Sound, modular and compositional verification of the input/output behavior of programs

Willem Penninckx, Bart Jacobs, Frank Piessens

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium

ESOP 2015

Contents

- Introduction
- Requirements
- Verification approach
- Examples
- Formal Programming language semantics (with I/O)
- Formal assertion semanics
- Proof rules
- Soundness proof sketch
- Apply I/O style verification on software not performing I/O
- ...

(See paper)

Introduction

Introduction

memory, works.

- { true }
  x := 4
  {x > 3 }

  OK

- {true}
  x := 1
  {x>3}

  Not OK

  (program violates contract)

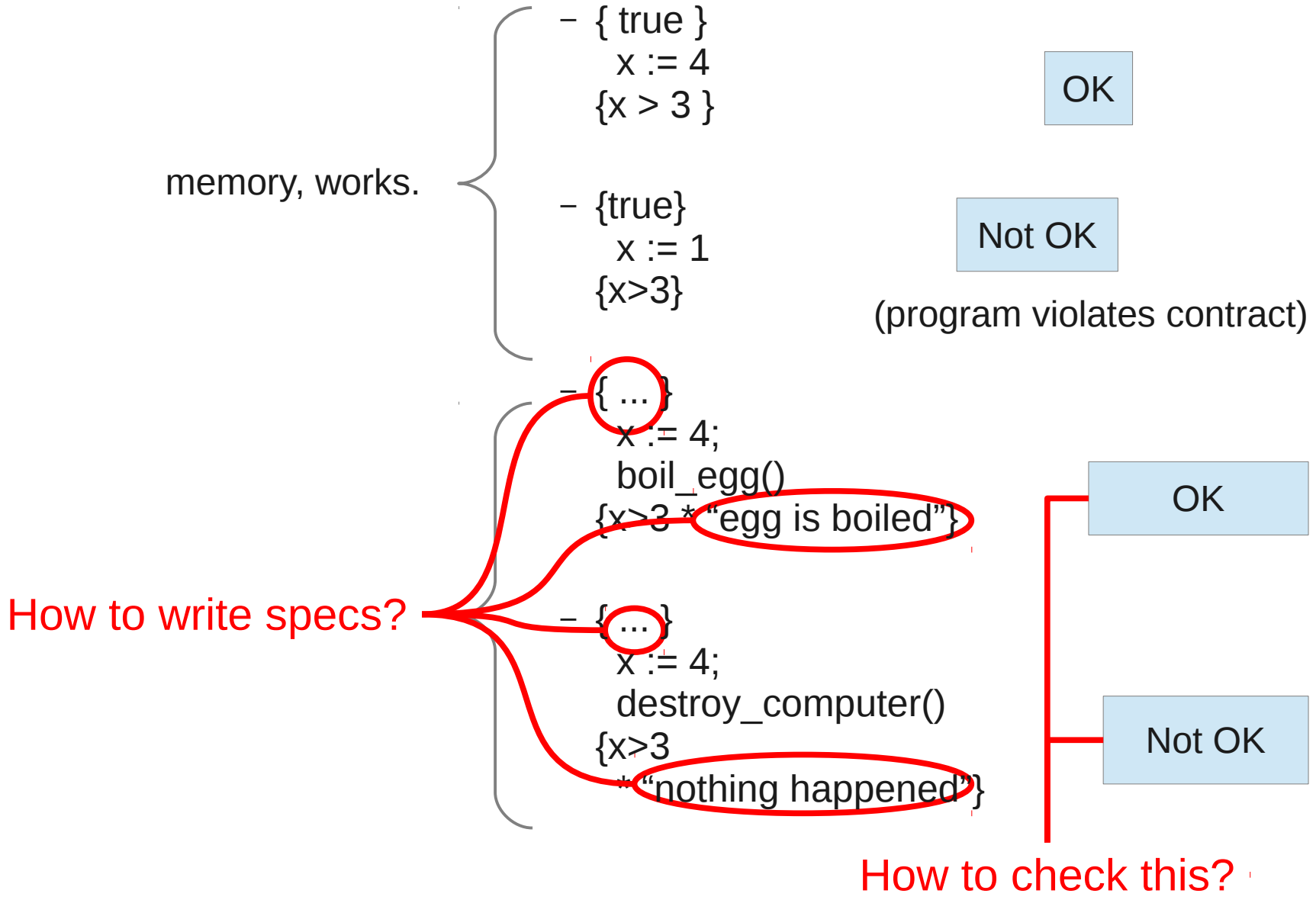
Input/output.
How to verify?

