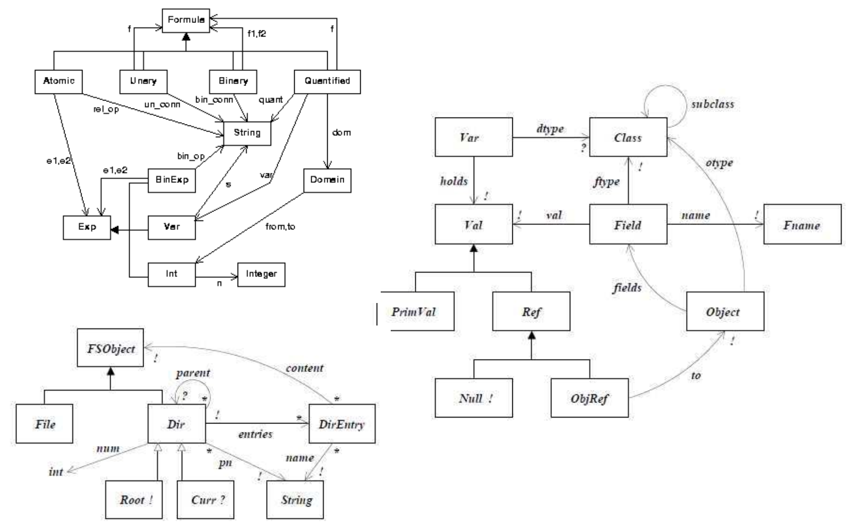
- **Pros**
  - expressive, well-studied, formal, good tool and analysis support
- **Cons**
  - lack of modularity mechanisms
  - predicate logic is undecidable

2. Alloy

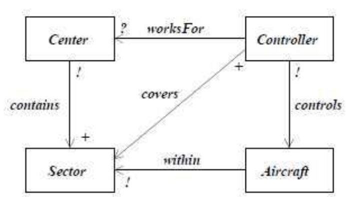
Alloy: What for?

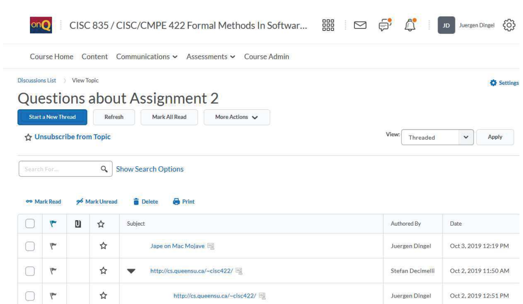
1. Formal approach to describing structure and relationships between objects (“class modeling”, “object modeling”)

Class models: For SW concepts



Class models: For non-SW concepts





Alloy: What for?

1. Formal approach to describing structure and relationships between objects

But, why not use UML (Class Diagrams & Object Diagrams)?

2. Analyze specifications automatically with respect to
   1. Correctness
   2. Consistency
   3. (Un-)desirable properties

Alloy: core ingredients

Alloy, the language:

- Declarative
- First-order logic + relational calculus
- *“Everything is a relation!”*

Alloy, the analysis:

- Automatic
- Satisfiability solving (SAT)

Alloy, the tool:

- Stable, usable, “light-weight”

Less is More

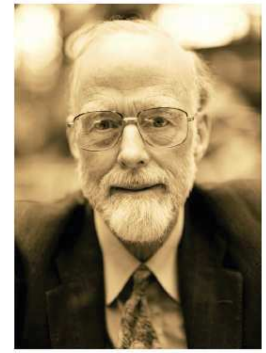
If Done Right

why declarative design?

I conclude there are two ways of constructing a software design.

One way is to make it so simple there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

– Tony Hoare [Turing Award Lecture, 1980]

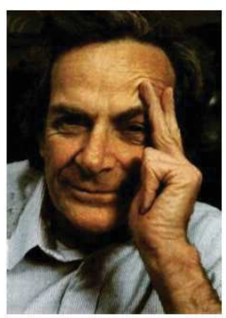


From http://alloy.mit.edu/alloy/tutorials/day-course/

why automated analysis?

The first principle is that you must not fool yourself, and you are the easiest person to fool.

