Computer-Aided Reasoning for Software

CSE507

Model Checking II

courses.cs.washington.edu/courses/cse507/14au/

Emina Torlak
emina@cs.washington.edu

Today

Today

Last lecture

- Model checking basics

Today

Last lecture

- Model checking basics

Today

- Software model checking with SLAM

Based on lectures by Tom Ball and Sriram K. Rajamani. See the **SLAM project** webpage for details.

Today

Last lecture

- Model checking basics

Today

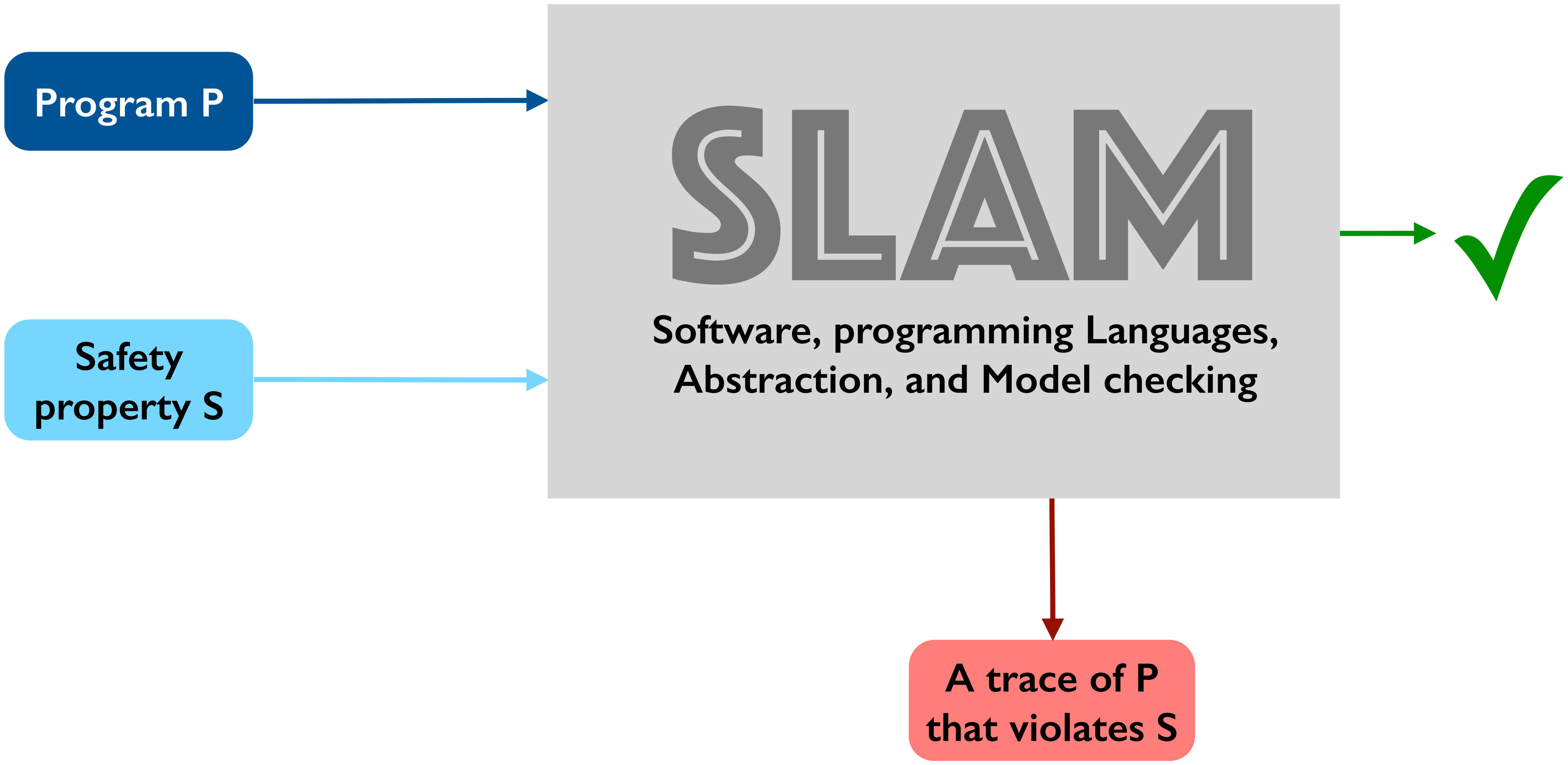
- Software model checking with SLAM

> Based on lectures by Tom Ball and Sriram K. Rajamani. See the **SLAM project** webpage for details.

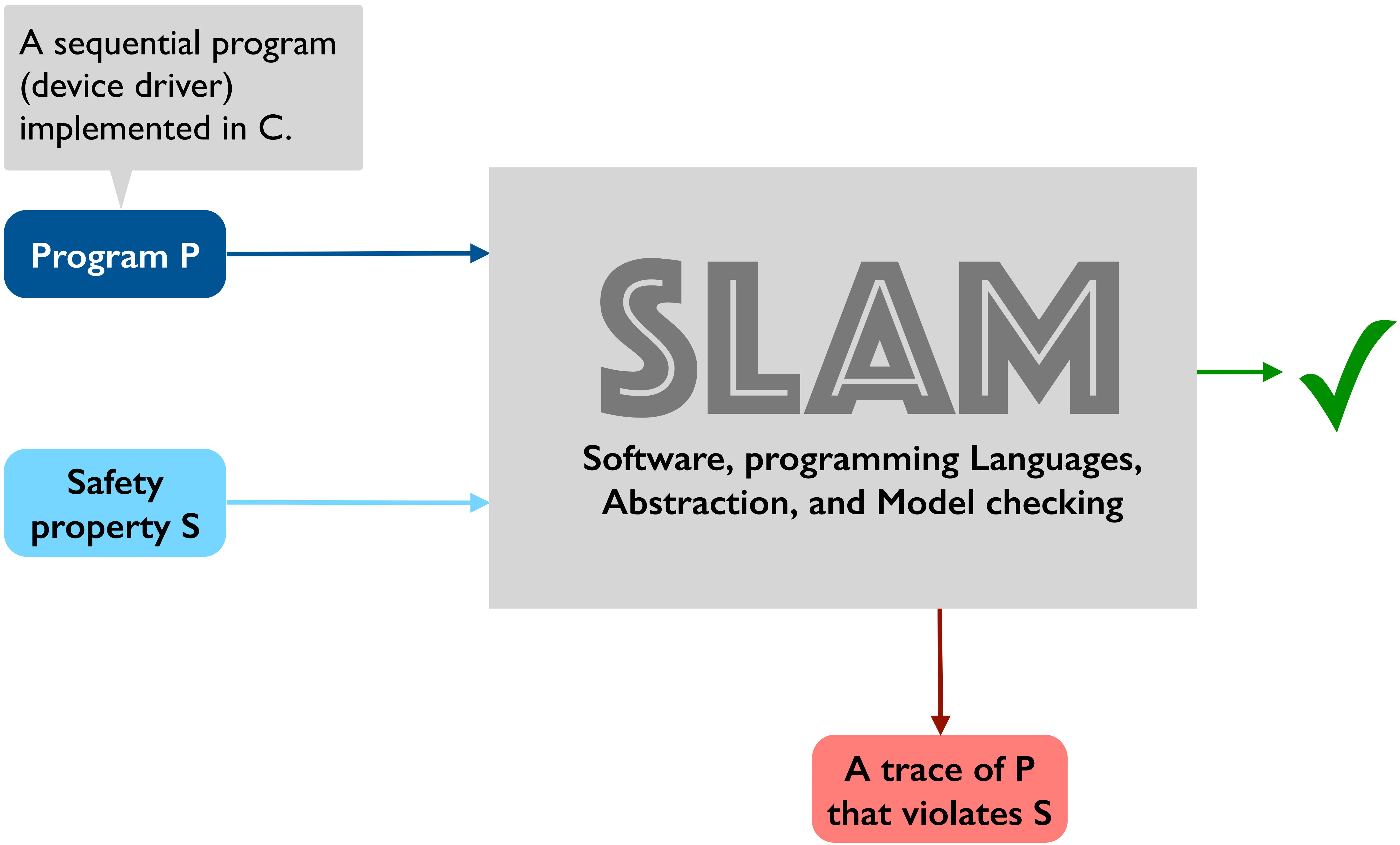
Reminders

- **Homework 3** is due on today at 11pm
- Project demos will be held on Dec 08, 10:30-12:20, in MGH 254

