CSE503: SOFTWARE ENGINEERING

ELECTRIFYING SPECIFICATIONS

David Notkin
Spring 2011

Formal methods

- We've covered some of the basics, although there is much more out there – we'll get to some of that
- One thing is clear so far: programs *execute* – they are kinetic – while formalisms just sit there – they represent potential
- This property makes many formalisms less attractive to many people, as the benefits are harder to see

"It is easier to change the specification to fit the program than vice versa." –**Perlis**

Electrifying formalisms

- Daniel Jackson (and others) have worked on addressing this concern by "electrifying" formalisms – that is, making them "executable" in some sense, or at least providing useful feedback to a developer quickly
- Alloy is Jackson's core approach to this, but it's not the only one out there

Executable specifications

- One way to electrify a formalism is to execute it – many formalisms represent high-level programs
  - Google Scholar found ~95K entries to "executable specifications"
  - Many such executable specifications look a lot like (various kinds of) logic programs or functional programs; much of this work is related to *automatic programming*
- The execution gives insight into what the specification means
- Performance of these "programs" is usually poor
- And automated refinement techniques to evolve from an executable specification to an efficient program seem to be limited
  - This work goes back to at least 1976 with Darlington, Burstall, Manna and others

Type checking

5

- Type checking is a good example of an electrified formalism – based on the type system and proof rules, the compiler reports immediately on situations where the program may be misusing typed values
- This is also a good example of an alternative style of electrification
- Rather than executing to "see what it does," it rather compares two different views of the computation – what the types constrain and whether the program satisfies those constraints

Comparing

6

- I believe that this comparative approach is extremely powerful – more powerful than the "let's make higher-level specifications that we can execute" in the long-run
- In some sense, that was the idea of proving desirable properties from axioms – the axioms described relationships of an ADT, and the posited properties were alternative views of what should be true for that ADT
  - But this wasn't electrified in any sense

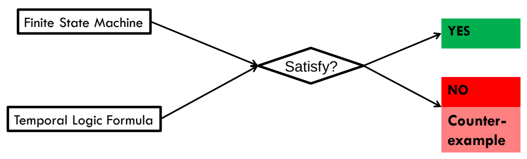
Model checking

7

- Model checking is one of the bases for electrifying comparisons of program views
- We'll look at this first from the high-level notion of model checking, then look at Alloy as an instance of electrified (bounded) model checking, and then look back at more conventional model checking

Model checking

8

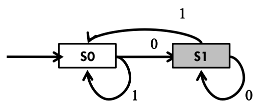


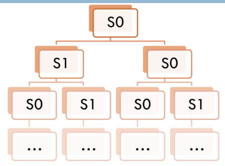
- What can the finite state machines and temporal logic formulae represent?
- What does "satisfy" mean? How does "satisfy" work?
- Why should we care?
- What is a counterexample?
- What does "Yes" actually mean?

ACM 2007 Turing Award Citation

In 1981, Edmund M. Clarke and E. Allen Emerson, working in the USA, and Joseph Sifakis working independently in France, authored seminal papers that founded what has become the highly successful field of Model Checking. This verification technology provides an algorithmic means of determining whether an abstract model--representing, for example, a hardware or software design--satisfies a formal specification expressed as a temporal logic formula. Moreover, if the property does not hold, the method identifies a counterexample execution that shows the source of the problem. The progression of Model Checking to the point where it can be successfully used for complex systems has required the development of sophisticated means of coping with what is known as the state explosion problem. Great strides have been made on this problem over the past 27 years by what is now a very large international research community. As a result many major hardware and software companies are now using Model Checking in practice. Examples of its use include the verification of VLSI circuits, communication protocols, software device drivers, real-time embedded systems, and security algorithms. …

A computation tree over an FSM





- Represent all possible FSM paths with a computation tree – even when infinite, the structure is constrained because of finite states
- The computation tree is generated from the state machine
- The temporal logic formula queries the computation tree

- Model checking answers questions about this tree structure – kinds of queries include
  - Does every accepting input include a **0**? A **1**?
  - Does any accepting input include a **0**? A **1**?
  - Does every accepting input that has a **1** have a **1** in the remaining input?
- More generally, safety and liveness properties of many kinds