- {true}
  x := 4;
  boil_egg()
  {x>3}

  Maybe OK?

- {true}
  x := 4;
  destroy_computer()
  {x>3}

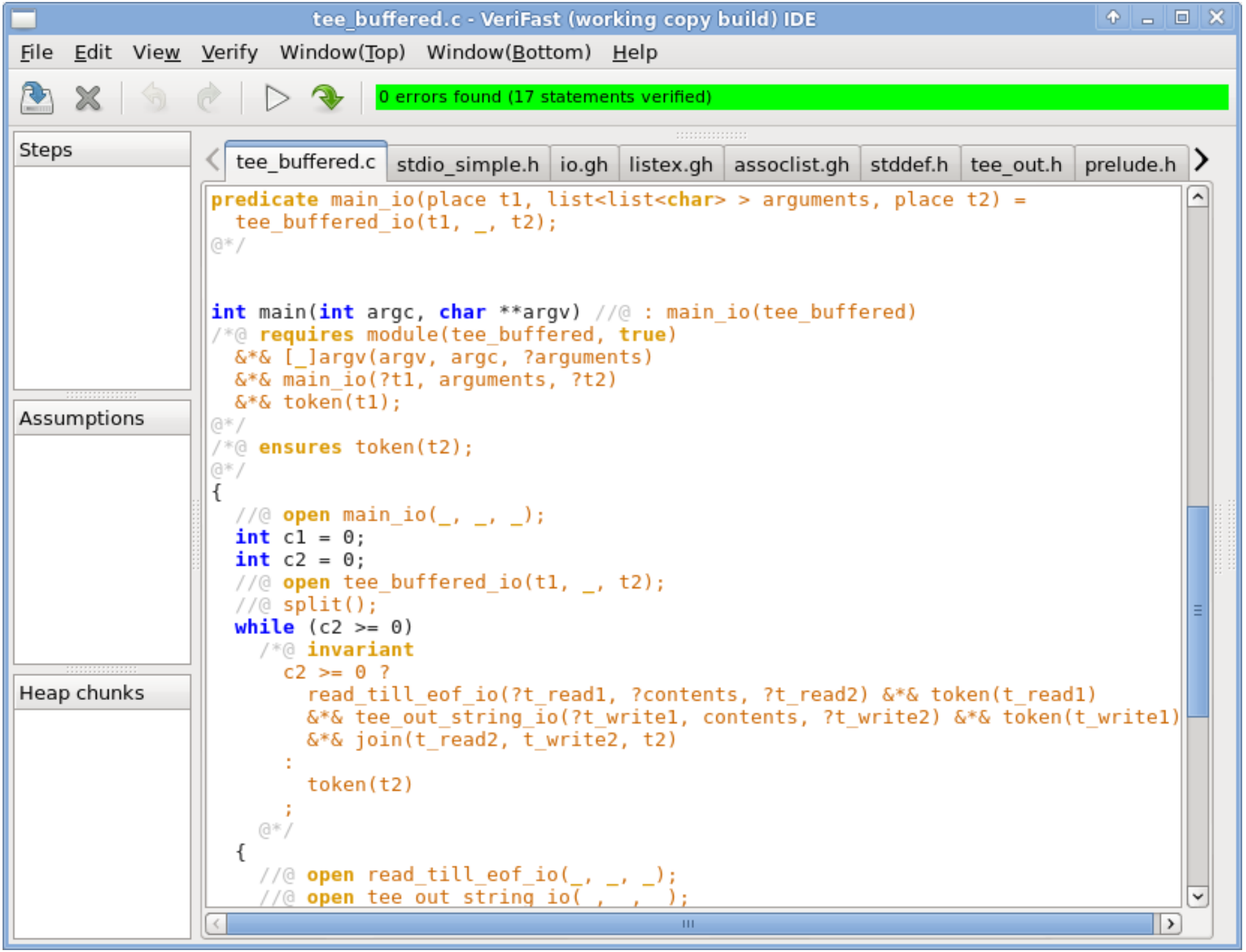
  Maybe not OK?