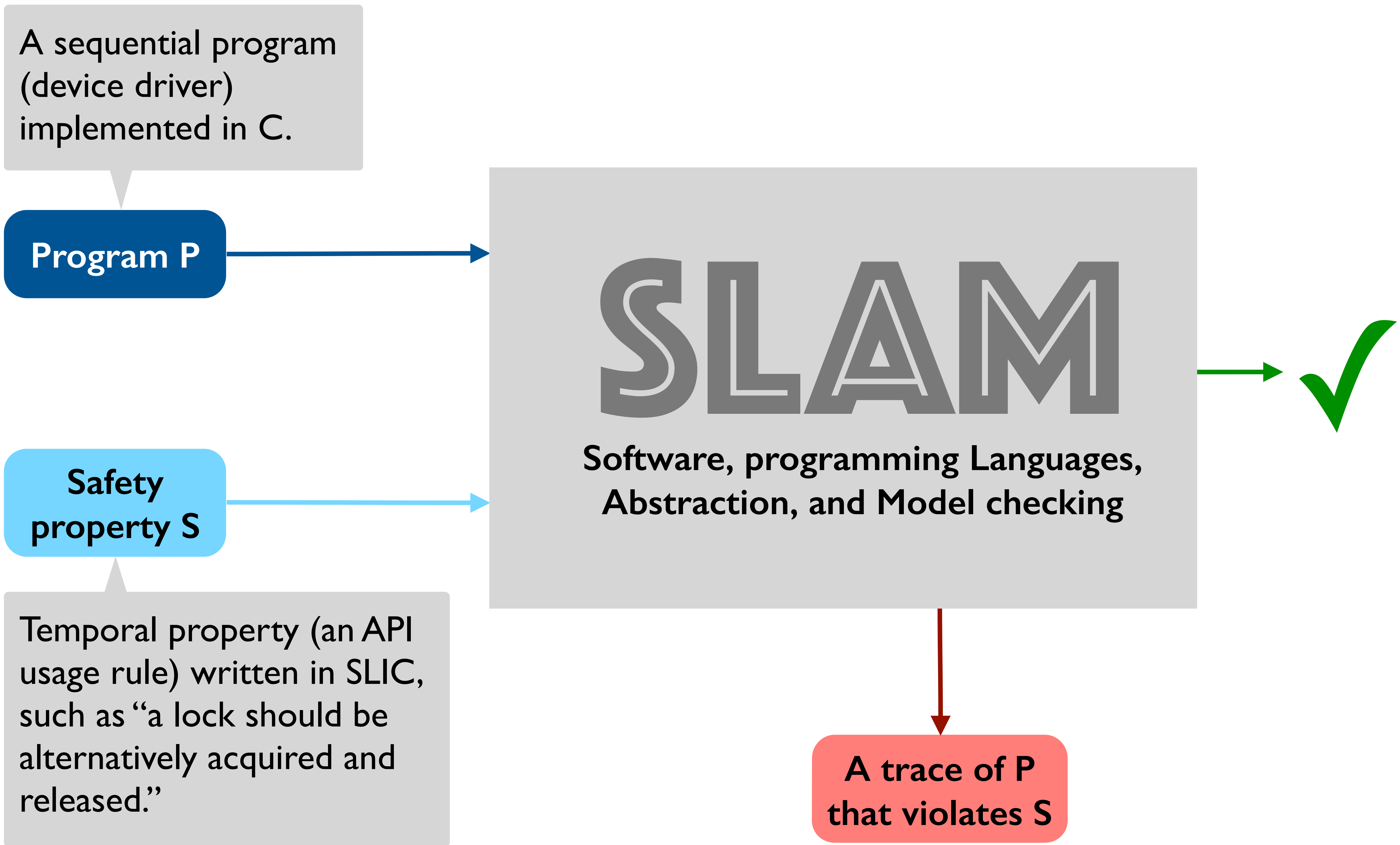
Overview of SLAM



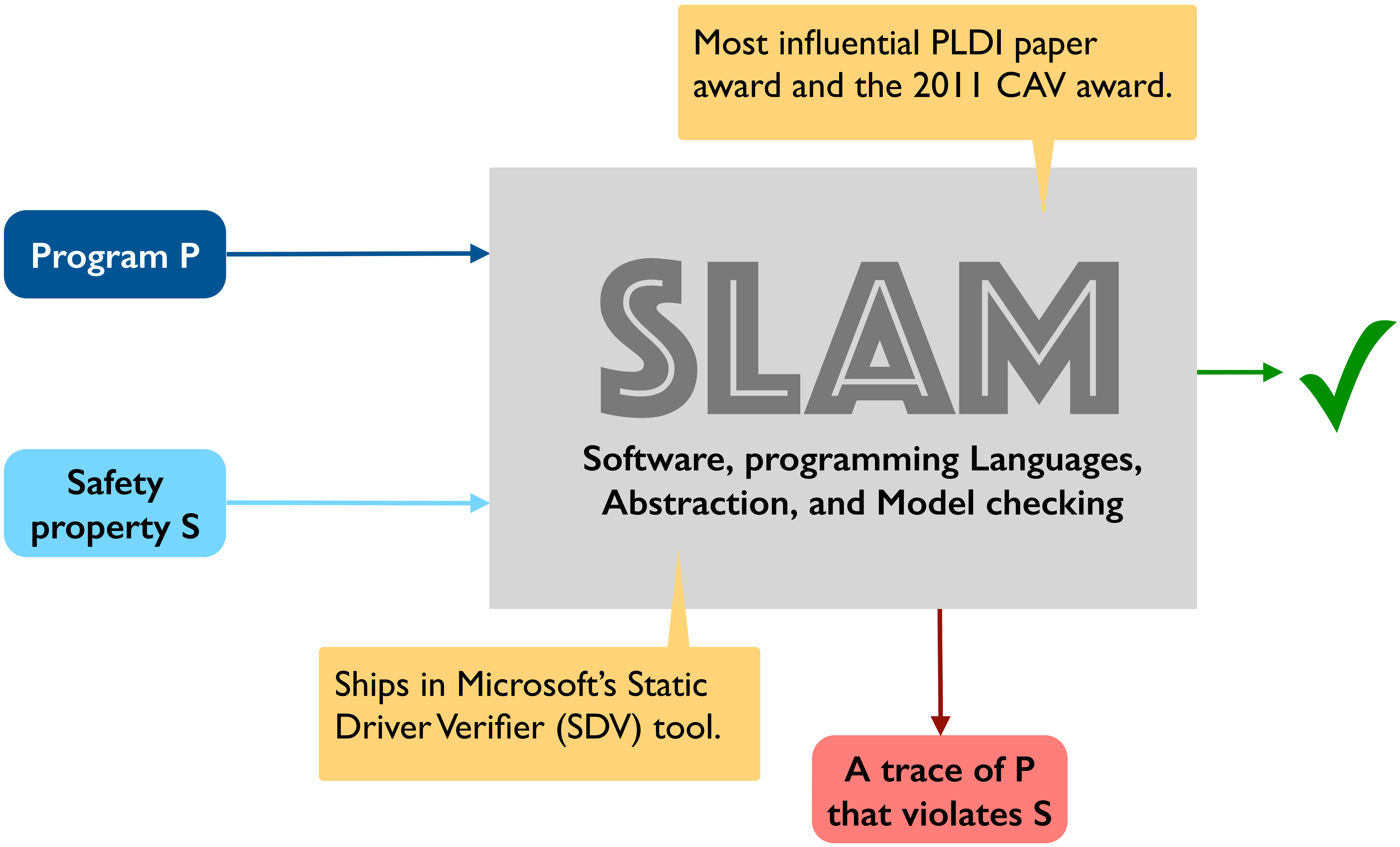
Overview of SLAM



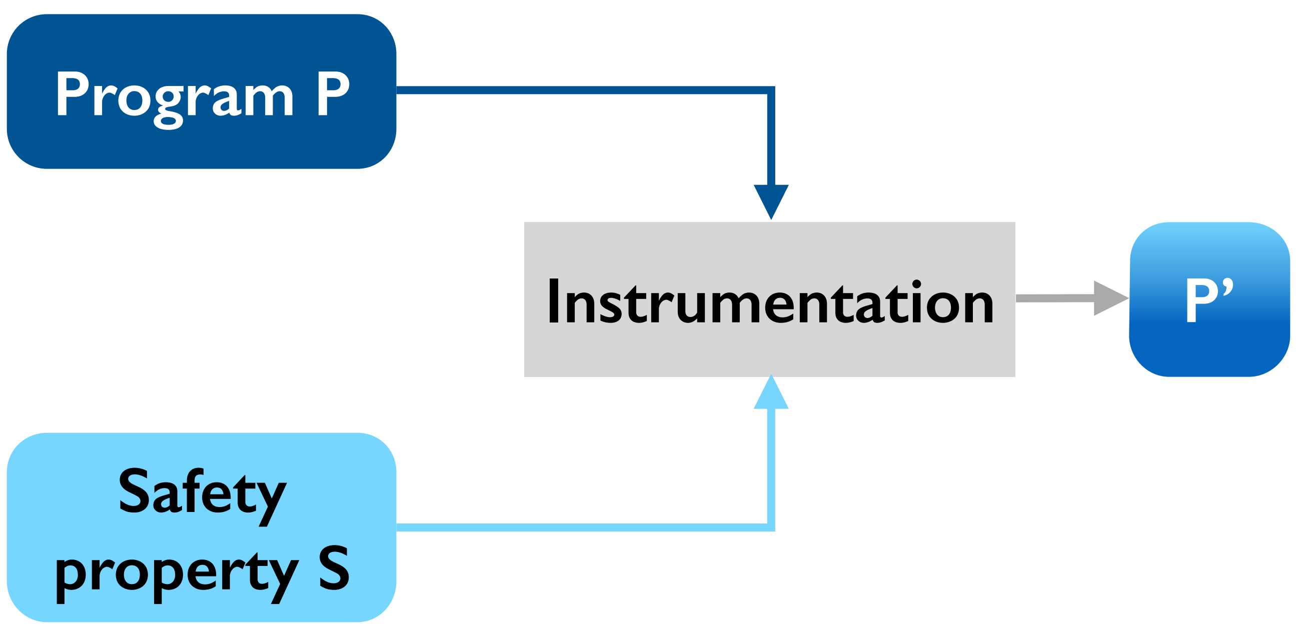
Overview of SLAM



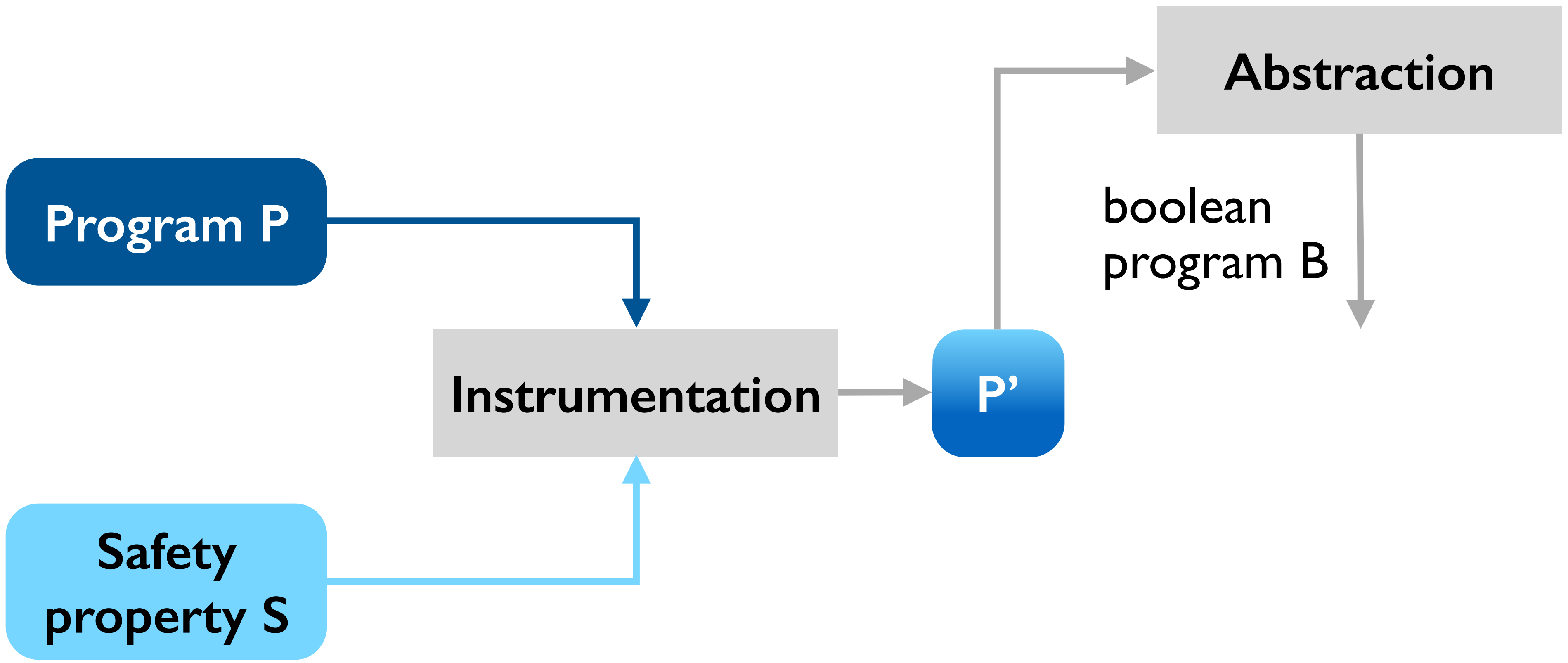
Overview of SLAM



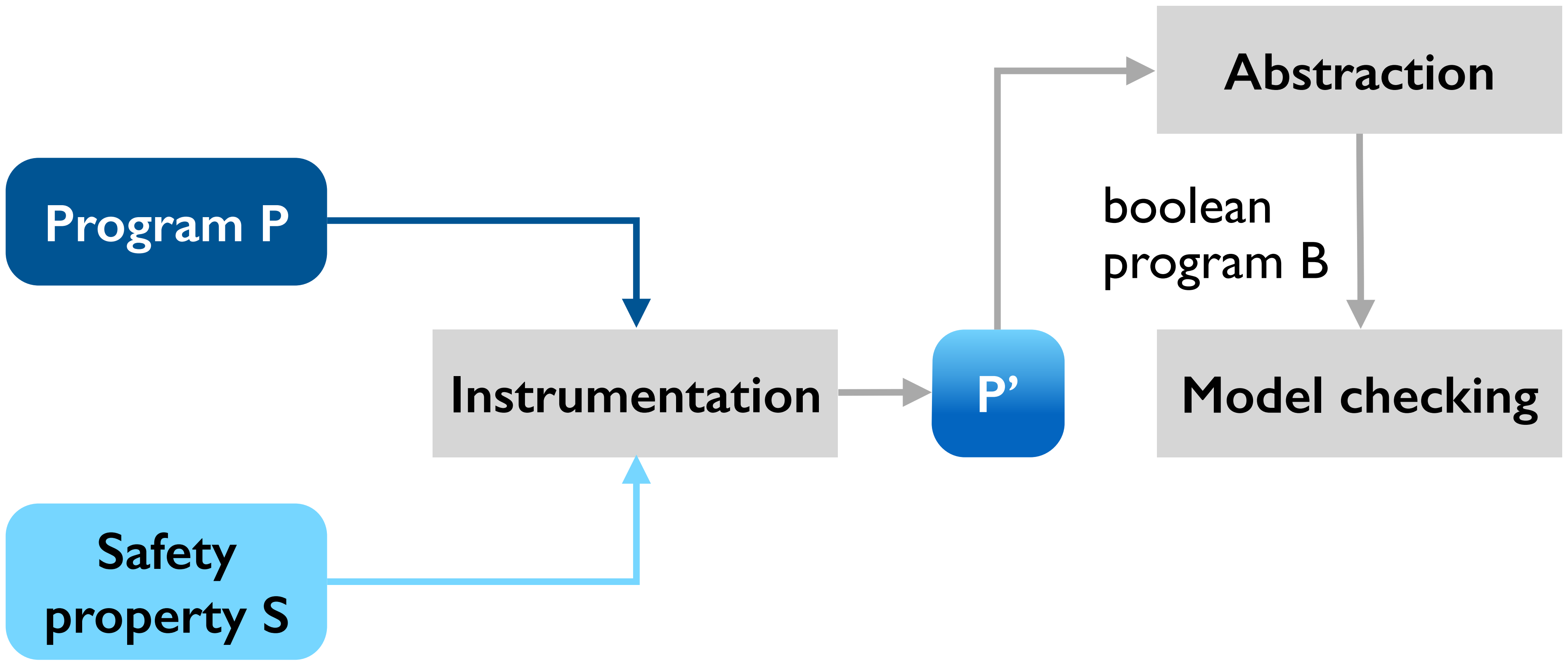
