A Formal Executable Semantics of PROMELA

Byoungho Son, Kyungmin Bae

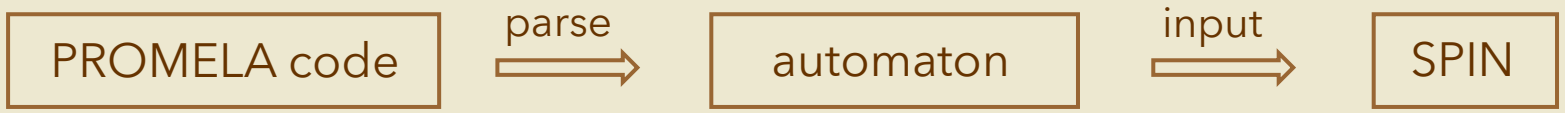
POSTECH, South Korea

VMCAI 2026, Rennes, France

POSTECH

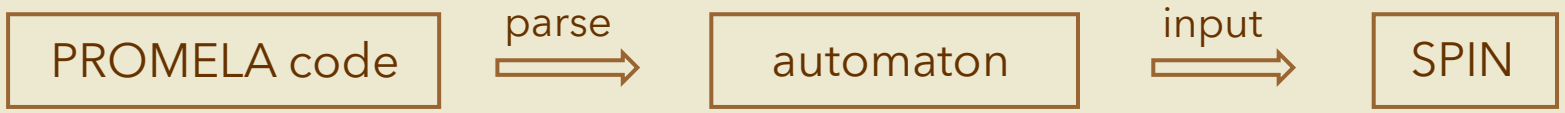
The PROMELA/SPIN tandem

- Widely used for modeling & verifying concurrent/distributed systems
  - **PROMELA** is the input modeling language
  - **SPIN** is the model checker
  - workflow :



The PROMELA/SPIN tandem

- Widely used for modeling & verifying concurrent/distributed systems
  - **PROMELA** is the input modeling language
  - **SPIN** is the model checker
  - workflow :



- Strength :
  - SPIN : fully automatic & efficient
  - PROMELA : intuitive high-level modeling language
  - many application domains (e.g., crypto protocols, linux system calls, etc.)
  - received ACM Software System Award 2001

Limitations of PROMELA/SPIN

- SPIN only supports **explicit** model checking
- Cannot verify properties for **infinite** systems
- No support for **code-level** deductive verification
  - no prior work on PROMELA semantics aimed at deductive reasoning
  - prior work focus on translation from PROMELA to automaton

Our goal

- Define an **executable** semantics of PROMELA
- **Mechanize** the semantics to enable automatic generation of tools
- Derive a **code-level** deductive verifier from the mechanized semantics

- hope : enable wider range of analysis of existing PROMELA models

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

variables (e.g., integers, channels)

concurrent processes

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Inter-process communication

- c : handshake channel (synchronous)
- d : buffered channel (asynchronous)

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Inter-process communication

- c : handshake channel (synchronous)
- d : buffered channel (asynchronous)

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Inter-process communication

- c : handshake channel (synchronous)
- d : buffered channel (asynchronous)

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Execution trace

- sequence of atomic actions
- interleaved / synchronized

p1	p2
d ! 42	
	d ? x
x++	
	x++
c ! 42	c ? x

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Execution trace

- sequence of atomic actions
- interleaved / synchronized

p1	p2
d ! 42	
	d ? x
x++	
	x++
c ! 42	c ? x

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Execution trace

- sequence of atomic actions
- interleaved / synchronized

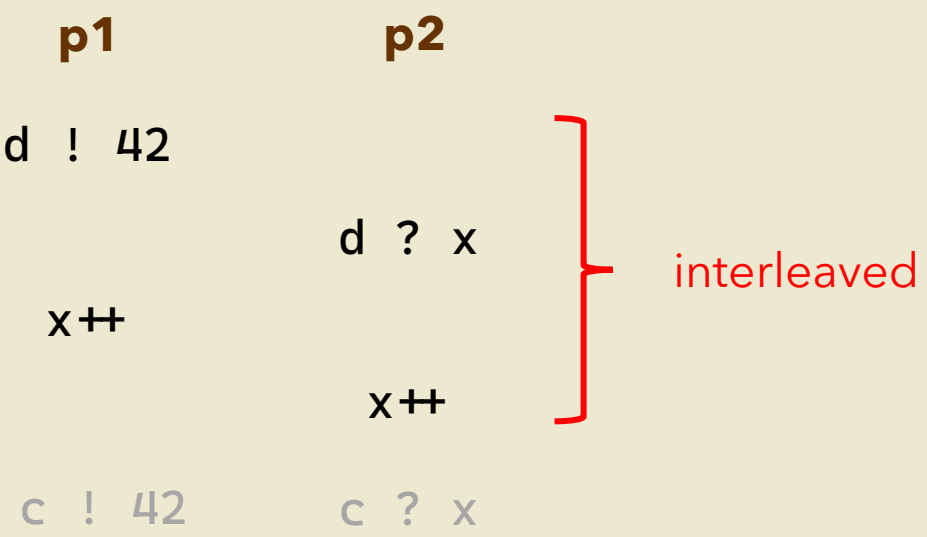
p1	p2
d ! 42	
	d ? x
x++	
	x++
c ! 42	c ? x