FAQ

- What can the finite state machines and temporal logic formulae represent?
  - See the Turing Award citation (and more…)
- What does “satisfy” mean? How does “satisfy” work?
  - Satisfy means that the temporal logic formula is guaranteed to hold over the computation tree defined by the FSM
- Why should we care?
  - Guarantees can be a good thing
- What is a counterexample?
  - A path through the computation tree that contradicts the temporal logic formula
  - There is a mismatch between the two descriptions – but one cannot tell which is “wrong” without further work
- What does “Yes” actually mean?
  - It is a guarantee that the property holds, but it provides no guarantee that the property or the FSM are what the developers thinks they are

Two Approaches to Model Checking

- Explicit – represent all states
  - Use conventional state-space search
  - Reduce state space by folding equivalent states together
- Symbolic – represent sets of states using boolean formulae
  - Reduce huge state spaces by considering large sets of states simultaneously – to the first order, this is the meeting of BDDs (binary decision diagrams) and model checking (more later)
  - Convert state machines, logic formulae, etc. to boolean representations
  - Perform state space exploration using boolean operators to perform set operations
  - SAT solvers are often at the base of symbolic model checking

Bounded model checking

- Restricting to finite abstractions work in many situations – protocols, etc.
- But programs themselves are not generally restricted to finite abstractions – so how could model checking apply?
- Simply, one can constrain the search space: exhaustively search within a finite space with the expectation (hope?) that the finite space is a good approximation to the infinite space
- D. Jackson calls this the “small scope hypothesis”

Basic claim and idea

- Most counterexamples can be identified with a failing trace that is short, by some definition
  - The number of operations
  - The complexity of the input structures (e.g., how many nodes)
  - Size of a device
  - Length of a pathname
  - …
- If no counterexample is returned when checking the truncated space, one builds confidence in the model but cannot be certain of the properties
- Eliminating counterexamples likely helps through the infinite space

A little pre-Alloy history

- Model-based descriptions
- The Z specification language

Model-oriented

- Model a system by describing its state together with operations over that state
  - An operation is a function that maps a value of the state together with values of parameters to the operation onto a new state value
- A model-oriented language typically describes mathematical objects (e.g. data structures or functions) that are structurally similar to the required computer software

Z ("zed")

17

- Perhaps the most widely known and used model-based specification language
- Good for describing state-based abstract descriptions roughly in the abstract data type style
- Based on typed set theory and predicate logic
- A few commercial and honorary successes
  - I'll come back to one reengineering story afterwards

Basics

18

- Static schemas
  - States a system can occupy
  - Invariants that must be maintained in every system state
- Dynamic schemas
  - Operations that are permitted
  - Relationship between inputs and outputs of those operations
  - Changes of states

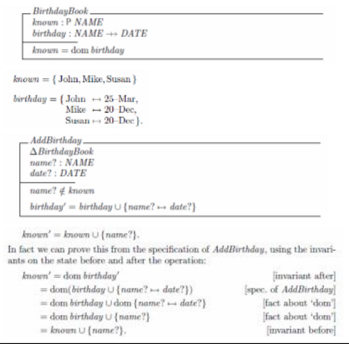
The classic example

19

- A "birthday book" that tracks people's birthdays and can issue reminders of those birthdays
  - There are tons of web-based versions of these now
- There are two basic types of atomic elements in this example
  - [NAME,DATE]
  - An inherent degree of abstraction: nothing about formats, possible values, etc.

Taken directly from Spivey's ZRM

20



BirthdayBook

$known : \mathbb{P}\ NAME$

$birthday : NAME \rightarrowtail DATE$

$known = \mathrm{dom}\ birthday$

$$known = \{ \text{John}, \text{Mike}, \text{Susan} \}$$

$$birthday = \{ \text{John} \mapsto \text{25–Mar}, \text{Mike} \mapsto \text{20–Dec}, \text{Susan} \mapsto \text{20–Dec} \}.$$