– Richard P. Feynman



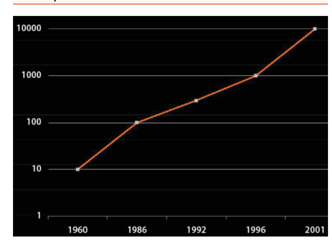
From http://alloy.mit.edu/alloy/tutorials/day-course/

Why SAT?

The Complexity of Theorem-Proving Procedures
Stephen A. Cook
University of Toronto

- Quintessential hard problem
  - First problem to be proven NP-complete [Cook 1971]
  - Lots of other common problems can be solved using SAT
- Hard, but not impossible
  - Heuristical SAT-solvers solve problems w/ ~10K variables, enough to deal w/ many practical problems
    - HW verification
      - E.g., circuit for z=x/y where x,y,z are 128-bit floats: 2^{256} combinations
      - Non-solution: manual
      - Solution: random-constraint test gen.
    - SW verification
    - Planning, scheduling

SAT performance



logic: everything's a relation

- sets are unary (1 column) relations

```
Name = {(N0),       Addr = {(A0),       Book = {(B0),
        (N1),               (A1),               (B1)}
        (N2)}               (A2)}
```

- scalars are singleton sets

```
myName   = {(N1)}
yourName = {(N2)}
myBook   = {(B0)}
```

- binary relation

```
names = {(B0, N0),
         (B0, N1),
         (B1, N2)}
```

- ternary relation

```
addrs = {(B0, N0, A0),
         (B0, N1, A1),
         (B1, N1, A2),
         (B1, N2, A2)}
```


logic: relations

```
addrs = {(B0, N0, A0), (B0, N1, A1),
         (B1, N1, A2), (B1, N2, A2)}
```

B0	**N0**	**A0**
B0	**N1**	**A1**
B1	**N1**	**A2**
B1	**N2**	**A2**

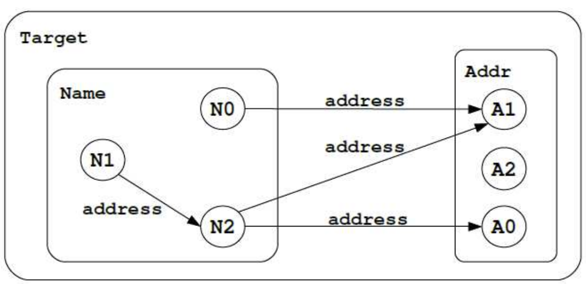
size = 4

arity = 3

- rows are unordered
- columns are ordered but unnamed
- all relations are first-order
  - relations cannot contain relations, no sets of sets

logic: address book example

```
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1), (A2)}
Target = {(N0), (N1), (N2), (A0), (A1), (A2)}
address = {(N0, A1), (N1, N2), (N2, A1), (N2, A0)}
```



logic: set operators

+	*union*
&	*intersection*
-	*difference*
in	*subset*
=	*equality*

```
Name  = {(N0), (N1), (N2)}
Alias = {(N1), (N2)}
Group = {(N0)}
RecentlyUsed = {(N0), (N2)}

Alias + Group = {(N0), (N1), (N2)}
Alias & RecentlyUsed  = {(N2)}
Name - RecentlyUsed   = {(N1)}
RecentlyUsed in Alias = false
RecentlyUsed in Name  = true
Name = Group + Alias  = true
```

```
greg = {(N0)}
rob = {(N1)}

greg + rob   = {(N0), (N1)}
greg = rob   = false
rob in none  = false
```

```
cacheAddr = {(N0, A0), (N1, A1)}
diskAddr = {(N0, A0), (N1, A2)}

cacheAddr + diskAddr =
cacheAddr & diskAddr =
cacheAddr = diskAddr =
```