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Execution trace

- sequence of atomic actions
- interleaved / synchronized

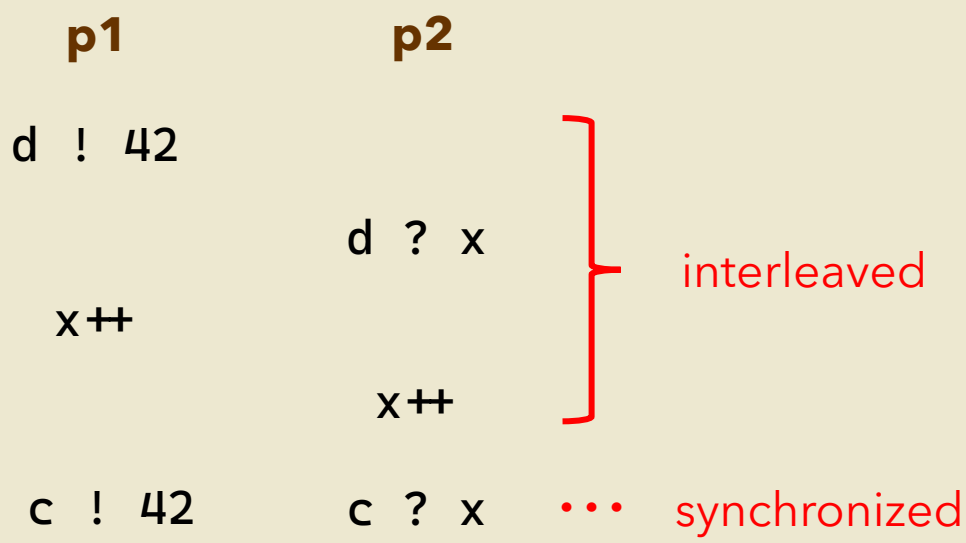


PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Execution trace

- sequence of atomic actions
- interleaved / synchronized

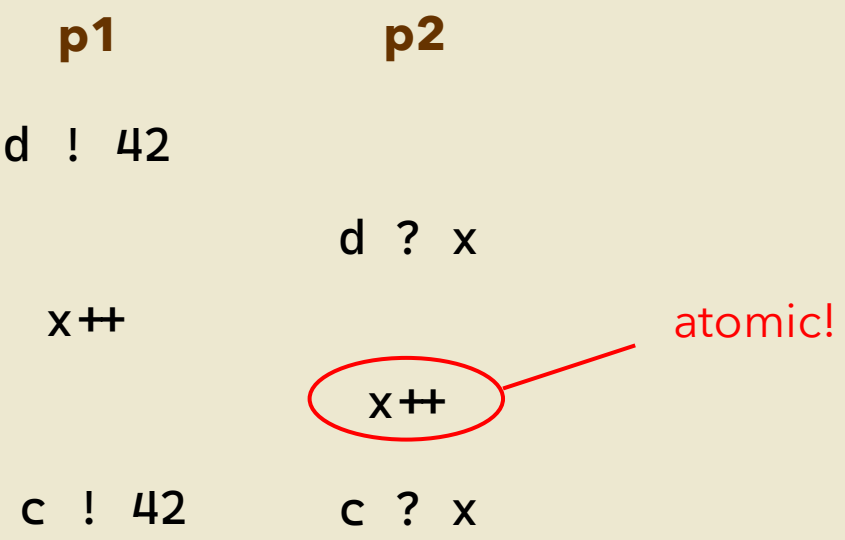


PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Execution trace

- sequence of atomic actions
- interleaved / synchronized



PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
     :: c ! 42 → x++; // branch 1
     :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
     :: x == 1 → x++; // branch 3
     :: do
          :: c ? x → x++; // branch 4
          :: d ? x → x++; // branch 5
        od
  fi
}
```

Nondeterminism

- `if … fi` : nondeterministic selection
- `do … od` : nondeterministic repetition

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Enabledness

- A statement can be executed iff it is enabled
- A branch can be selected iff the guard is enabled

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

selectable?

Enabledness

- A statement can be executed iff it is enabled
- A branch can be selected iff the guard is enabled

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Enabledness

- A statement can be executed iff it is enabled
- A branch can be selected iff the guard is enabled

enabled?

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Enabledness

- A statement can be executed iff it is enabled
- A branch can be selected iff the guard is enabled

enabled?

PROMELA in a nutshell

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

enabled?

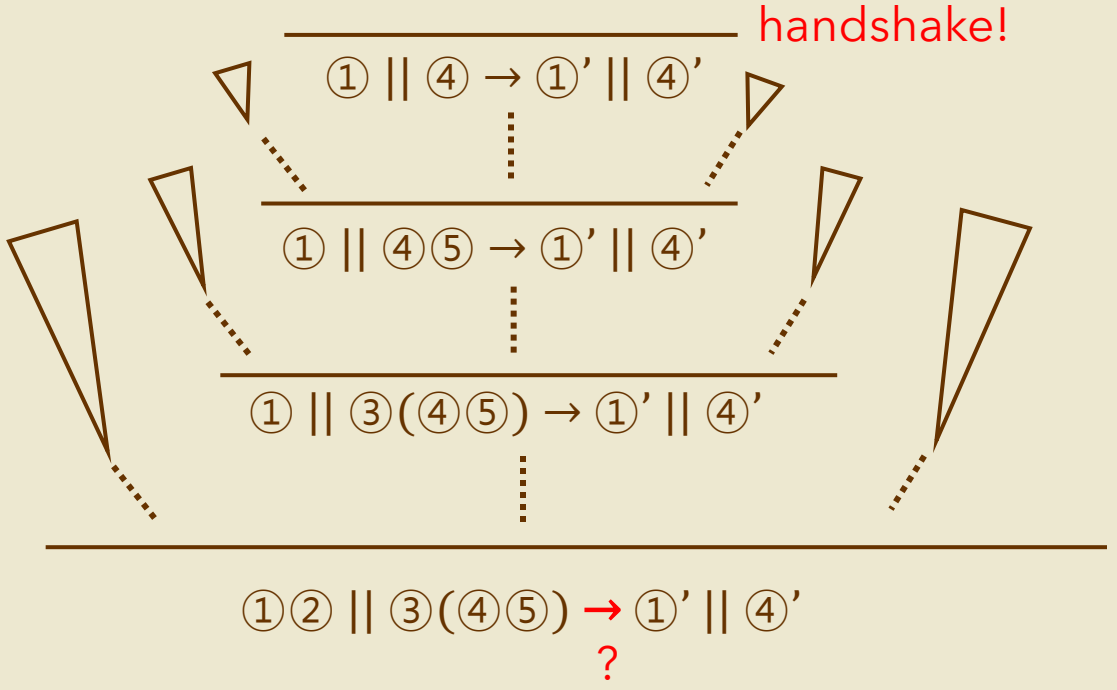
Enabledness

- A statement can be executed iff it is enabled
- A branch can be selected iff the guard is enabled

PROMELA in a nutshell

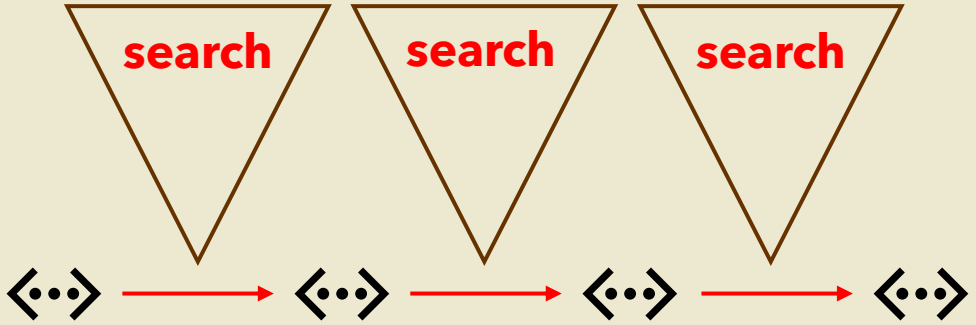
```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