The SLAM process



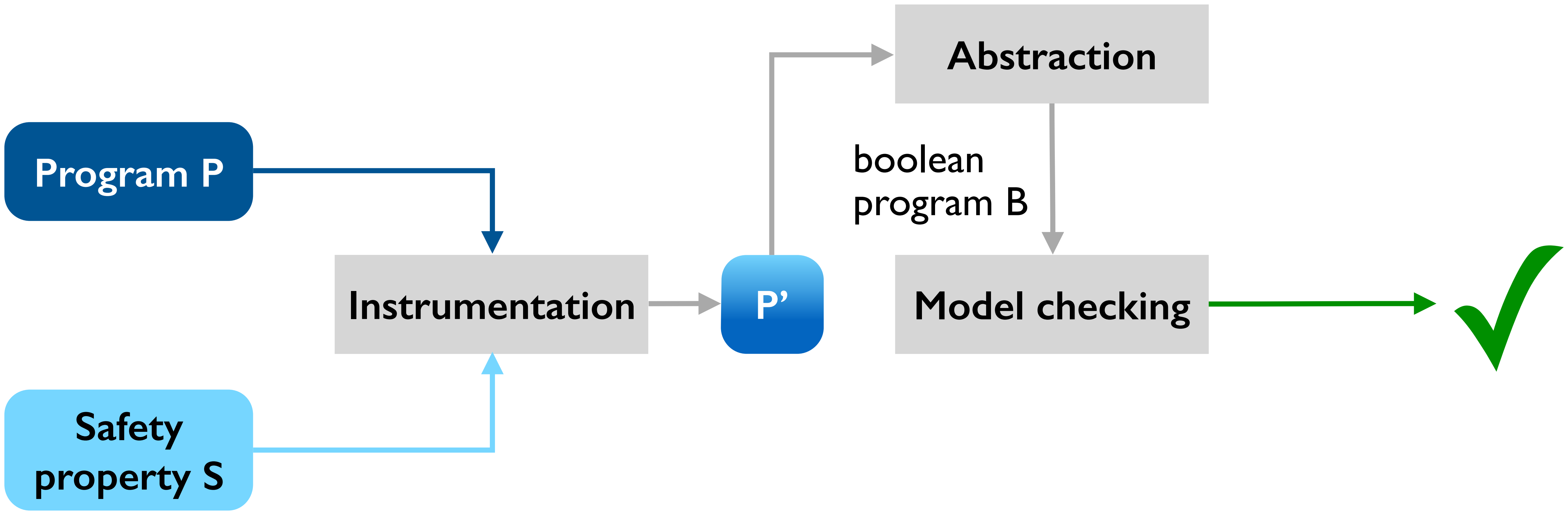
The SLAM process



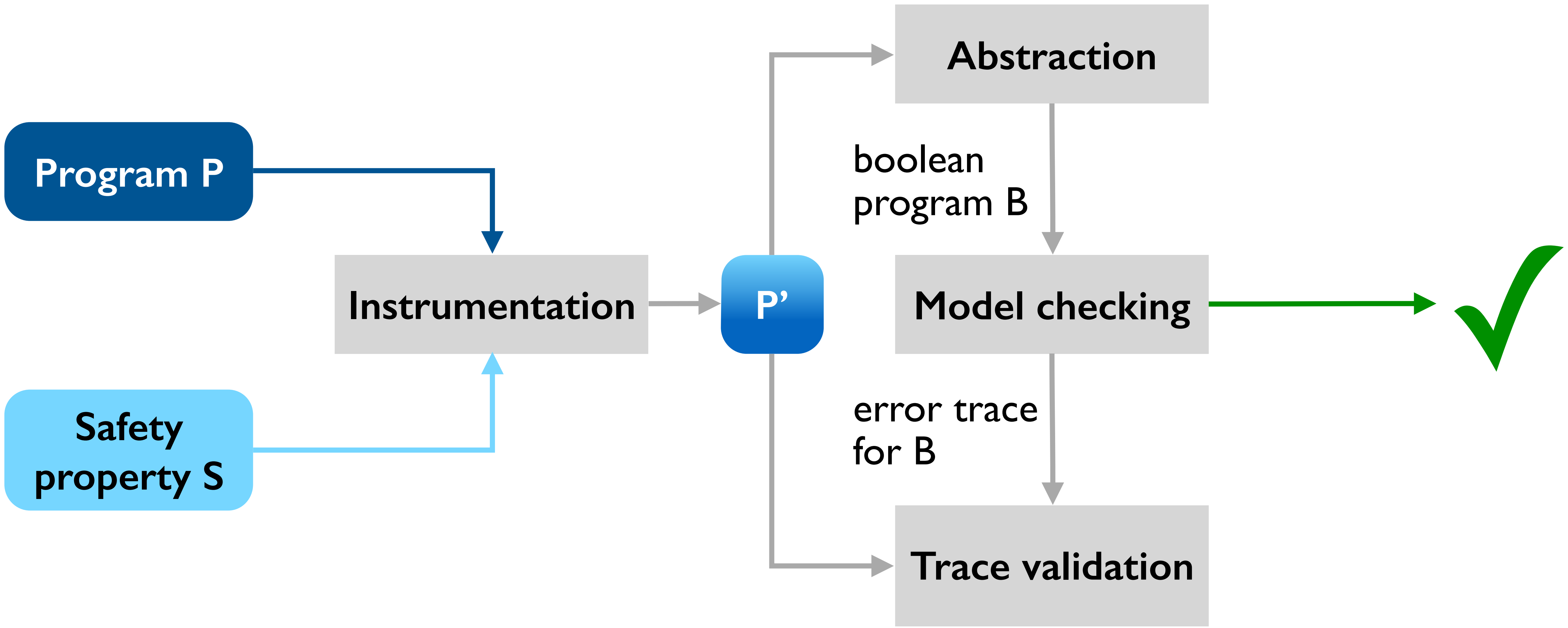
The SLAM process



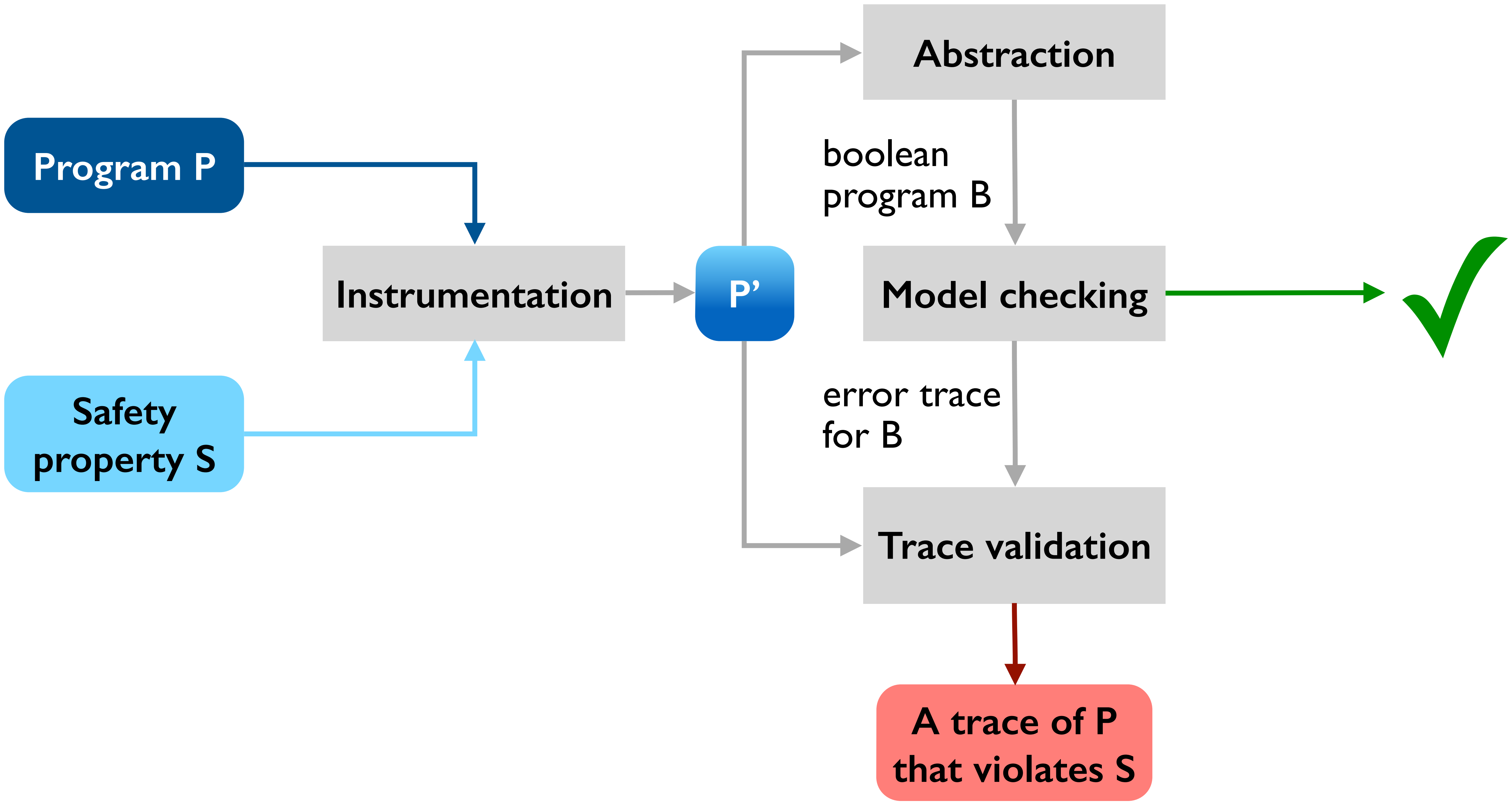
The SLAM process



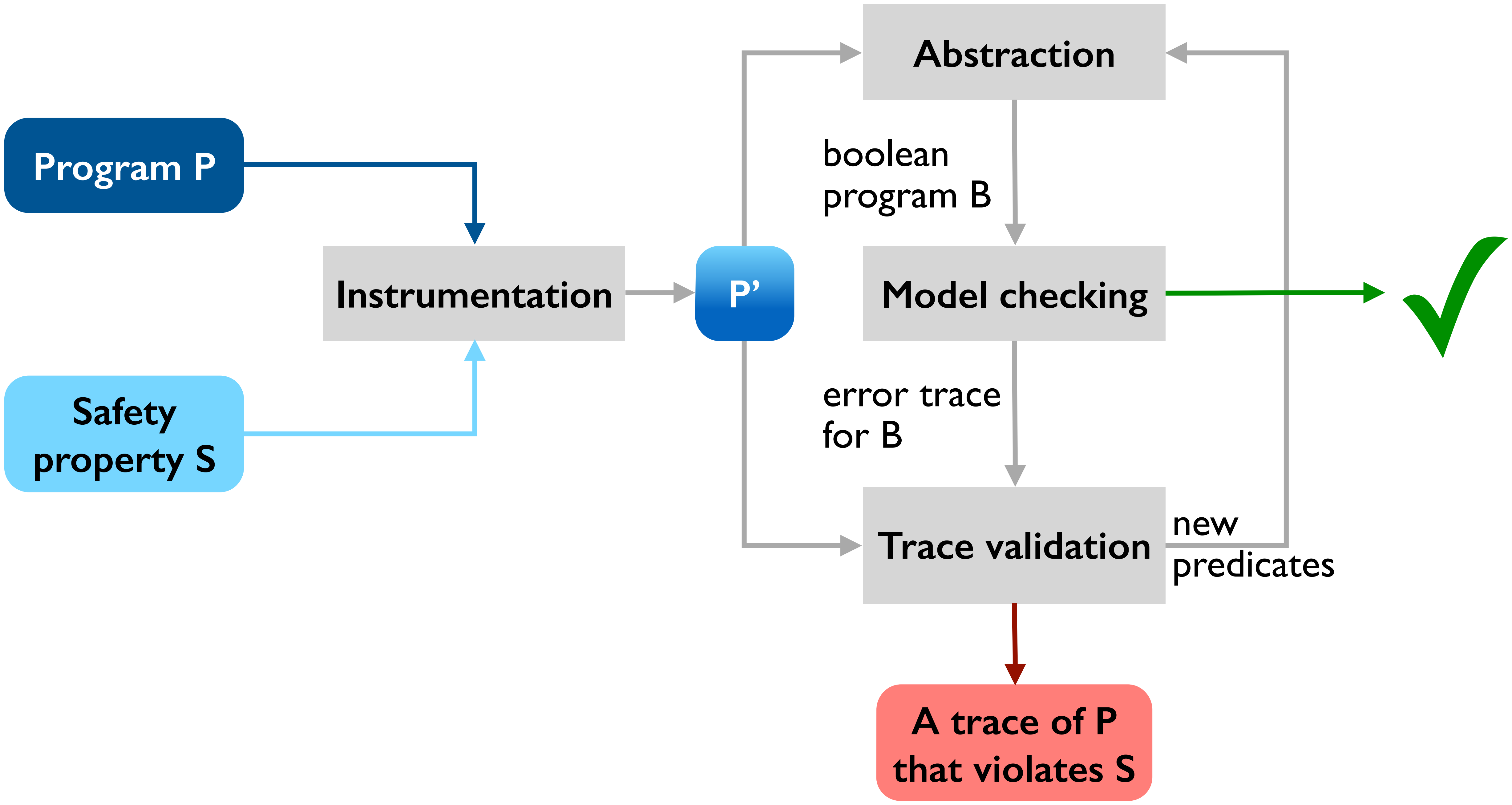
The SLAM process