memory, works.
– { true }
x := 4
{x > 3 }
OK
– {true}
x := 1
{x>3}
Not OK
(program violates contract)
How to write specs?
– { ... }
x := 4;
boil_egg()
{x>3 * “egg is boiled”}
– { ... }
x := 4;
destroy_computer()
{x>3
* “nothing happened”}
OK
Not OK
How to check this?

Requirements

Requirements

- Compositionality
  - of code: f(){ f2(); f3() }
  - of specs: { spec1 } main() { ... } { } spec1 = spec2 * spec3
- Modularity
  - combine code & specs written by different developers
- non-terminating & terminating
- constrain intermediate state / actions
- sound
- static
- imprecise specifications
- ...
- (liveness properties like: “program will actually do I/O”: not in this paper)


tee_buffered.c - VeriFast (working copy build) IDE
File Edit View Verify Window(Top) Window(Bottom) Help
0 errors found (17 statements verified)
Steps
Assumptions
Heap chunks
tee_buffered.c
stdio_simple.h
io.gh
listex.gh
assoclist.gh
stddef.h
tee_out.h
prelude.h
predicate main_io(place t1, list<list<char> > arguments, place t2) =
tee_buffered_io(t1, _, t2);
@*/
int main(int argc, char **argv) //@ : main_io(tee_buffered)
/*@ requires module(tee_buffered, true)
&*& [_]argv(argv, argc, ?arguments)
&*& main_io(?t1, arguments, ?t2)
&*& token(t1);
@*/
/*@ ensures token(t2);
@*/
{
//@ open main_io(_, _, _);
int c1 = 0;
int c2 = 0;
//@ open tee_buffered_io(t1, _, t2);
//@ split();
while (c2 >= 0)
/*@ invariant
c2 >= 0 ?
read_till_eof_io(?t_read1, ?contents, ?t_read2) &*& token(t_read1)
&*& tee_out_string_io(?t_write1, contents, ?t_write2) &*& token(t_write1)
&*& join(t_read2, t_write2, t2)
:
token(t2)
;
@*/
{
//@ open read_till_eof_io(_, _, _);
//@ open tee_out_string_io(_, _, _);