Proof search over nondeterminism & concurrency



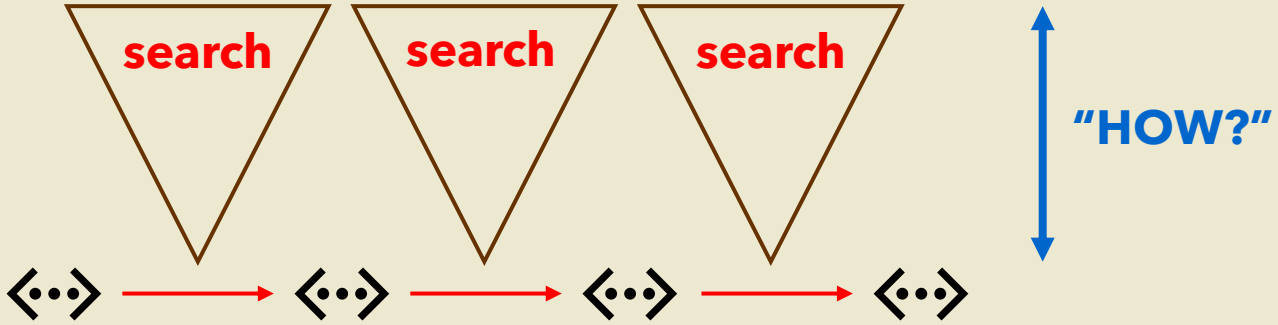
Challenge

- Problem: dichotomy b/w **structural** & **operational** steps



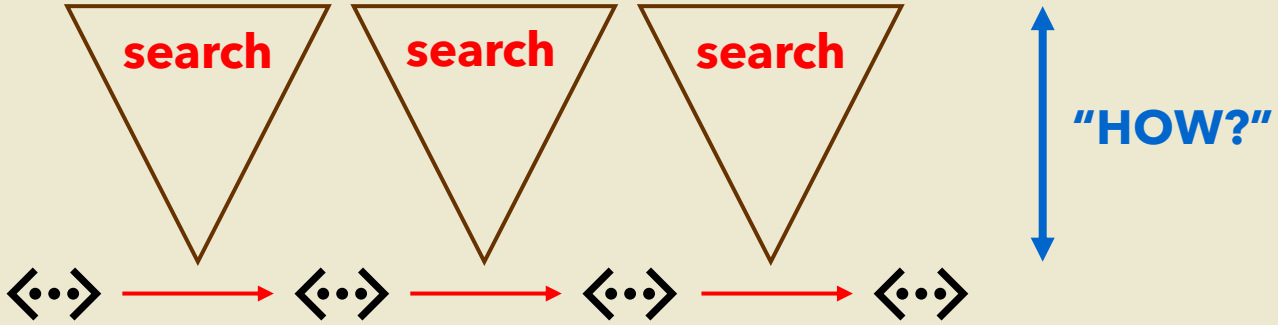
Challenge

- Problem: dichotomy b/w structural & operational steps



Challenge

- Problem: dichotomy b/w structural & operational steps



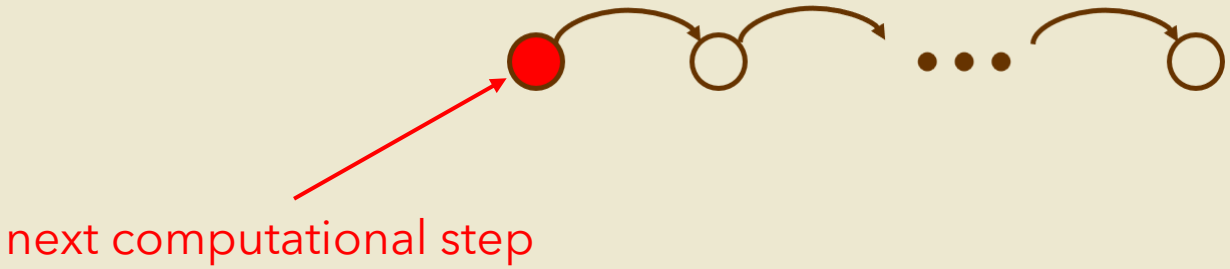
- **Challenge**
  - **design an executable semantics as concrete as an interpreter**
  - so that every operational steps can be executed without any reasoning

Our approach

- Represent PROMELA programs as **continuations** of computational tasks

Our approach

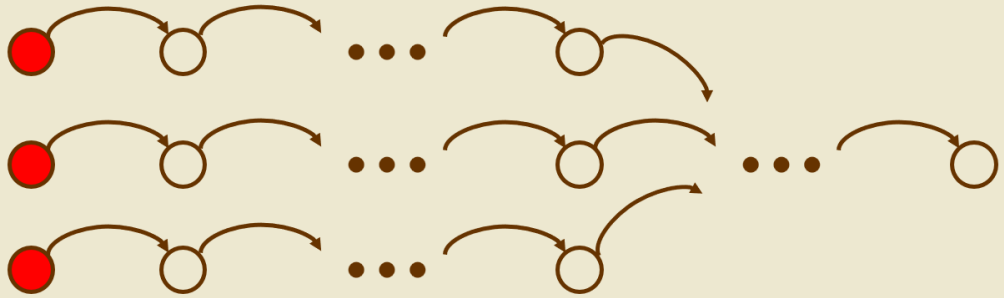
- Represent PROMELA programs as continuations of computational tasks
- Inspired by continuation-based semantics
  - programs are **flattened** into sequences of computational tasks
  - internal computational steps are represented **explicitly**



Our approach

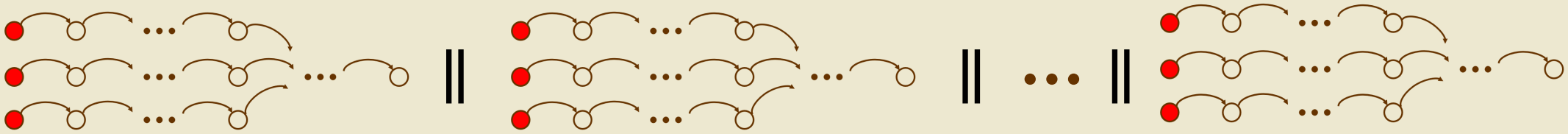
- Represent PROMELA programs as continuations of computational tasks
- Inspired by continuation-based semantics
  - programs are flattened into as sequences of computational tasks
  - internal computational steps are represented explicitly

- Generalization: **Forked continuation**
  - **proof obligations** as special cases of computational tasks



Nondeterminism under concurrency

- Nondeterministic concurrent processes as a **forest**
  - trees for processes
  - branches for nondeterministic options

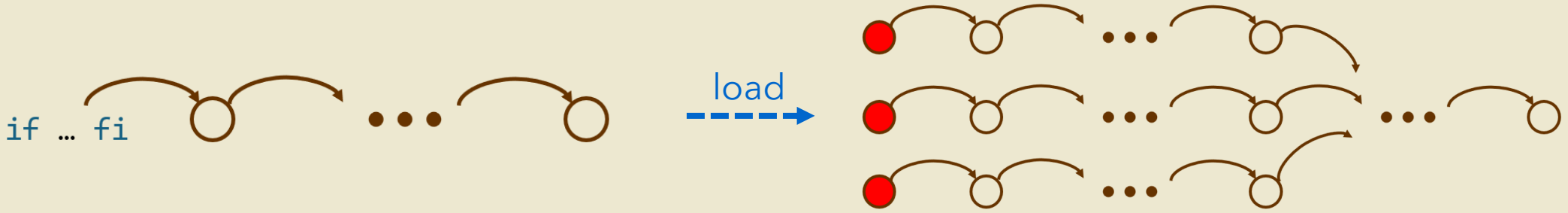


Key Idea: Load-and-Fire

- Semantics design pattern for PROMELA
  - **load rules**: proof search (w/o side-effects)
  - **fire rules**: discharge a proof obligation (with side-effects)

