Toward Formally Verified Finance

Lessons from Model Checking

Quinn Dougherty

Casper Association

2024-05-02

1. ACTUS
2. Logic and quality assurance
3. Logic and time
4. Automata
5. Model checking
6. TODO: delete

Overview

- ACTUS: finance down to state machine abstraction
- Formal verification: way better quality assurance
- Temporal logic: specify behaviors with time
- Automata: execution abiding by temporal logic specifications
- Reactive systems and model checking

Algorithmic Contract Types Unified Standard

ACTUS simulates *cash flows* with a *state machine* abstraction.

ACTUS state machine

writes about the actus state machines

A logical formula is some propositions connected by the operators

- To "prove" a formula is to convince a skeptic that the formula is "true" or "valid"
- Formulae are types, proofs are programs

Modus ponens

If it is raining, then the ground is wet. It is raining. Therefore, the ground is wet

Modus tollens

If it is snowing, then it is cold outside. It is not cold outside. Therefore, it is not snowing

Modus ponens

$$(p \to q \land p) \to q$$

Modus tollens

$$(p \to q \land \neg q) \to \neg p$$

Formal Verification

Logic can set itself on any kind of mathematical object or phenomenon

When we use logic to study software, we're doing "formal verification"

Formal verification is kin with ordinary software testing, but much stronger

Providing steeper assurances

Baby testing: units

Come up with the cases you have time to enumerate

```
def is_even(x: int) -> bool:
  ...

assert is_even(2)
assert is_even(4)
```

Intermediate testing: properties

Procedurally generate unit tests (100-10000 cases)

```python
def is_even(x: int) -> bool:
  if x == 2: return True
  elif x == 4: return True
  else: return False

@hypothesis.given(hypothesis.strategies.integers())
def is_even_agrees(x: int) -> bool:
  is_even(x) == (x % 2 == 0)
# Test fails with generated counterexample x = 6
```

Ascended testing: formal verification

A type checker in a total language can **exploit the structure** of datatypes and functions to prove over a whole type without exhaustively checking every value

Truth when?

We can make logical formulae specify properties involving time by introducing new operators called *modal operators*

- $\Box p$:= always p
- $\Diamond p$:= eventually p
- pUq := p until q

Automata

draws a big ole automata with the `finite` library Automata

Automata theory for finance

Automata theory for finance

It turns out

ACTUS' notion of state machine is a special case of what's studied in *automata theory*

Automata theory for finance

It turns out

ACTUS’ notion of state machine is a special case of what’s studied in *automata theory*

Automata form semantics for temporal formulae

Automata theory for finance

It turns out

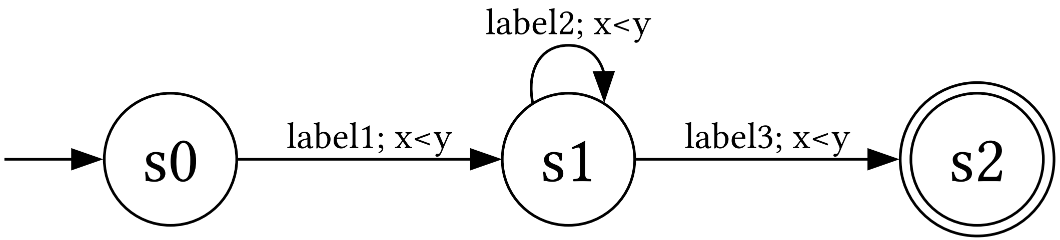
ACTUS' notion of state machine is a special case of what's studied in *automata theory*