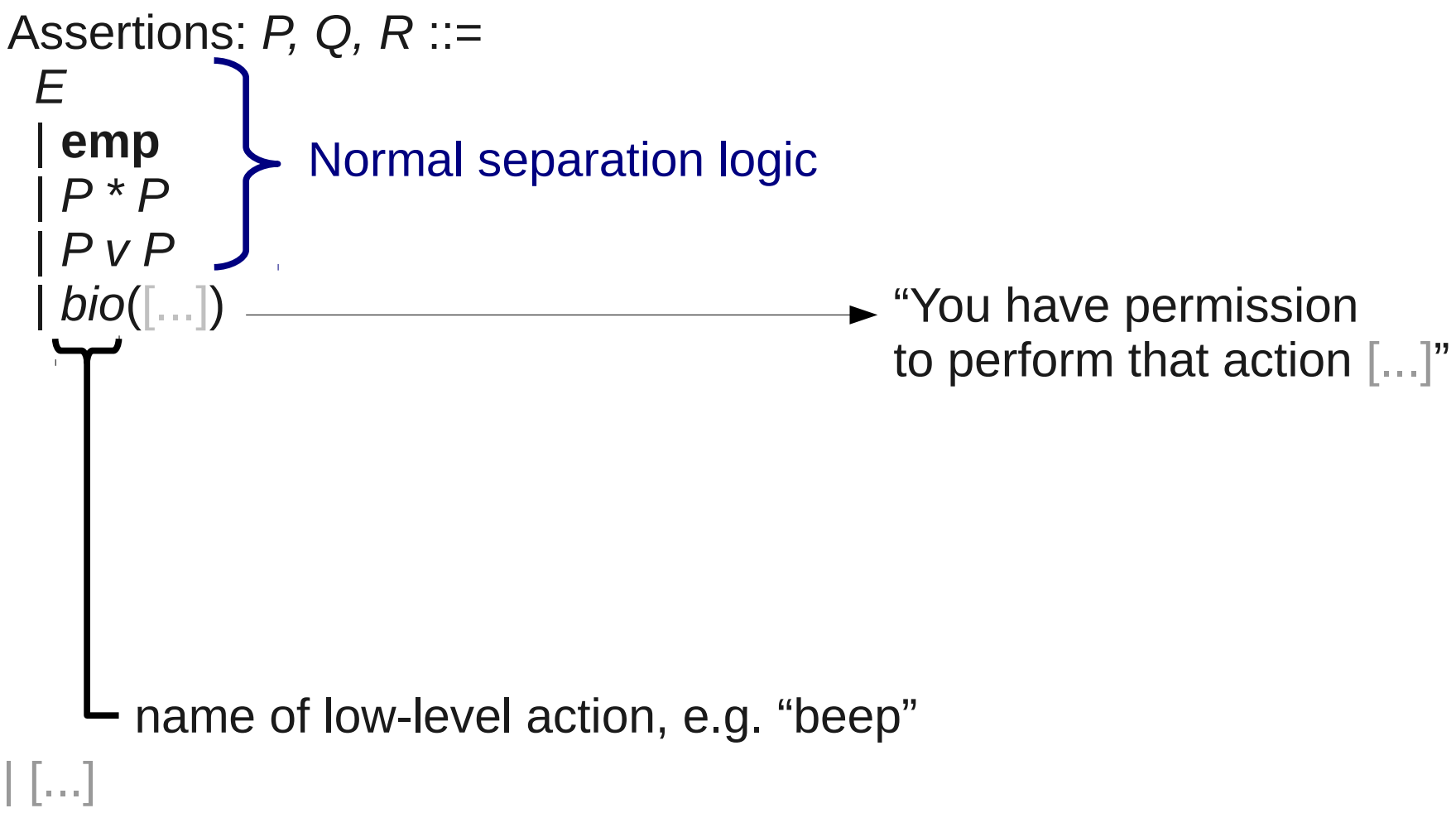
- program: x=read(); write(x+x)
- produces one of the following traces:
  - read(1) :: write(2, [...]) :: nil
  - read(7) :: write(14, [...]) :: nil
  - ...
- So, let's do this:
  - contract describes a set of traces
  - proof rules to check whether program breaks its contract

- { beep([...]) } beep() { [...] }

  Two ways to look at it:

  - precondition describes the traces: beep([...]) :: nil
  - precondition describes permissions to execute actions

Let's start simple



- { beep([...]) [...] }
  beep()
  { [...] }

  Permission disappears after use

- Program: “beep(); led_on();”
- { beep(T1, T2)
  * led_on(T2, T3)
  * **token**(T1) }
    beep();
  { led_on(T2, T3) * **token**(T2) }
    led_on();
  { **token**(T3) }

token “moves”

postcondition: if terminates,
actions must have been performed

Assertions: *P*, *Q*, *R* ::=

E
| **emp**
| *P* * *P*
| *P* v *P*

Normal separation logic

| *bio*(*E*, $\overline{E}$, *E*, *E*)

stop place
input arguments
output argument
start place
name of low-level action

| **token**(E)

| [...]