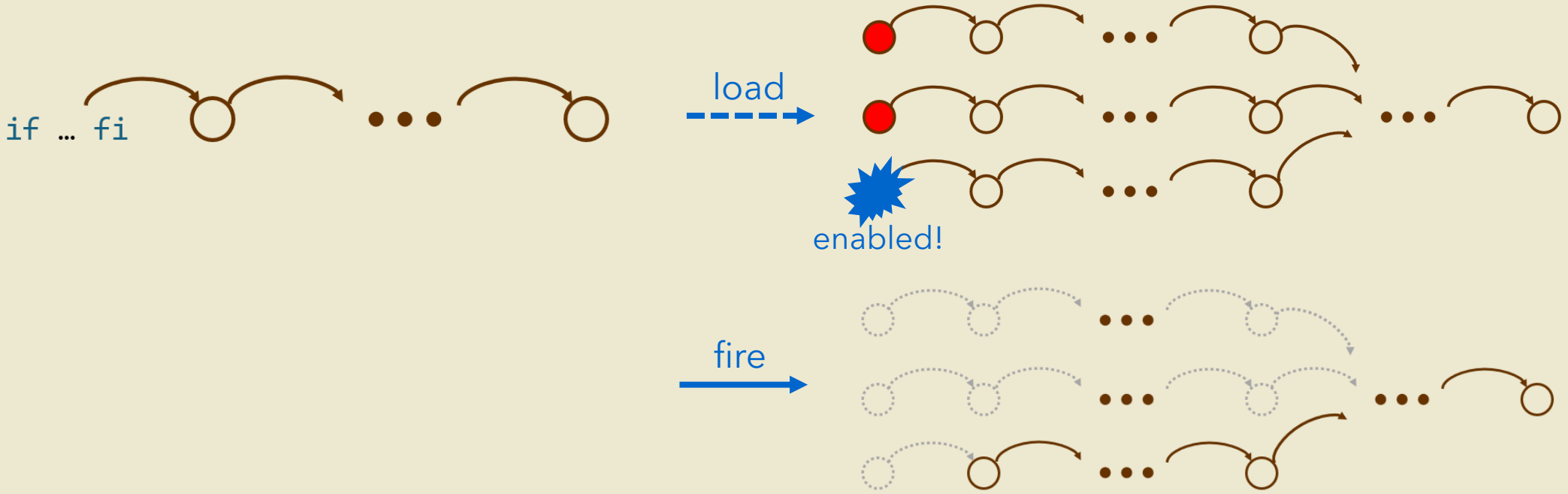
Key Idea: Load-and-Fire

- Semantics design pattern for PROMELA
  - load rules: **proof search** (w/o side-effects)
  - fire rules: discharge a proof obligation (with side-effects)



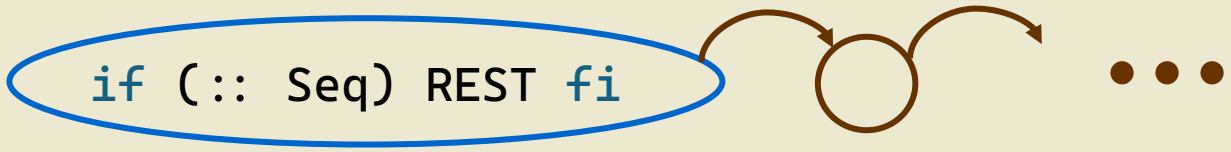
Key Idea: Load-and-Fire

- Semantics design pattern for PROMELA
  - load rules: proof search (w/o side-effects)
  - fire rules: **discharge** a proof obligation (with side-effects)



Semantics of if-statements

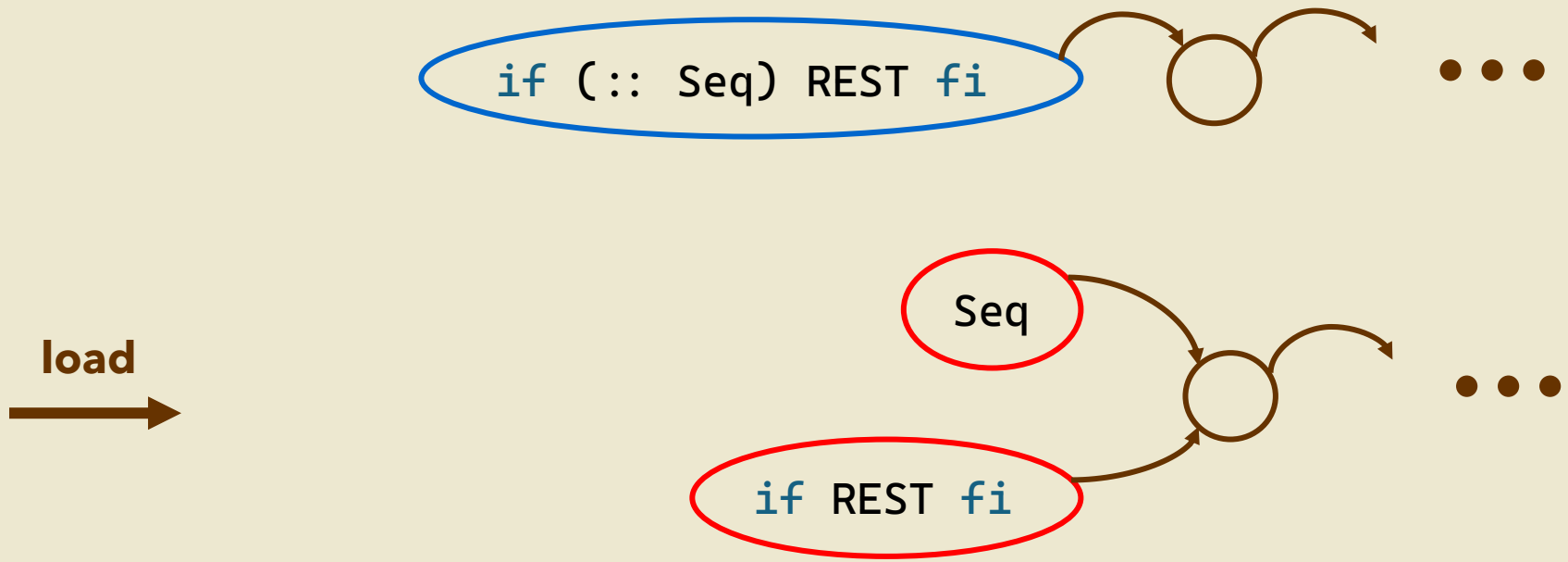
- Specified by a load rule
  - "decompose a proof obligation at leaf"



load

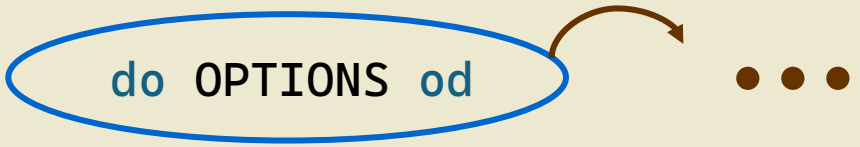
Semantics of if-statements

- Specified by a load rule
  - "decompose a proof obligation at leaf"