Automata form semantics for temporal formulae

Automata provide the execution environment for model checking

Automata theory for finance

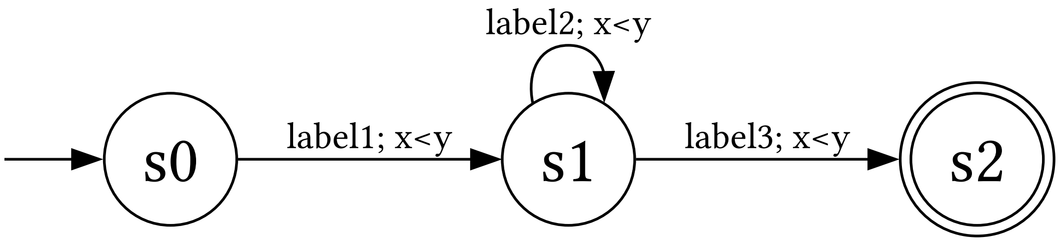
An automaton is an abstraction of computation consisting of states connected by events



Automata theory for finance

An automaton is an abstraction of computation consisting of states connected by events

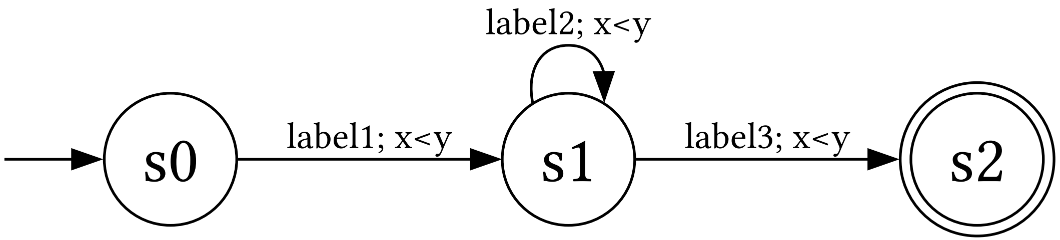
- In a *finite* automaton, some states are distinguished as "final"



Automata theory for finance

An automaton is an abstraction of computation consisting of states connected by events

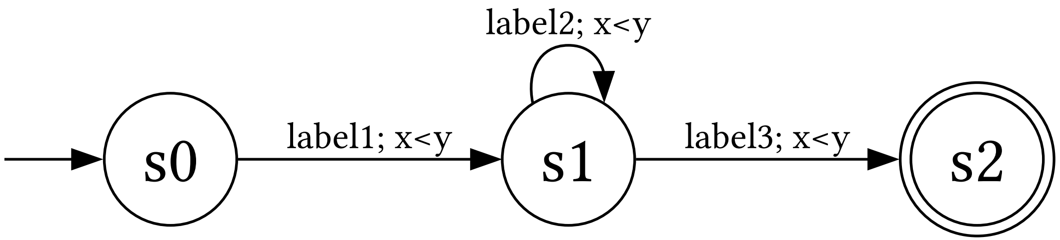
- In a *finite* automaton, some states are distinguished as "final"
- In a *timed* automaton, transitions (traversing along events) increment some "clocks", and events can only fire if "guard conditions" on those clocks are met