"You have permission to perform that action if the start-place has a token [...]"

"E has at least one token"

assertion:

heap:

trace:

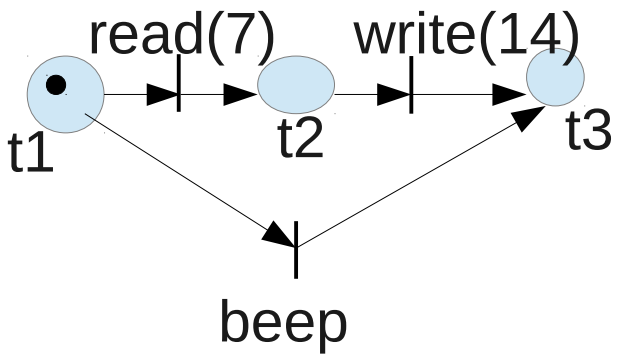
token(T1)
* read(T1, X, T2)
* write(T2, X+X, T3)
* beep(T1, T3)

{**token**(t1), read(t1, 7, t2),
write(t2, 14, t3),
beep(t1, t3)}

read(7) :: write(14) :: nil

beep() :: nil

...

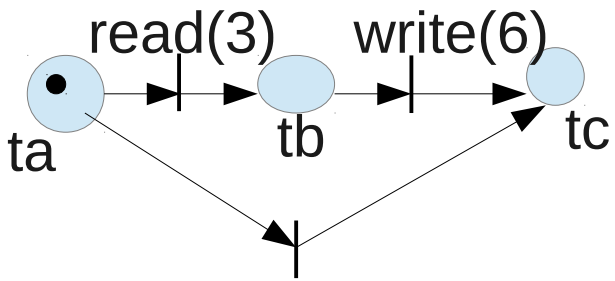


{**token**(ta), read(ta, 3, tb),
write(tb, 6, tc), beep(ta, tc)}

read(3) :: write(6) :: nil

beep() :: nil

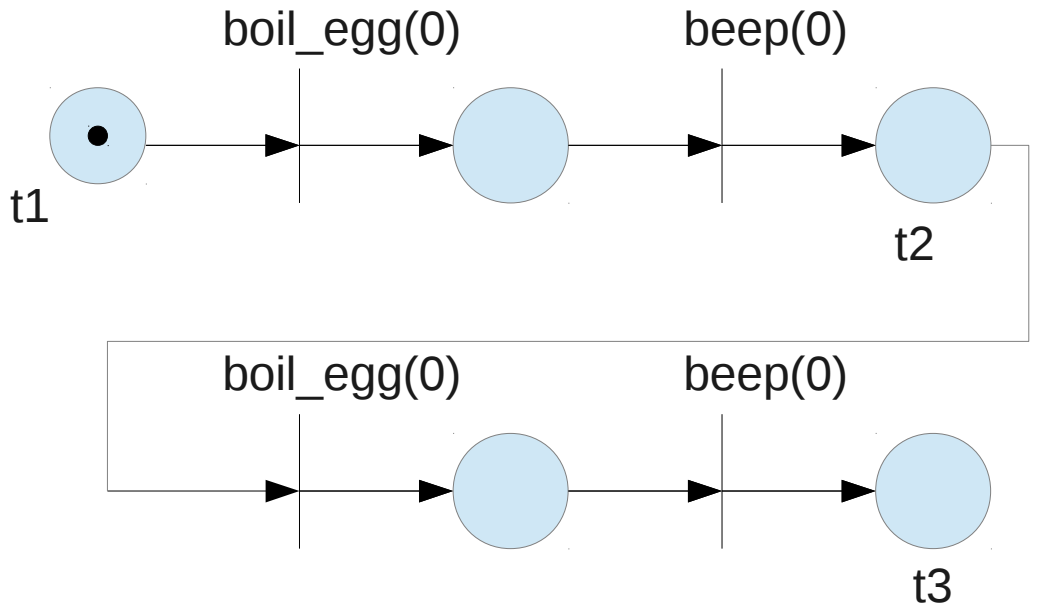
...