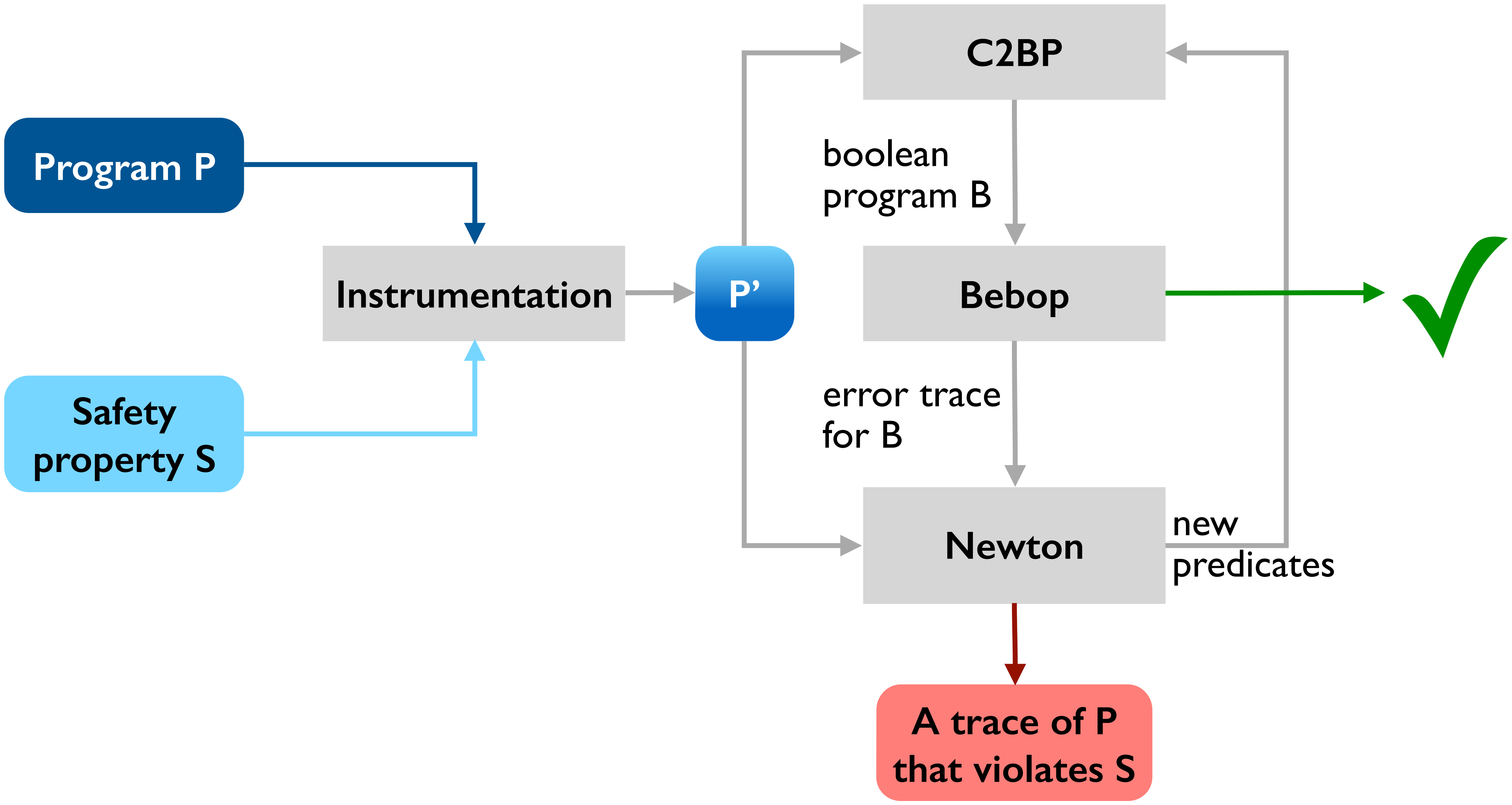
The SLAM process



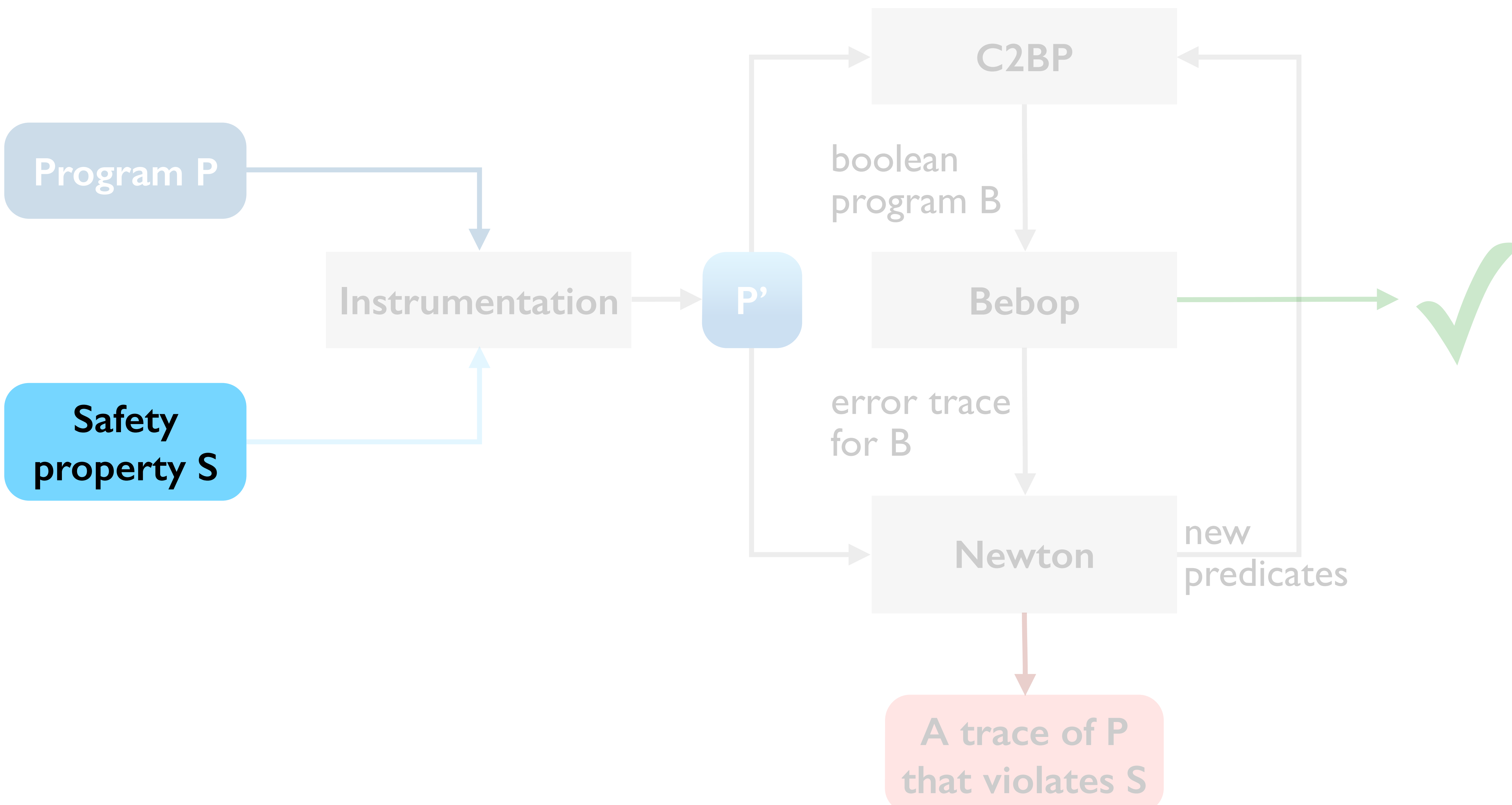
The SLAM process



The SLAM process



The SLAM process: specifying safety properties



Specification Language for Interface Checking

Specification Language for Interface Checking