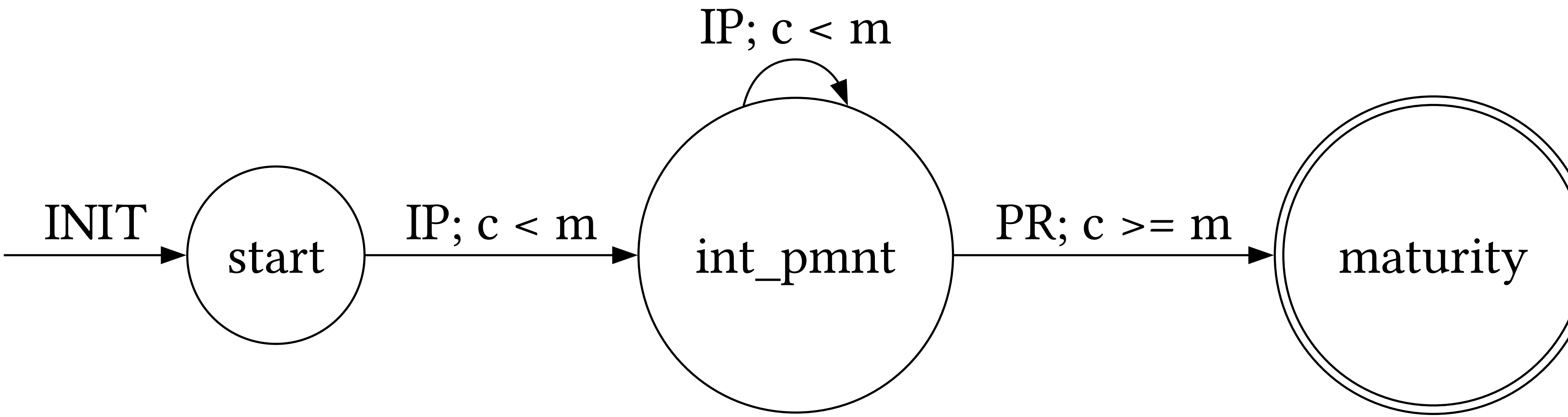
Automata theory for finance

An automaton is an abstraction of computation consisting of states connected by events

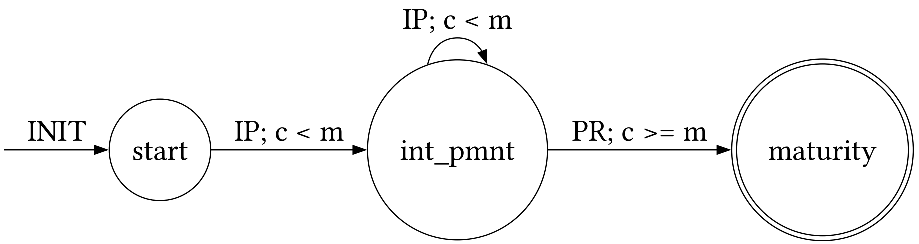
- In a *finite* automaton, some states are distinguished as "final"
- In a *timed* automaton, transitions (traversing along events) increment some "clocks", and events can only fire if "guard conditions" on those clocks are met
- In our case, the *guard conditions* are *event labels*



Principal at Maturity (PAM)



- c: *clock*
- m: *maturity date*
- IP: interest payment event
- PR: principal repayment event
- All events increment clock c by 1



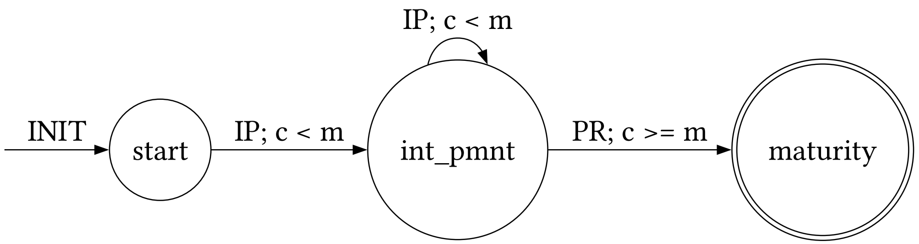
We can *run* PAM as a timed finite automaton to elicit a *trace*

The trace (for m = 2):

0. Start with empty trace

[]

Principal At Maturity (PAM)



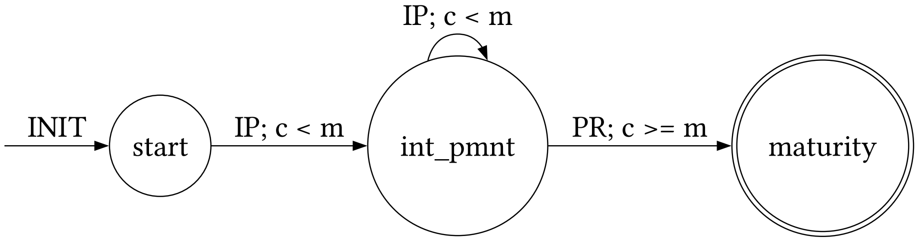
We can *run* PAM as a timed finite automaton to elicit a *trace*

The trace (for m = 2):

1. Enter contract at `start` state
   - c = 0
   - push `INIT` to trace

Result: [`INIT`]

Principal At Maturity (PAM)