Compositionality

cook(Ta, Tc) =
* boil_egg(Ta, 0, Tb)
* beep(Tb, 0, Tc)

{boil_egg(t1, 0, t11), beep(t11, 0, t2),
boil_egg(t2, 0, t22), beep(t22, 0, t3),
token(t1)}

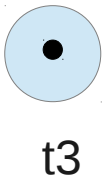


{ **token**(T1)
* cook(T1, T2)
* cook(T2, T3) }

// ...

{ **token**(T3) }

{**token**(t3)}



Interleaving

Assertions: *P*, *Q*, *R* ::=

E
| **emp**
| *P* * *P*
| *P* v *P*
| *bio*(*E*, $\overline{E}$, *E*, *E*)
| **token**(*E*)
| *p*($\overline{E}$)
| **split**(*E*, *E*, *E*)
| **join**(*E*, *E*, *E*)

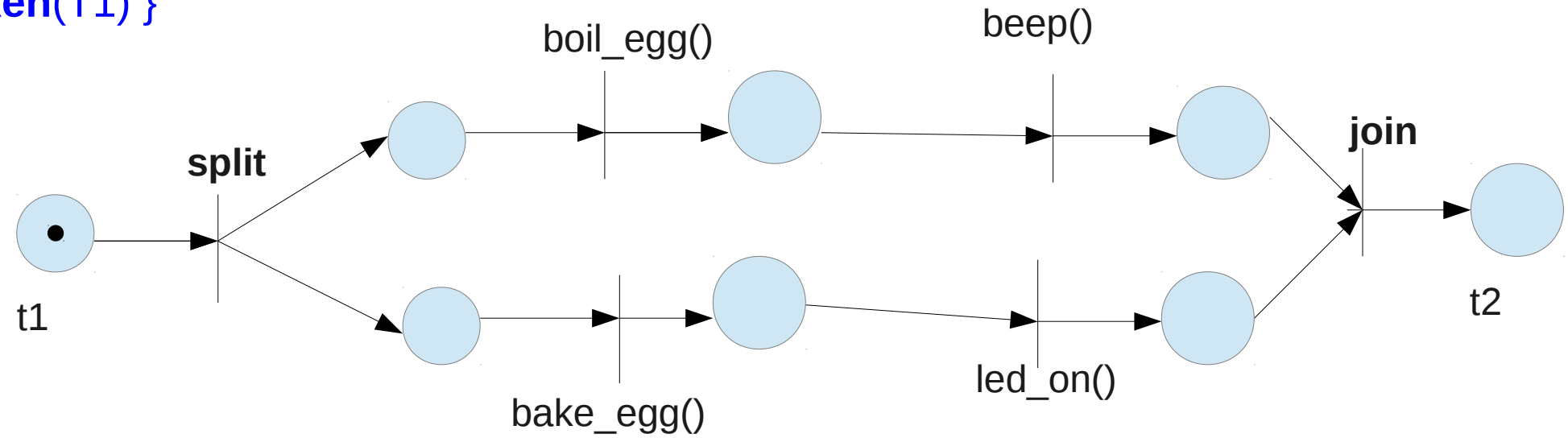
"Split the token in two"
{ **token**(T1) * split(T1, T2, T3) * ... }
{ **token**(T2) * **token**(T3) * ... }

Interleaving

{ **split**(T1, Ta, Tb)
  * cook(Ta, Tj1)
  * bake_egg(Tb, Tb1)
  * led_on(Tb1, Tj2)
* **join**(Tj1, Tj2, T2)
* **token**(T1) }

→

{ **split**(t1, ta, tb),
  boil_egg(ta, ta1), beep(ta2, tj1),
  bake_egg(tb, tb1), led_on(tb1, tj2)
  join(tj1, tj2, t2),
  token(t1) }



Allowed traces:

boil_egg :: bake_egg :: beep :: led_on :: nil
bake_egg :: led_on :: boil_egg :: beep :: nil
...