AddBirthday

$\Delta BirthdayBook$

$name? : NAME$

$date? : DATE$

$name? \notin known$

$birthday' = birthday \cup \{name? \mapsto date?\}$

$$known' = known \cup \{name?\}.$$

In fact we can prove this from the specification of *AddBirthday*, using the invariants on the state before and after the operation:

$$\begin{aligned} known' &= \mathrm{dom}\ birthday' && \text{[invariant after]} \\ &= \mathrm{dom}(birthday \cup \{name? \mapsto date?\}) && \text{[spec. of } AddBirthday\text{]} \\ &= \mathrm{dom}\ birthday \cup \mathrm{dom}\ \{name? \mapsto date?\} && \text{[fact about `dom']} \\ &= \mathrm{dom}\ birthday \cup \{name?\} && \text{[fact about `dom']} \\ &= known \cup \{name?\}. && \text{[invariant before]} \end{aligned}$$

static schema

legal system state

dynamic schema

Proof of expected property

Some more schemas

21

```
FindBirthday
ΞBirthdayBook
name? : NAME
date! : DATE
---
name? ∈ known
date! = birthday(name?)
```

schema w/ unchanged inputs

```
Remind
ΞBirthdayBook
today? : DATE
cards! : ℙ NAME
---
cards! = { n : known | birthday(n) = today? }
```

schema w/ unchanged inputs

```
InitBirthdayBook
BirthdayBook
---
known = ∅
```

initializing schema

What about errors?

22

- Entering a birthday for someone already in the book?
- Looking up a birthday for an unknown person?
- What is the meaning of the specification in the face of errors like these – unsatisfied pre-conditions, in most situations?
- Rewrite the whole specification?

Or use the schema calculus!

23

REPORT ::= *ok* | *already_known* | *not_known*.

add an enumerated type

```
Success
result! : REPORT
---
result! = ok
```

schema

AddBirthday ∧ *Success*.

conjoin the schemas

```
AlreadyKnown
ΞBirthdayBook
name? : NAME
result! : REPORT
---
name? ∈ known
result! = already_known
```

schema

RAddBirthday ≙ (*AddBirthday* ∧ *Success*) ∨ *AlreadyKnown*.

schema calculus

Schema calculus: sweet!

24

- The schema calculus allows us to combine specifications using logical operators (e.g., ∧, ∨, ⇒, ¬)
  - This allows us to define the common and error cases separately, for example, and then just ∧-ing them together
- In some sense, it allows us to get a cleaner, smaller specification

But don't try this on programs!

- Wouldn't it be fantastic if we had the equivalent of the schema calculus on programs?
  - Write your error cases separately and then just $\wedge$ them together
  - Write a text editor and a spell checker and integrate them by $\wedge$-ing them together
  - So you want to build a program that doesn't blow up a nuclear power plant?
    - Just build one that does, and then negate it ☺!
- Programs are not logic
  - Some classes of programming languages – largely logic and functional languages – come closer than imperative and OO languages, but they are not logic

Points about Z

- This isn't proving correctness between a specification and a program
  - There isn't a program!
- Even the specification without the implementation has (some) value
  - The most obvious example is when a theorem is posited and then is proven from the rest of the specification – albeit by hand
- The actual notation seems more effective that some others
  - Z is intended to be in bite-sized chucks (schema), interspersed with natural language explanations

Z used to improve CICS/ESA_V3.1

- A broadly used IBM transaction processing system
- Integrated into IBM's existing and well-established development process
- Many measurements of the process indicated that they were able to reduce their costs for the development by almost five and a half million dollars
- Early results from customers also indicated significantly fewer problems, and those that have been detected are less severe than would be expected otherwise

Queen's Award for Technological Achievement

- "Her Majesty the Queen has been graciously pleased to approve the Prime Minister's recommendation that The Queen's Award for Technological Achievement should be conferred this year upon Oxford University Computing Laboratory.
- "Oxford University Computing Laboratory gains the Award jointly with IBM United Kingdom Laboratories Limited for the development of a programming method based on elementary set theory and logic known as the Z notation, and its application in the IBM Customer Information Control System (CICS) product. …
- "The use of Z reduced development costs significantly and improved reliability and quality. Precision is achieved by basing the notation on mathematics, abstraction through data refinement, re-use through modularity and accuracy through the techniques of proof and derivation.
- "CICS is used worldwide by banks, insurance companies, finance houses and airlines etc. who rely on the integrity of the system for their day-to-day business."