Semantics of do-statements

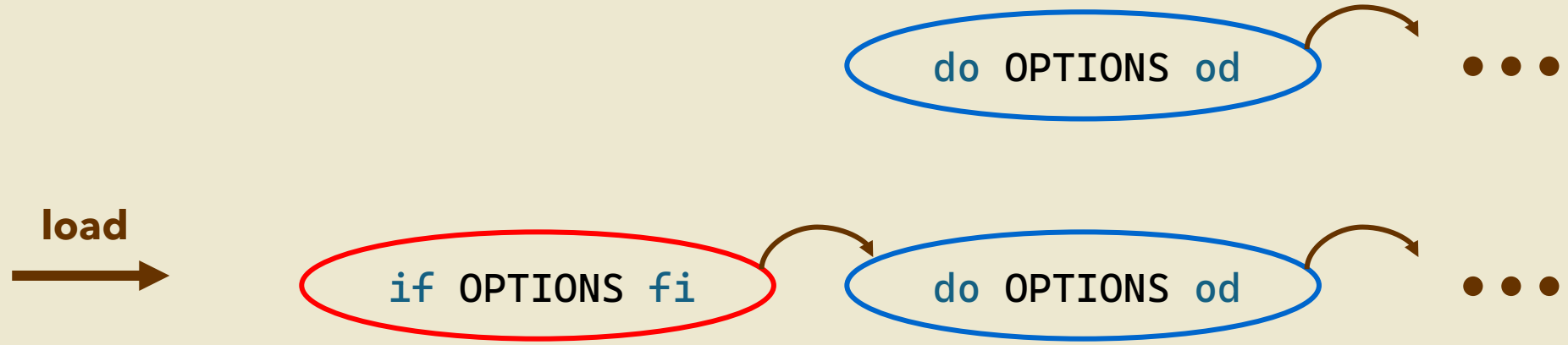
- Specified by a load rule
  - "generate a new proof obligation at leaf"



load

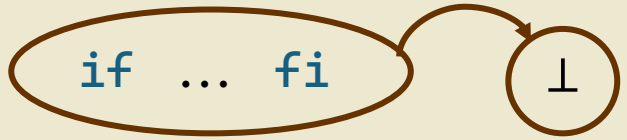
Semantics of do-statements

- Specified by a load rule
  - "generate a new proof obligation at leaf"



Running example 1

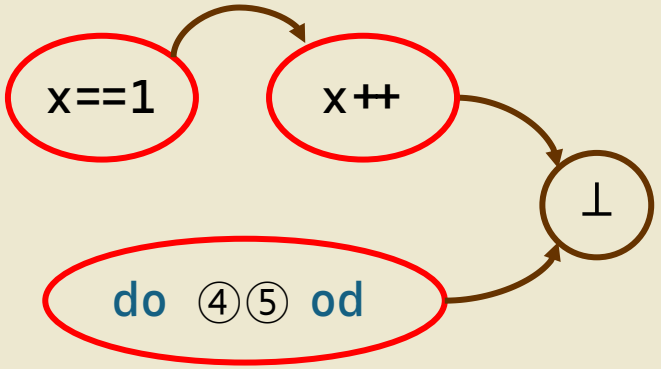
```
proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```



Running example 1

```
proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

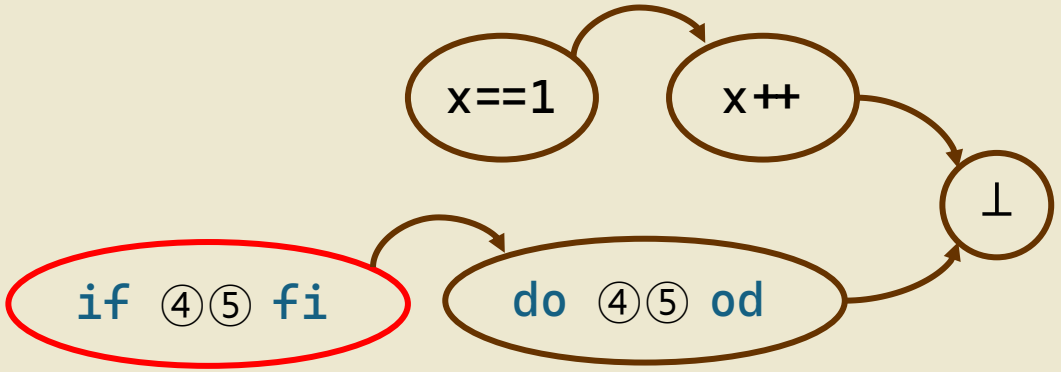
load



Running example 1

```
proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

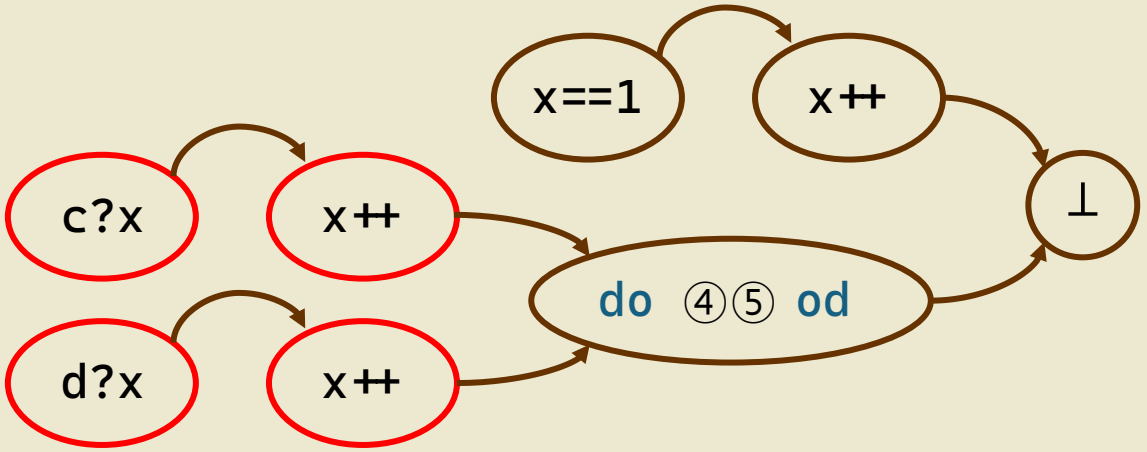
load



Running example 1