A finite state language for stating rules for API usage

- Temporal safety properties expressed as *safety automata* that monitor program's execution behavior at the level of function calls and returns.
- Familiar C syntax.

Specification Language for Interface Checking

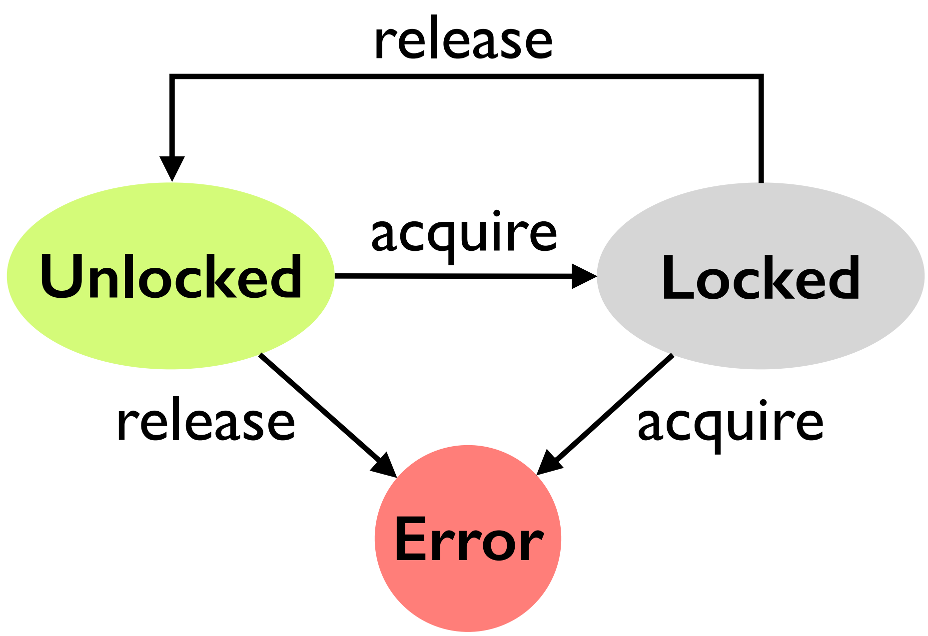
A finite state language for stating rules for API usage

- Temporal safety properties expressed as *safety automata* that monitor program's execution behavior at the level of function calls and returns.
- Familiar C syntax.