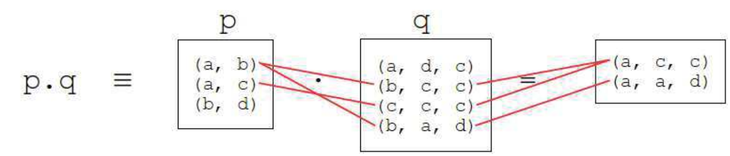
logic: product operator

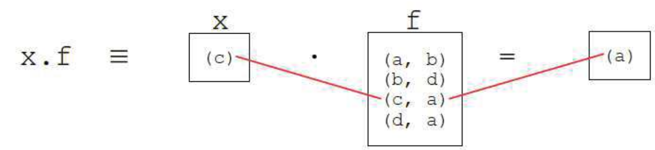
-> *cross product*

```
Name = {(N0), (N1)}
Addr = {(A0), (A1)}
Book = {(B0)}

Name->Addr = {(N0, A0), (N0, A1),
              (N1, A0), (N1, A1)}
Book->Name->Addr =
  {(B0, N0, A0), (B0, N0, A1),
   (B0, N1, A0), (B0, N1, A1)}
```

```
b  = {(B0)}
b' = {(B1)}
address  = {(N0, A0), (N1, A1)}
address' = {(N2, A2)}

b->b' = [redacted]

b->address + b'->address' =
  [redacted]
```


logic: relational join





CISC/CMPE 422/835, (Formal) specifications and Alloy intro From http://alloy.mit.edu/tutorials/day-course/

logic: unary operators

~	*transpose*
^	*transitive closure*
*	*reflexive transitive closure*

apply only to binary relations

```
^r = r + r.r + r.r.r + …
*r = iden + ^r
```

```
Node = {(N0), (N1), (N2), (N3)}
next = {(N0, N1), (N1, N2), (N2, N3)}

~next = {(N1, N0), (N2, N1), (N3, N2)}
^next = {(N0, N1), (N0, N2), (N0, N3),
         (N1, N2), (N1, N3),
         (N2, N3)}
*next = {(N0, N0), (N0, N1), (N0, N2), (N0, N3),
         (N1, N1), (N1, N2), (N1, N3),
         (N2, N2), (N2, N3), (N3, N3)}
```

```
first = {(N0)}
rest = {(N1), (N2), (N3)}

first.^next = rest
first.*next = Node
```

CISC/CMPE 422/835, (Formal) specifications and Alloy intro From http://alloy.mit.edu/tutorials/day-course/

logic: boolean operators

!	**not**	*negation*
&&	**and**	*conjunction*
||	**or**	*disjunction*
=>	**implies**	*implication*
	else	*alternative*
<=>	**iff**	*bi-implication*

four equivalent constraints:

F => G **else** H

F **implies** G **else** H

(F && G) || ((!F) && H)

(F **and** G) **or** ((**not** F) **and** H)

CISC/CMPE 422/835, (Formal) specifications and Alloy intro From http://alloy.mit.edu/tutorials/day-course/

logic: quantifiers

```
all x: e | F
all x: e1, y: e2 | F
all x, y: e | F
all disj x, y: e | F
```

all	*F holds for **every** x in e*
some	*F holds for **at least one** x in e*
no	*F holds for **no** x in e*
lone	*F holds for **at most one** x in e*
one	*F holds for **exactly one** x in e*

```
some n: Name, a: Address | a in n.address
```
some name maps to some address — address book not empty

```
no n: Name | n in n.^address
```

```
all n: Name | lone a: Address | a in n.address
```

```
all n: Name | no disj a, a': Address | (a + a') in n.address
```


logic: set declarations

```
x: m e
```

```
Q x: m e
```

```
x: e <=> x: one e
```

set	*any number*
one	*exactly one*
lone	*zero or one*
some	*one or more*

```
RecentlyUsed: set Name
```
RecentlyUsed is a subset of the set Name

```
senderAddress: Addr
```
senderAddress is a singleton subset of Addr

```
senderName: lone Name
```
senderName is either empty or a singleton subset of Name

```
receiverAddresses: some Addr
```
receiverAddresses is a nonempty subset of Addr

logic: relation declarations

```
r: A m -> n B
Q r: A m -> n B
```

```
r: A -> B <=>
r: A set -> set B
```

```
(r: A m -> n B) <=>
   ((all a: A | n a.r) and (all b: B | m r.b))
```

```
workAddress: Name -> lone Addr
```
each alias refers to at most one work address

```
homeAddress: Name -> one Addr
```
each alias refers to exactly one home address

```
members: Name lone -> some Addr
```
address belongs to at most one group name and group contains at least one address

```
r: A -> (B m -> n C) <=>
all a: A | a.r: B m -> n C
```

```
r: (A m -> n B) -> C <=>
all c: C | r.c: A m -> n B
```