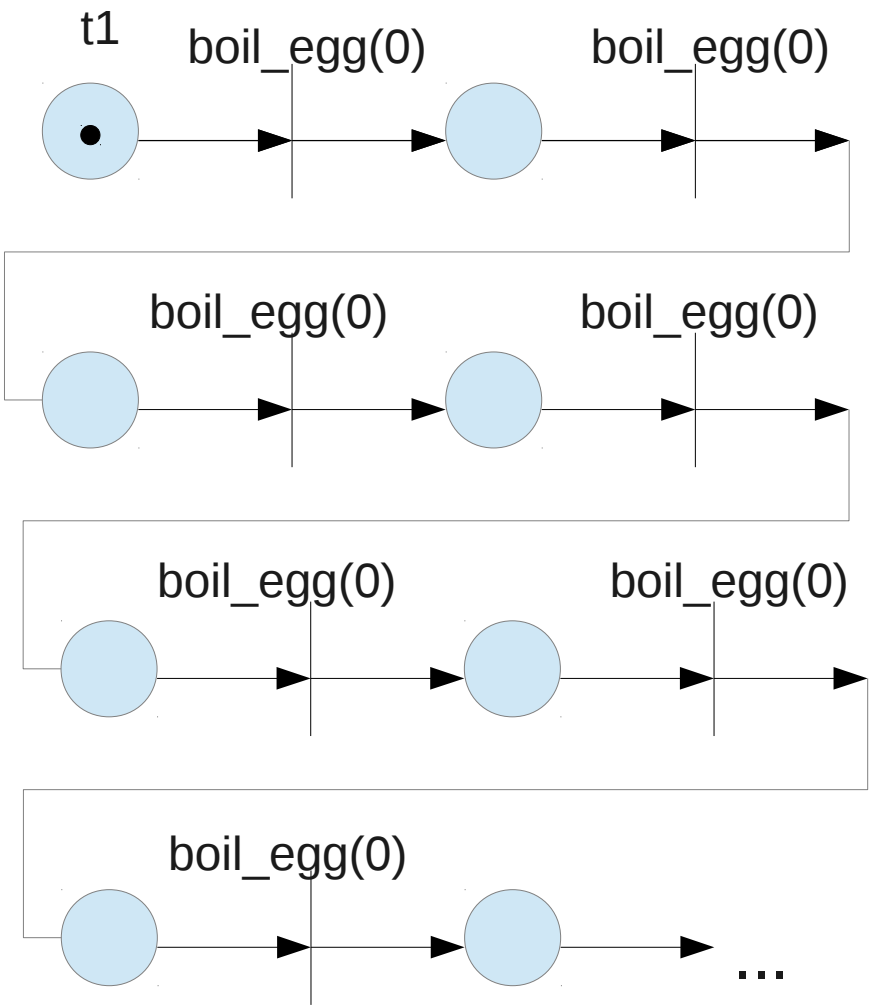
Non-termination

```
keep_on_boiling_eggs(Ta) =
    boil_egg(Ta, 0, Tb)
    * keep_on_boiling_eggs(Tb)
```

```
    { token(T1)
      * keep_on_boiling_eggs(T1)
    }
```

```
       while true do
         // ...
```

```
    { false }
```



Summary

- Precondition: allowed traces / permissions
- Ordering: places and tokens
- Postcondition: obtained token
- Environment's choice: multiple heaps for an assertion
  e.g. read(T1, X, T2) * beep_nb_times(T2, X, T3)
- Programmer's choice: multiple permissions in assertion
  e.g. cook_egg(T1, T2) * boil_egg(T1, T2)
- Easy interleaving: split/join
- Compositionality: abstract predicates
- Non-termination: coinductive predicates
- Combine