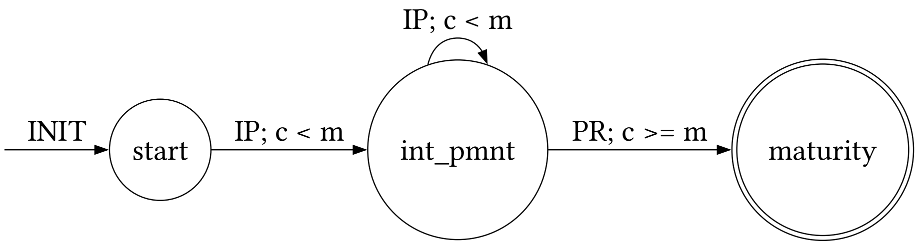
We can *run* PAM as a timed finite automaton to elicit a *trace*

The trace (for m = 2):

2. Apply an interest payment with the `IP` event, evaluating the guard $0 < 2$ to enter `int_pmnt` state
   - c = 1
   - push `IP` to trace

Result: [`INIT`, `IP`]

Principal At Maturity (PAM)



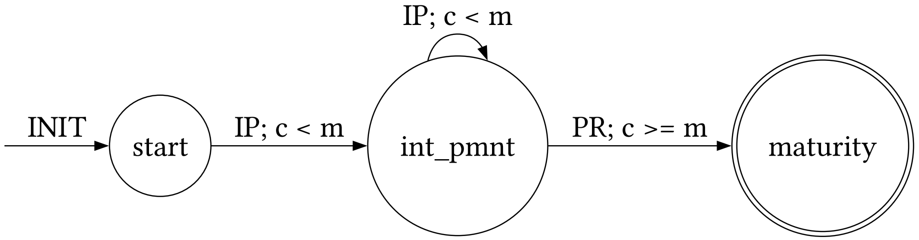
We can *run* PAM as a timed finite automaton to elicit a *trace*

The trace (for m = 2):

3. Apply an interest payment with the `IP` event, evaluating the guard $1 < 2$ to enter `int_pmnt` state
   - c = 2
   - push `IP` to trace

Result: [`INIT`, `IP`, `IP`]

Principal At Maturity (PAM)



We can *run* PAM as a timed finite automaton to elicit a *trace*

The trace (for m = 2):

4. $2 \nless 2$, but $2 \geq 2$, so we take the `PR` (principal repayment) event instead, entering the `maturity` state which is final
   - c = 3 (doesn't matter)
   - push `PR` to trace

So the run produces trace: [`INIT`, `IP`, `IP`, `PR`]

Reactive systems

- A **reactive system** is a software system embedded in an environment that responds to sensor input
  - often in continuous/infinite time horizon
  - often with actuator output effecting the environment

Formal verification for reactive systems

Recall that formal verification deals in mathematical proofs of software correctness

Formal verification for reactive systems

Recall that formal verification deals in mathematical proofs of software correctness

- A temporal logic forms a **specification language** in which normative constraints for reactive systems can be captured

Formal verification for reactive systems

Recall that formal verification deals in mathematical proofs of software correctness

- A temporal logic forms a **specification language** in which normative constraints for reactive systems can be captured
- **Model checking** is the discipline of turning programs into automata and showing that the automata is validated by a spec
  - Two key types of properties are *safety* properties and *liveness* properties

Safety

nothing bad ever happens

Liveness

something good eventually happens

notes (not actually gonna be a slide)

- it’d be good to have a section on the different usecases of formal verification, like modeling vs verification of prod code
- unpack what model checking is

top

bottom

Lorem ipsum dolor sit amet, consectetur adipiscing elit.	Lorem ipsum dolor sit amet, consectetur adipiscing elit.	Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Lorem ipsum dolor sit amet, consectetur adipiscing elit.	Lorem ipsum dolor sit amet, consectetur adipiscing elit.	Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Lorem ipsum dolor sit amet, consectetur adipiscing elit.	Lorem ipsum dolor sit amet, consectetur adipiscing elit.	Lorem ipsum dolor sit amet, consectetur adipiscing elit.