```
proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

load

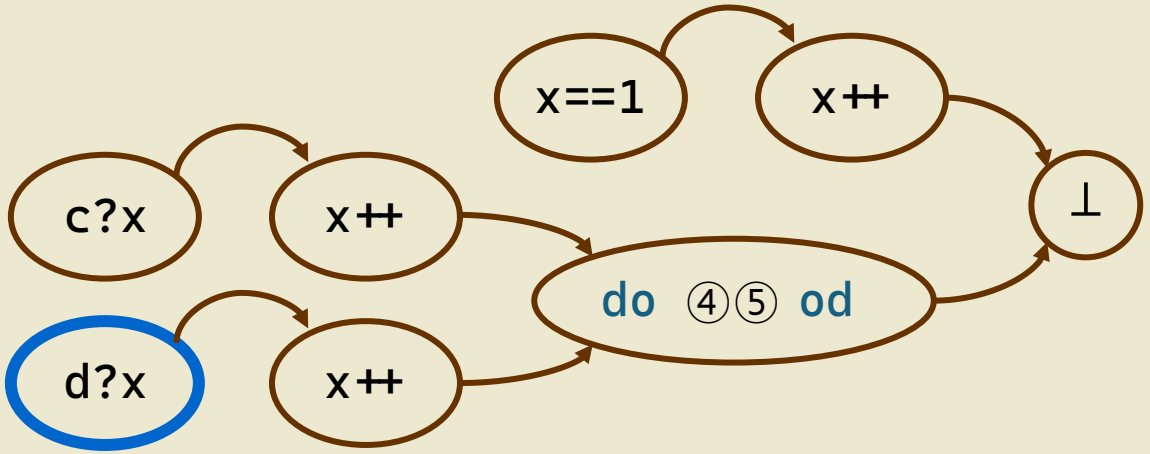


Running example 1

```
proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

load

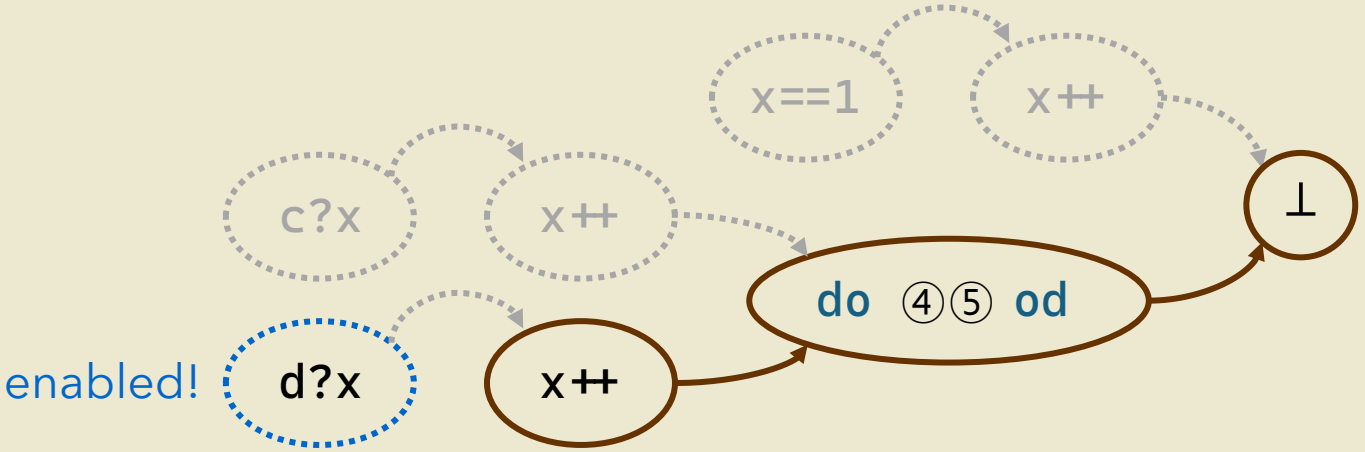
assume d is nonempty



Running example 1

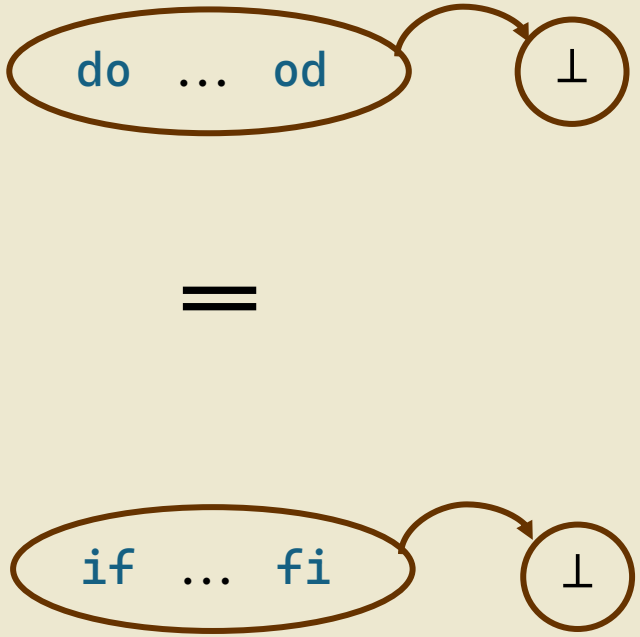
```
proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```