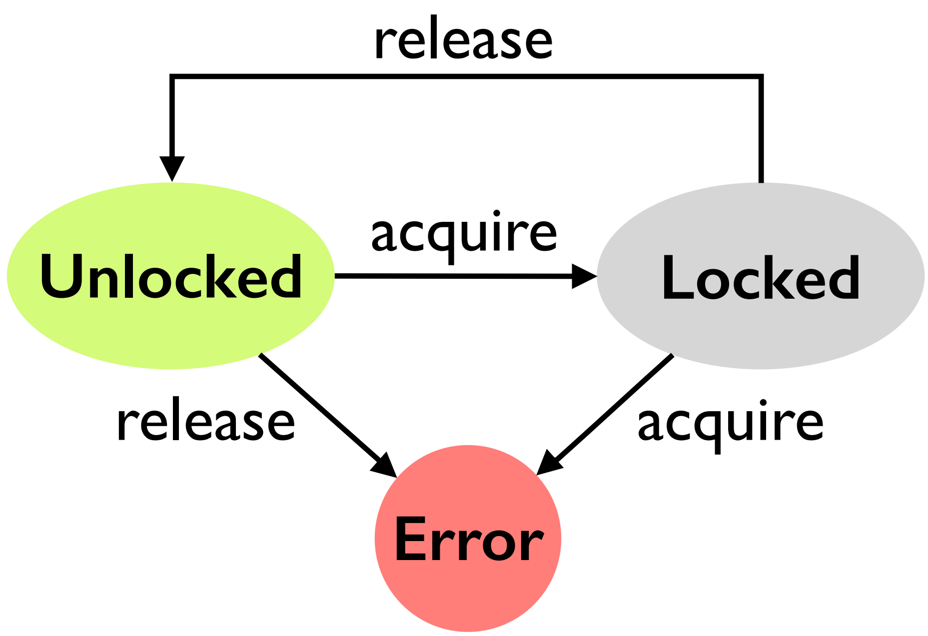
Suitable for control-dominated properties

- E.g., ordering of function calls with associated constraints on data values at the API boundary.

A locking protocol in SLIC



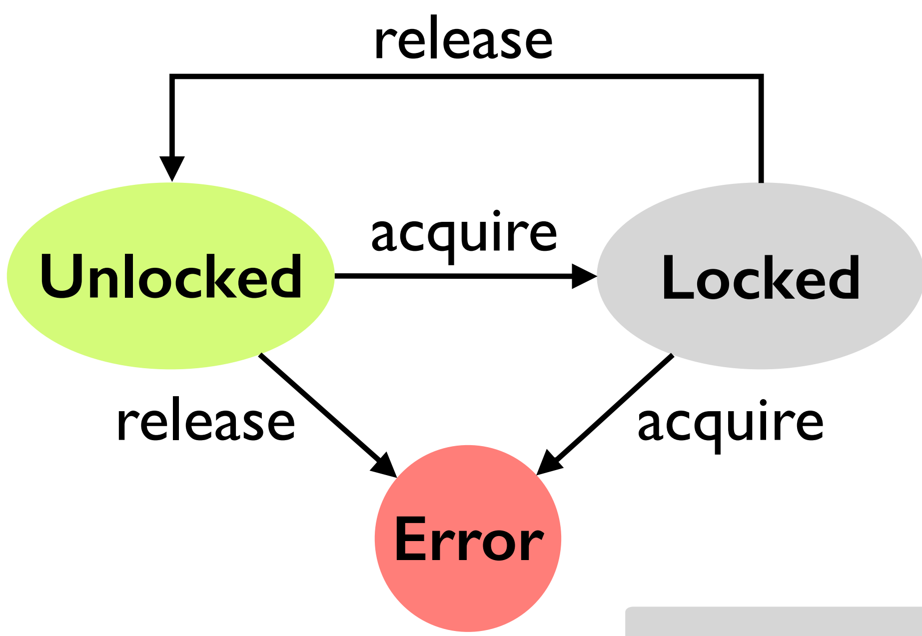
A locking protocol in SLIC




```
state {
  enum {Locked, Unlocked}
   state = Unlocked;
}
```

The global *state structure* defines a static set of *state variables*.

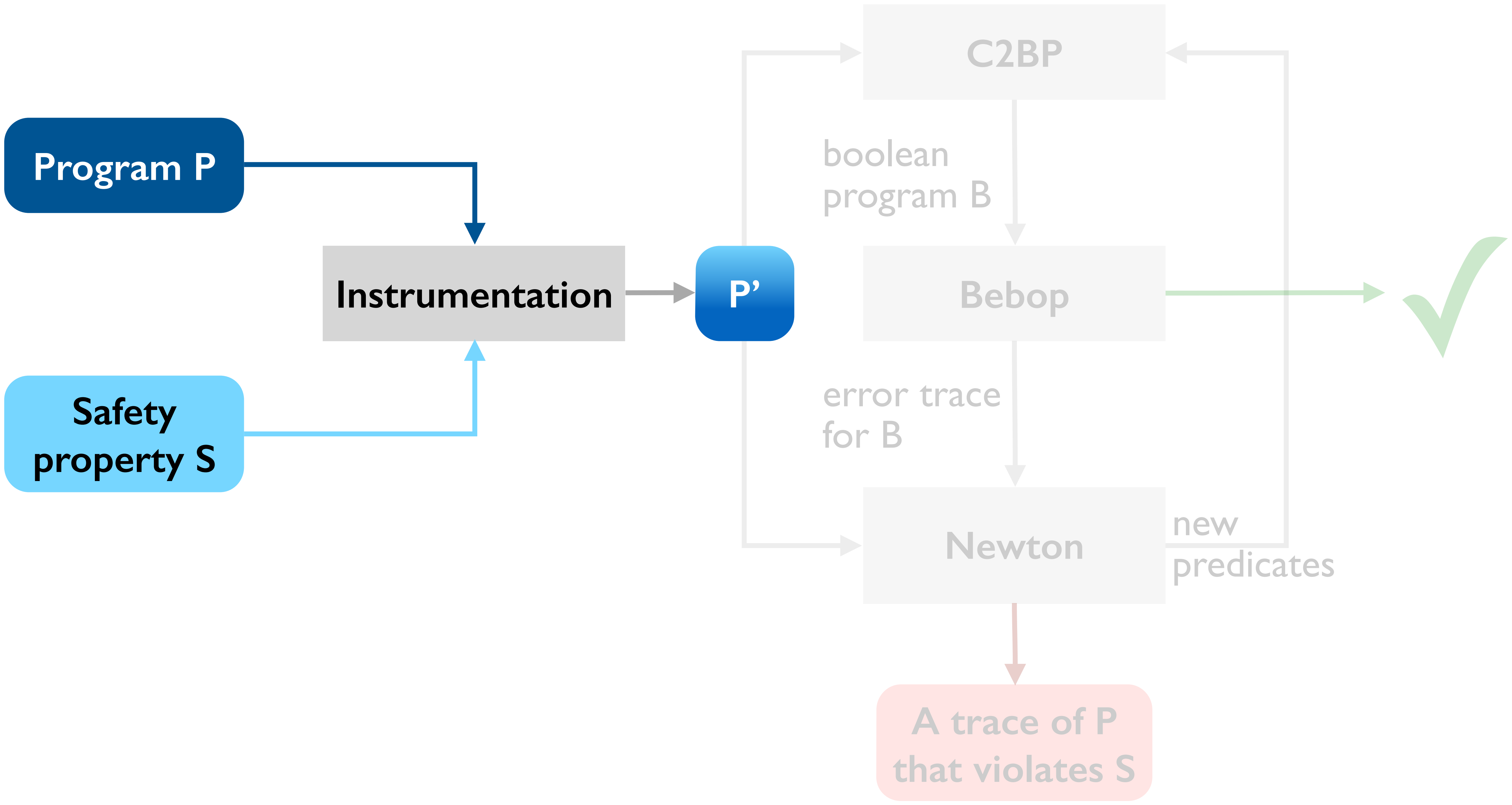
A locking protocol in SLIC



Transfer functions define events and event handlers that describe state transitions on events.

```
state {
  enum {Locked, Unlocked}
   state = Unlocked;
}

KeAcquireSpinLock.return {
  if (state == Locked)
    abort;
  else
    state = Locked;
}

KeReleaseSpinLock.return {
  if (state == Unlocked)
    abort;
  else
    state = Unlocked;
}
```