Sir Charles Antony Richard Hoare

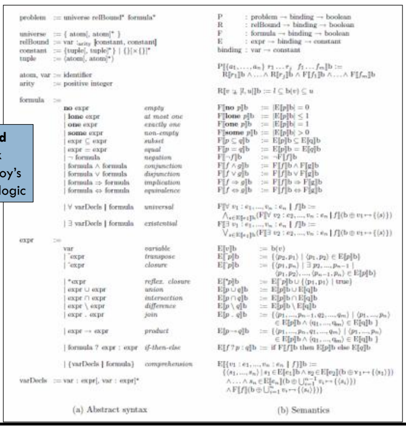
Z to Alloy: an electrifying path…

- At some reasonable level, Alloy is Z with an analysis engine to checked bounded spaces
  - Up to five friends, up to five birthdays, etc.
- Alloy returns either true or else a counterexample within that bounded state space
  - Again, recall the small scope hypothesis – in Alloy's words, the analysis is sound and "complete up to scope"
- Alloy has been designed explicitly to support automatic analysis
- Alloy supports complex data structures, such as trees, and thus is a rich way to describe state

Underneath: SAT solving

- Since Nitpick, Alloy's predecessor, Jackson's analysis engines have been based on broad set of SAT-solvers
- They have techniques and optimizations for translating from relational to boolean logic
- They are currently using their own SAT-solver, Kodkod
  - first order logic with relations, transitive closure, and partial models
  - analyses for both satisfiable and unsatisfiable problems; much of this is based on the ability to use partial models

From Torlak's **kodkod** dissertation – syntax and semantics of Alloy's bounded relational logic



```
problem   := universe relBound* formula*

universe  := { atom[, atom]* }
relBound  := var :arity [constant, constant]
constant  := {tuple[, tuple]* } | {}[×{}]*
tuple     := ⟨atom[, atom]*⟩

atom, var := identifier
arity     := positive integer

formula   :=
             no expr                      empty
             | lone expr                  at most one
             | one expr                   exactly one
             | some expr                  non-empty
             | expr ⊆ expr                subset
             | expr = expr                equal
             | ¬ formula                  negation
             | formula ∧ formula          conjunction
             | formula ∨ formula          disjunction
             | formula ⇒ formula          implication
             | formula ⇔ formula          equivalence

             | ∀ varDecls | formula       universal

             | ∃ varDecls | formula       existential

expr      :=
             var                          variable
             | ~expr                      transpose
             | ^expr                      closure

             | *expr                      reflex. closure
             | expr ∪ expr                union
             | expr ∩ expr                intersection
             | expr \ expr                difference
             | expr . expr                join

             | expr → expr                product

             | formula ? expr : expr      if-then-else

             | {varDecls | formula}       comprehension

varDecls  := var : expr[, var : expr]*
```

(a) Abstract syntax

$$
\begin{array}{ll}
\mathrm{P} & : \text{problem} \to \text{binding} \to \text{boolean} \\
\mathrm{R} & : \text{relBound} \to \text{binding} \to \text{boolean} \\
\mathrm{F} & : \text{formula} \to \text{binding} \to \text{boolean} \\
\mathrm{E} & : \text{expr} \to \text{binding} \to \text{constant} \\
\text{binding} & : \text{var} \to \text{constant}
\end{array}
$$

$$\mathrm{P}[\![\{a_1,\ldots,a_n\}\ r_1\ldots r_j\ f_1\ldots f_m]\!]\mathrm{b} := \mathrm{R}[\![r_1]\!]\mathrm{b} \wedge \ldots \wedge \mathrm{R}[\![r_j]\!]\mathrm{b} \wedge \mathrm{F}[\![f_1]\!]\mathrm{b} \wedge \ldots \wedge \mathrm{F}[\![f_m]\!]\mathrm{b}$$

$$\mathrm{R}[\![v :_k [l, u]]\!]\mathrm{b} := l \subseteq \mathrm{b}(v) \subseteq u$$