fire



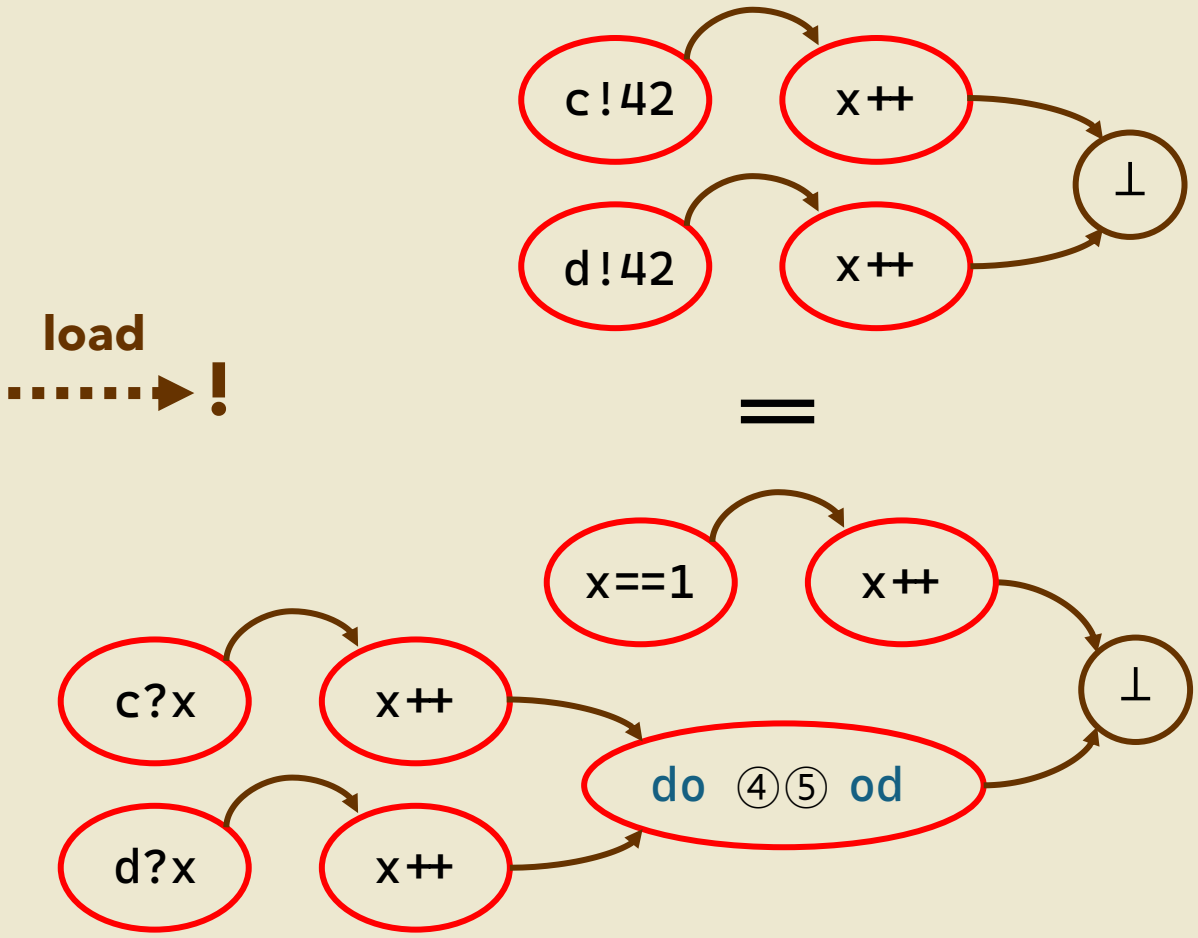
Running example 2

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```



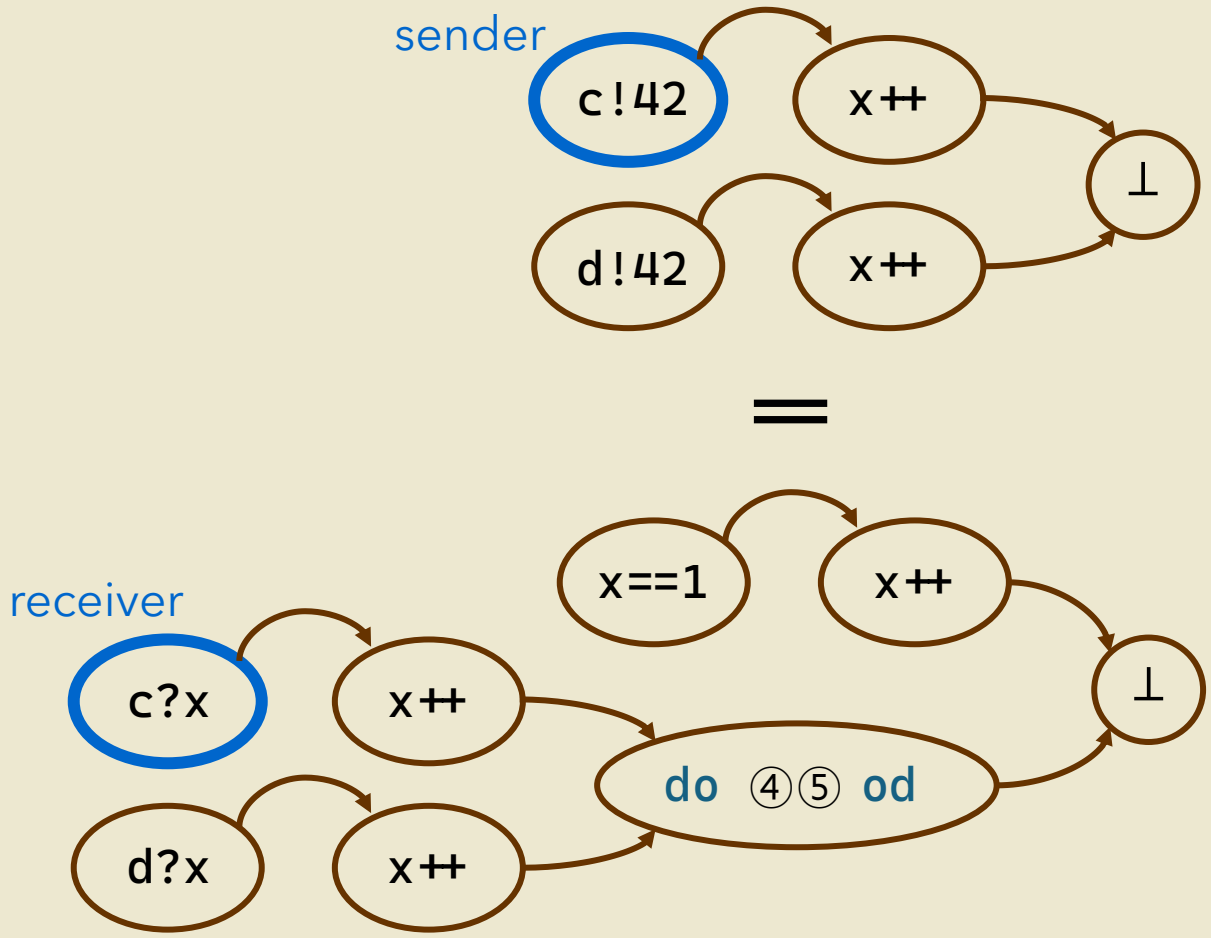
Running example 2

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```



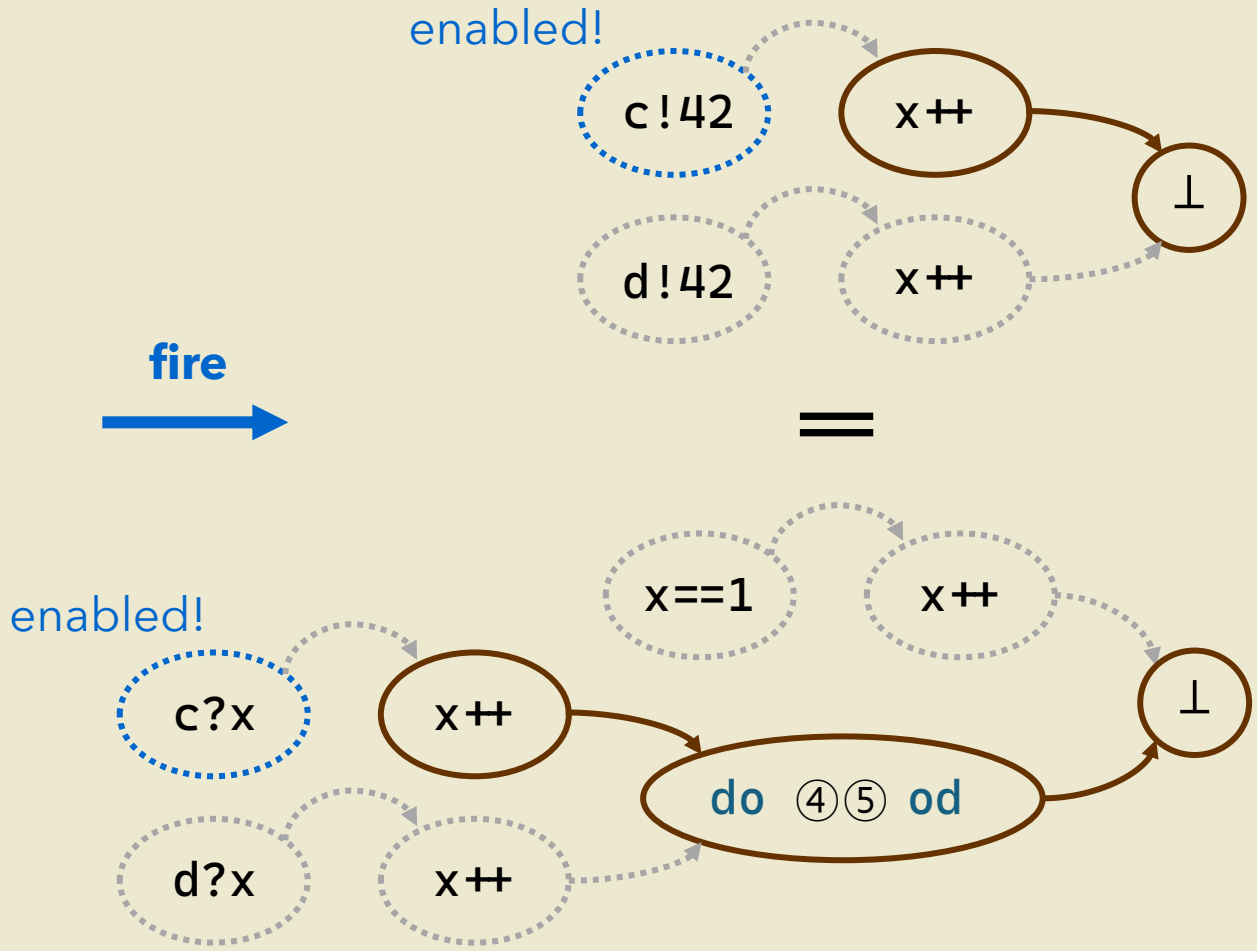
Running example 2

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```



Running example 2

```
chan c = [0] of { int };
chan d = [3] of { int };
int x = 0;

proctype p1() {
  do
    :: c ! 42 → x++; // branch 1
    :: d ! 42 → x++; // branch 2
  od
}

proctype p2() {
  if
    :: x == 1 → x++; // branch 3
    :: do
         :: c ? x → x++; // branch 4
         :: d ? x → x++; // branch 5
       od
  fi
}
```



Load-and-Fire: summary

- Load rules bring proof search into the semantics
- Fire rules fire without structural reasoning
- Alternation of two such stages enables executable semantics

Mechanizing the semantics

- We mechanized our PROMELA semantics in **K Framework**
  - K: semantics framework based on **continuations**
  - formalized most of the core features in PROMELA

Mechanizing the semantics

- We mechanized our PROMELA semantics in K Framework
  - K: semantics framework based on continuations
  - formalized most of the core features in PROMELA
- By design, semantics in K is highly **executable**
  - **automatically** derives model checkers, deductive verifiers, etc. from the semantics
  - success stories: C, Java, Ethereum, etc.

Mechanizing the semantics

- We mechanized our PROMELA semantics in K Framework
  - K: semantics framework based on continuations
  - formalized most of the core features in PROMELA
- By design, semantics in K is highly executable
  - automatically derives model checkers, deductive verifiers, etc. from the semantics
  - success stories: C, Java, Ethereum, etc.
- Used K's **model checker** to empirically **validate** our semantic definitions

Mechanizing the semantics

- We mechanized our PROMELA semantics in K Framework
  - K: semantics framework based on continuations
  - formalized most of the core features in PROMELA
- By design, semantics in K is highly executable
  - automatically derives model checkers, deductive verifiers, etc. from the semantics
  - success stories: C, Java, Ethereum, etc.
- Used K's model checker to empirically validate our semantic definitions
- Used K's **deductive verifier** to formally **verify** PROMELA programs

Case study: deductive verification

- Target: **invariant** properties for **distributed** systems

Case study: deductive verification

- Target: invariant properties for distributed systems
- Sometimes K's deductive verifier can be **more powerful** than SPIN

Case study: deductive verification

- Target: invariant properties for distributed systems
- Sometimes K's deductive verifier can be more powerful than SPIN
- Example: mutex for Lamport's bakery algorithm
  - **infinite** number of reachable states
  - **NOT verifiable by SPIN**
  - verifiable by K's deductive verifer

Lamport's bakery algorithm

- A distributed **mutual exclusion** algorithm proposed by Leslie Lamport

```
int disp = 0, serv = 0, crit = 0;
active [2] proctype p() { // activates p1, p2
  int tick = 0;
  do
    :: atomic { tick = disp; disp = disp + 1 }
      ; atomic { tick == serv; crit = crit + 1 }
      ; atomic { serv = serv + 1; crit = crit - 1 }
  od
}
active proctype monitor() { freeze } // for mutex
```

Lamport's bakery algorithm

- A distributed mutual exclusion algorithm proposed by Leslie Lamport
- Two processes enter C.S. by getting **tickets** from the global counter

```
int disp = 0, serv = 0, crit = 0;
active [2] proctype p() { // activates p1, p2
  int tick = 0;
  do
    :: atomic { tick = disp; disp = disp + 1 }
      ; atomic { tick == serv; crit = crit + 1 }
      ; atomic { serv = serv + 1; crit = crit - 1 }
  od
}
active proctype monitor() { freeze } // for mutex
```

get the ticket from the counter

enter critical section

exit critical section

Lamport's bakery algorithm

- A distributed mutual exclusion algorithm proposed by Leslie Lamport
- Two processes enter C.S. by getting tickets from the global counter
- This is an **INFINITE** system! (out of SPIN's verification scope)

```
int disp = 0, serv = 0, crit = 0;
active [2] proctype p() { // activates p1, p2
  int tick = 0;
  do
    :: atomic { tick = disp; disp = disp + 1 }
      ; atomic { tick == serv; crit = crit + 1 }
      ; atomic { serv = serv + 1; crit = crit - 1 }
  od
}
active proctype monitor() { freeze } // for mutex
```

get the ticket from the counter

enter critical section

exit critical section

Verifying mutual exclusion

pre-condition : “ticket dispenser and server are identically initialized”

```
do
  :: atomic { tick = disp; disp = disp + 1 }
    ; atomic { tick == serv; crit = crit + 1 }
    ; atomic { serv = serv + 1; crit = crit - 1 }
od
```

||

```
do
  :: atomic { tick = disp; disp = disp + 1 }
    ; atomic { tick == serv; crit = crit + 1 }
    ; atomic { serv = serv + 1; crit = crit - 1 }
od
```

post-condition : “number of process in critical section <= 1”

Verifying mutual exclusion

pre-condition : “ticket dispenser and server are identically initialized”

```
do
  :: atomic { tick = disp; disp = disp + 1 }
    ; atomic { tick == serv; crit = crit + 1 }
    ; atomic { serv = serv + 1; crit = crit - 1 }
od
```

||

```
do
  :: atomic { tick = disp; disp = disp + 1 }
    ; atomic { tick == serv; crit = crit + 1 }
    ; atomic { serv = serv + 1; crit = crit - 1 }
od
```

post-condition : “number of process in critical section <= 1”

- Result
  - deductive verifier automatically verified the spec within 5 mins!
  - some auxiliary specs were used additionally to aid the reasoning

Conclusion

- We designed an **executable** semantics of PROMELA, enabled by the semantic pattern **Load-and-Fire** based on **forked continuations**

Conclusion

- We designed an **executable** semantics of PROMELA, enabled by the semantic pattern **Load-and-Fire** based on **forked continuations**
- We **mechanized** the semantics in the K framework, which opens the door to code-level **deductive reasoning** over PROMELA programs

Conclusion

- We designed an **executable** semantics of PROMELA, enabled by the semantic pattern **Load-and-Fire** based on **forked continuations**
- We **mechanized** the semantics in the K framework, which opens the door to code-level **deductive reasoning** over PROMELA programs
- We demonstrated that our deductive verifier can verify systems w/ **infinite number of states**, which is out of SPIN's capability

Conclusion

- We designed an **executable** semantics of PROMELA, enabled by the semantic pattern **Load-and-Fire** based on **forked continuations**
- We **mechanized** the semantics in the K framework, which opens the door to code-level **deductive reasoning** over PROMELA programs
- We demonstrated that our deductive verifier can verify systems w/ **infinite number of states**, which is out of SPIN's capability
- Future work
  - extend the deductive reasoning to **LTL** properties
  - extend the case study for **parametric** models with **arbitrary** number of processes

Conclusion

- We designed an **executable** semantics of PROMELA, enabled by the semantic pattern **Load-and-Fire** based on **forked continuations**
- We **mechanized** the semantics in the K framework, which opens the door to code-level **deductive reasoning** over PROMELA programs
- We demonstrated that our deductive verifier can verify systems w/ **infinite number of states**, which is out of SPIN's capability
- Future work
  - extend the deductive reasoning to **LTL** properties
  - extend the case study for **parametric** models with **arbitrary** number of processes

Merci!