$$
\begin{array}{lcl}
\mathrm{F}[\![\textbf{no}\ p]\!]\mathrm{b} & := & |\mathrm{E}[\![p]\!]\mathrm{b}| = 0 \\
\mathrm{F}[\![\textbf{lone}\ p]\!]\mathrm{b} & := & |\mathrm{E}[\![p]\!]\mathrm{b}| \leq 1 \\
\mathrm{F}[\![\textbf{one}\ p]\!]\mathrm{b} & := & |\mathrm{E}[\![p]\!]\mathrm{b}| = 1 \\
\mathrm{F}[\![\textbf{some}\ p]\!]\mathrm{b} & := & |\mathrm{E}[\![p]\!]\mathrm{b}| > 0 \\
\mathrm{F}[\![p \subseteq q]\!]\mathrm{b} & := & \mathrm{E}[\![p]\!]\mathrm{b} \subseteq \mathrm{E}[\![q]\!]\mathrm{b} \\
\mathrm{F}[\![p = q]\!]\mathrm{b} & := & \mathrm{E}[\![p]\!]\mathrm{b} = \mathrm{E}[\![q]\!]\mathrm{b} \\
\mathrm{F}[\![\neg f]\!]\mathrm{b} & := & \neg\mathrm{F}[\![f]\!]\mathrm{b} \\
\mathrm{F}[\![f \wedge g]\!]\mathrm{b} & := & \mathrm{F}[\![f]\!]\mathrm{b} \wedge \mathrm{F}[\![g]\!]\mathrm{b} \\
\mathrm{F}[\![f \vee g]\!]\mathrm{b} & := & \mathrm{F}[\![f]\!]\mathrm{b} \vee \mathrm{F}[\![g]\!]\mathrm{b} \\
\mathrm{F}[\![f \Rightarrow g]\!]\mathrm{b} & := & \mathrm{F}[\![f]\!]\mathrm{b} \Rightarrow \mathrm{F}[\![g]\!]\mathrm{b} \\
\mathrm{F}[\![f \Leftrightarrow g]\!]\mathrm{b} & := & \mathrm{F}[\![f]\!]\mathrm{b} \Leftrightarrow \mathrm{F}[\![g]\!]\mathrm{b}
\end{array}
$$

$$\mathrm{F}[\![\forall\, v_1 : e_1,\ldots,v_n : e_n \mid f]\!]\mathrm{b} := \bigwedge_{s \in \mathrm{E}[\![e_1]\!]\mathrm{b}} (\mathrm{F}[\![\forall\, v_2 : e_2,\ldots,v_n : e_n \mid f]\!](\mathrm{b} \oplus v_1 \mapsto \{\langle s\rangle\}))$$

$$\mathrm{F}[\![\exists\, v_1 : e_1,\ldots,v_n : e_n \mid f]\!]\mathrm{b} := \bigvee_{s \in \mathrm{E}[\![e_1]\!]\mathrm{b}} (\mathrm{F}[\![\exists\, v_2 : e_2,\ldots,v_n : e_n \mid f]\!](\mathrm{b} \oplus v_1 \mapsto \{\langle s\rangle\}))$$

$$
\begin{array}{lcl}
\mathrm{E}[\![v]\!]\mathrm{b} & := & \mathrm{b}(v) \\
\mathrm{E}[\![\sim p]\!]\mathrm{b} & := & \{\langle p_2, p_1\rangle \mid \langle p_1, p_2\rangle \in \mathrm{E}[\![p]\!]\mathrm{b}\} \\
\mathrm{E}[\![\hat{}\,p]\!]\mathrm{b} & := & \{\langle p_1, p_n\rangle \mid \exists\, p_2,\ldots,p_{n-1} \mid \\
 & & \quad \langle p_1, p_2\rangle,\ldots,\langle p_{n-1}, p_n\rangle \in \mathrm{E}[\![p]\!]\mathrm{b}\} \\
\mathrm{E}[\![{*}p]\!]\mathrm{b} & := & \mathrm{E}[\![\hat{}\,p]\!]\mathrm{b} \cup \{\langle p_1, p_1\rangle \mid \text{true}\} \\
\mathrm{E}[\![p \cup q]\!]\mathrm{b} & := & \mathrm{E}[\![p]\!]\mathrm{b} \cup \mathrm{E}[\![q]\!]\mathrm{b} \\
\mathrm{E}[\![p \cap q]\!]\mathrm{b} & := & \mathrm{E}[\![p]\!]\mathrm{b} \cap \mathrm{E}[\![q]\!]\mathrm{b} \\
\mathrm{E}[\![p \setminus q]\!]\mathrm{b} & := & \mathrm{E}[\![p]\!]\mathrm{b} \setminus \mathrm{E}[\![q]\!]\mathrm{b} \\
\mathrm{E}[\![p\,.\,q]\!]\mathrm{b} & := & \{\langle p_1,\ldots,p_{n-1}, q_2,\ldots,q_m\rangle \mid \langle p_1,\ldots,p_n\rangle \\
 & & \quad \in \mathrm{E}[\![p]\!]\mathrm{b} \wedge \langle q_1,\ldots,q_m\rangle \in \mathrm{E}[\![q]\!]\mathrm{b}\} \\
\mathrm{E}[\![p \to q]\!]\mathrm{b} & := & \{\langle p_1,\ldots,p_n, q_1,\ldots,q_m\rangle \mid \langle p_1,\ldots,p_n\rangle \\
 & & \quad \in \mathrm{E}[\![p]\!]\mathrm{b} \wedge \langle q_1,\ldots,q_m\rangle \in \mathrm{E}[\![q]\!]\mathrm{b}\} \\
\mathrm{E}[\![f\ ?\ p : q]\!]\mathrm{b} & := & \text{if } \mathrm{F}[\![f]\!]\mathrm{b} \text{ then } \mathrm{E}[\![p]\!]\mathrm{b} \text{ else } \mathrm{E}[\![q]\!]\mathrm{b}
\end{array}
$$

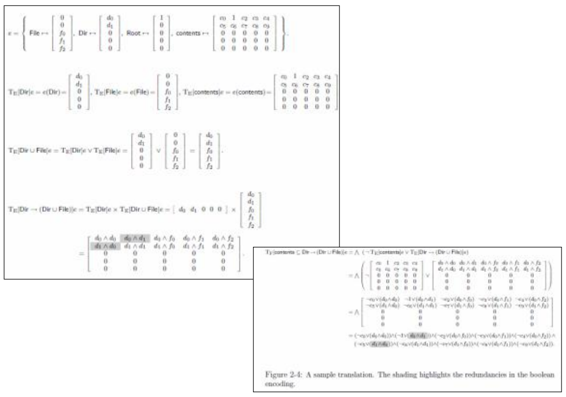
$$
\begin{array}{l}
\mathrm{E}[\![\{v_1 : e_1,\ldots,v_n : e_n \mid f\}]\!]\mathrm{b} := \\
\quad \{\langle s_1,\ldots,s_n\rangle \mid s_1 \in \mathrm{E}[\![e_1]\!]\mathrm{b} \wedge s_2 \in \mathrm{E}[\![e_2]\!](\mathrm{b} \oplus v_1 \mapsto \{\langle s_1\rangle\}) \\
\quad\ \wedge \ldots \wedge s_n \in \mathrm{E}[\![e_n]\!](\mathrm{b} \oplus \bigcup_{i=1}^{n-1} v_i \mapsto \{\langle s_i\rangle\}) \\
\quad\ \wedge \mathrm{F}[\![f]\!](\mathrm{b} \oplus \bigcup_{i=1}^{n} v_i \mapsto \{\langle s_i\rangle\})\}
\end{array}
$$

(b) Semantics

Translating to SAT

- Translation to boolean logic
- Break symmetries, handle sparseness, etc.
- Normalize to CNF
- Mapping back to original model

- Basic notion: a relation over a finite university can be represented as a matrix of boolean values
  - Relational constraints are represented as boolean constraints over the matrices



$$e = \left\{ \text{File} \mapsto \begin{bmatrix} 0 \\ 0 \\ f_0 \\ f_1 \\ f_2 \end{bmatrix}, \text{Dir} \mapsto \begin{bmatrix} d_0 \\ d_1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \text{Root} \mapsto \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \text{contents} \mapsto \begin{bmatrix} c_0 & 1 & c_2 & c_3 & c_4 \\ c_5 & c_6 & c_7 & c_8 & c_9 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \right\}.$$

$$\mathrm{T_E}[\text{Dir}]e = e(\text{Dir}) = \begin{bmatrix} d_0 \\ d_1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \mathrm{T_E}[\text{File}]e = e(\text{File}) = \begin{bmatrix} 0 \\ 0 \\ f_0 \\ f_1 \\ f_2 \end{bmatrix}, \mathrm{T_E}[\text{contents}]e = e(\text{contents}) = \begin{bmatrix} c_0 & 1 & c_2 & c_3 & c_4 \\ c_5 & c_6 & c_7 & c_8 & c_9 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathrm{T_E}[\text{Dir} \cup \text{File}]e = \mathrm{T_E}[\text{Dir}]e \vee \mathrm{T_E}[\text{File}]e = \begin{bmatrix} d_0 \\ d_1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \vee \begin{bmatrix} 0 \\ 0 \\ f_0 \\ f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ f_0 \\ f_1 \\ f_2 \end{bmatrix}.$$

$$\mathrm{T_E}[\text{Dir} \to (\text{Dir} \cup \text{File})]e = \mathrm{T_E}[\text{Dir}]e \times \mathrm{T_E}[\text{Dir} \cup \text{File}]e = \begin{bmatrix} d_0 & d_1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} d_0 \\ d_1 \\ f_0 \\ f_1 \\ f_2 \end{bmatrix}$$

$$= \begin{bmatrix} d_0 \wedge d_0 & d_0 \wedge d_1 & d_0 \wedge f_0 & d_0 \wedge f_1 & d_0 \wedge f_2 \\ d_1 \wedge d_0 & d_1 \wedge d_1 & d_1 \wedge f_0 & d_1 \wedge f_1 & d_1 \wedge f_2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

$$\mathrm{T_F}[\text{contents} \subseteq \text{Dir} \to (\text{Dir} \cup \text{File})]e = \bigwedge \left(\neg \mathrm{T_E}[\text{contents}]e \vee \mathrm{T_E}[\text{Dir} \to (\text{Dir} \cup \text{File})]e\right)$$

$$= \bigwedge \left(\neg \begin{bmatrix} c_0 & 1 & c_2 & c_3 & c_4 \\ c_5 & c_6 & c_7 & c_8 & c_9 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \vee \begin{bmatrix} d_0 \wedge d_0 & d_0 \wedge d_1 & d_0 \wedge f_0 & d_0 \wedge f_1 & d_0 \wedge f_2 \\ d_1 \wedge d_0 & d_1 \wedge d_1 & d_1 \wedge f_0 & d_1 \wedge f_1 & d_1 \wedge f_2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \right)$$

$$= \bigwedge \begin{bmatrix} \neg c_0 \vee (d_0 \wedge d_0) & \neg 1 \vee (d_0 \wedge d_1) & \neg c_2 \vee (d_0 \wedge f_0) & \neg c_3 \vee (d_0 \wedge f_1) & \neg c_4 \vee (d_0 \wedge f_2) \\ \neg c_5 \vee (d_1 \wedge d_0) & \neg c_6 \vee (d_1 \wedge d_1) & \neg c_7 \vee (d_1 \wedge f_0) & \neg c_8 \vee (d_1 \wedge f_1) & \neg c_9 \vee (d_1 \wedge f_2) \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= (\neg c_0 \vee (d_0 \wedge d_0)) \wedge (\neg 1 \vee (d_0 \wedge d_1)) \wedge (\neg c_2 \vee (d_0 \wedge f_0)) \wedge (\neg c_3 \vee (d_0 \wedge f_1)) \wedge (\neg c_4 \vee (d_0 \wedge f_2)) \wedge$$
$$(\neg c_5 \vee (d_1 \wedge d_0)) \wedge (\neg c_6 \vee (d_1 \wedge d_1)) \wedge (\neg c_7 \vee (d_1 \wedge f_0)) \wedge (\neg c_8 \vee (d_1 \wedge f_1)) \wedge (\neg c_9 \vee (d_1 \wedge f_2)).$$

Figure 2-4: A sample translation. The shading highlights the redundancies in the boolean encoding.

Facts vs. assertions

34

- Alloy *facts* define truths of the model
- Alloy *assertions* define truths you'd like check to make sure that they hold about the model
- Closely related to Jackson père's discussion of moods
  - Indicative mood: describes how things in the world are regardless of the behavior of the system: "Each seat is located in one and only one theater."
  - Optative mood: describes what you want the system to achieve: "Better seats should be allocated before worse seats at the same price."

And to designations and definitions

Also Jackson père

35

- *Designations* are atomic phenomena
  - e.g., genetic mother
- *Definitions* define terms using designations and other definitions
  - e.g., genetic child of
- Refutable descriptions can in principle be disproven
  - ∀m,x•Mother(m,x)→¬Mother(x,m)
  - Can't do this with definitions
- The precise distinction between these is, to me, far more important than the formalization

Alloy … a brief demo

36