The SLAM process: instrumentation



Instrumentation by example: 2 steps

```
state {
  enum {Locked, Unlocked}
   state = Unlocked;
}

KeAcquireSpinLock.return {
  if (state == Locked)
    abort;
  else
    state = Locked;
}

KeReleaseSpinLock.return {
  if (state == Unlocked)
    abort;
  else
    state = Unlocked;
}
```

Safety property S

```
void example() {
  do {
   KeAcquireSpinLock();

   nOld = nPackets;

   if (request) {
     request = request->next;
     KeReleaseSpinLock();
     nPackets++;
   }
  } while (nPackets != nOld);

  KeReleaseSpinLock();
}
```

Simplified code for a PCI device driver.

Program P

Step 1: translate the SLIC spec S to C

```
state {
  enum {Locked, Unlocked}
   state = Unlocked;
}

KeAcquireSpinLock.return {
  if (state == Locked)
    abort;
  else
    state = Locked;
}

KeReleaseSpinLock.return {
  if (state == Unlocked)
    abort;
  else
    state = Unlocked;
}
```

Safety property S

```
enum {Locked=0, Unlocked=1}
   state = Unlocked;

void slic_abort() {
  SLIC_ERROR: ;
}

void KeAcquireSpinLock_return {
  if (state == Locked)
    slic_abort();
  else
    state = Locked;
}

void KeReleaseSpinLock_return {
  if (state == Unlocked)
    slic_abort();
  else
    state = Unlocked;
}
```

Distinguished error label.

Step 2: insert calls to SLIC functions into P

```
void example() {
  do {
   KeAcquireSpinLock();

   nOld = nPackets;

   if (request) {
     request = request->next;
     KeReleaseSpinLock();
     nPackets++;
   }
  } while (nPackets != nOld);

  KeReleaseSpinLock();
}
```

Program P

```
void example() {
  do {
   KeAcquireSpinLock();
   KeAcquireSpinLock_return();

   nOld = nPackets;

   if (request) {
     request = request->next;
     KeReleaseSpinLock();
     KeReleaseSpinLock_return();
     nPackets++;
   }
  } while (nPackets != nOld);

  KeReleaseSpinLock();
  KeReleaseSpinLock_return();
}
```

Program P’

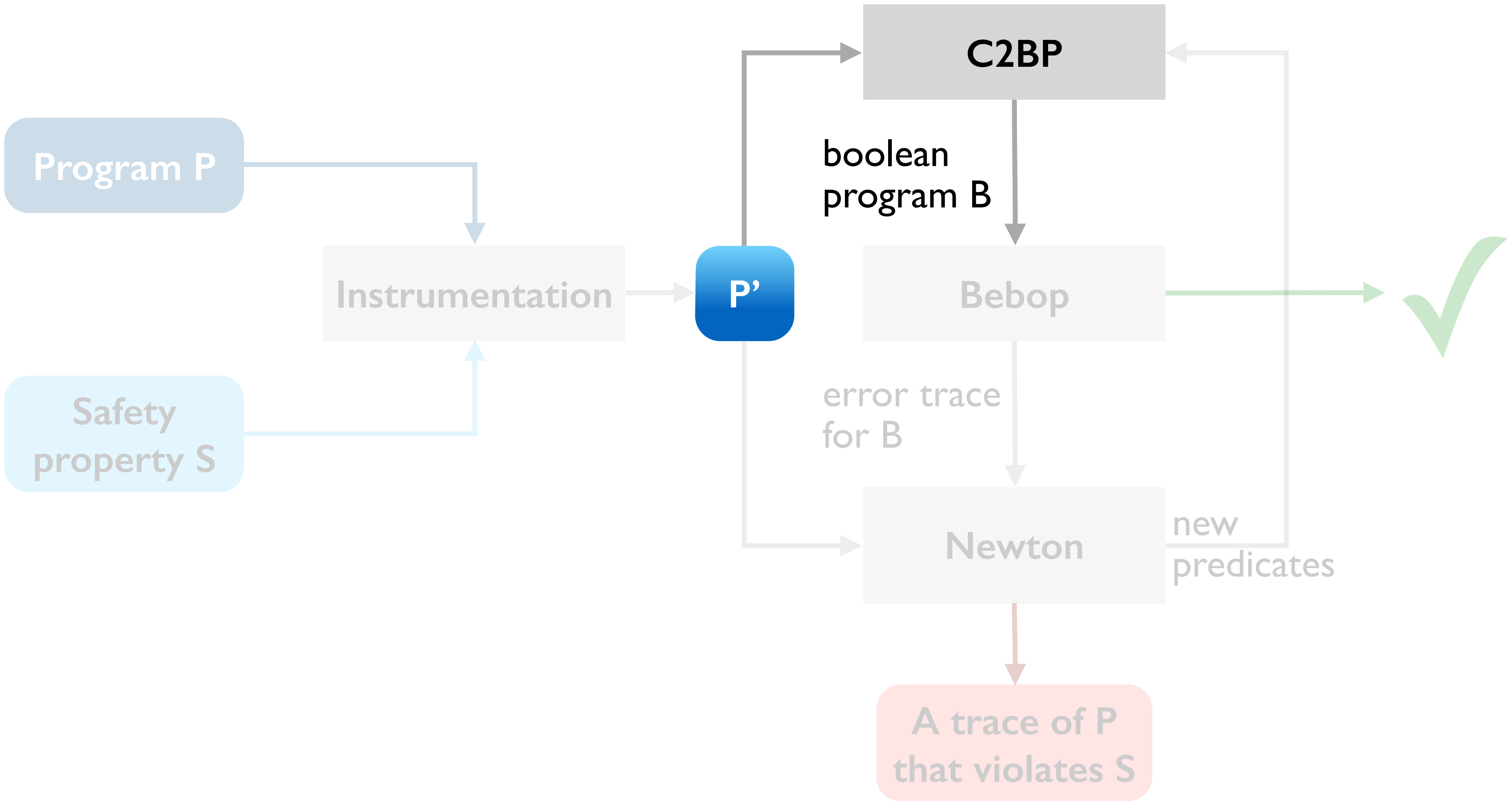
P satisfies S iff SLIC_ERROR is unreachable in P’

```
void example() {
  do {
   KeAcquireSpinLock();
   KeAcquireSpinLock_return();

   nOld = nPackets;

   if (request) {
     request = request->next;
     KeReleaseSpinLock();
     KeReleaseSpinLock_return();
     nPackets++;
   }
  } while (nPackets != nOld);

  KeReleaseSpinLock();
  KeReleaseSpinLock_return();
}
```

```
enum {Locked=0, Unlocked=1}
   state = Unlocked;


void slic_abort() {
  SLIC_ERROR: ;
}

void KeAcquireSpinLock_return {
  if (state == Locked)
    slic_abort();
  else
    state = Locked;
}

void KeReleaseSpinLock_return {
  if (state == Unlocked)
    slic_abort();
  else
    state = Unlocked;
}
```

Program P’

The SLAM process: predicate abstraction



Predicate abstraction of C Programs

Predicate abstraction of C Programs

Given a program P and a finite set E of predicates, C2BP creates a *boolean program* B that is a sound over-approximation of P.

- B has the same control-flow structure as P, but only |E| boolean variables.
- For any path p feasible in P, there is a corresponding feasible path in B.

Predicate abstraction of C Programs

Given a program P and a finite set E of predicates, C2BP creates a *boolean program* B that is a sound over-approximation of P.

- B has the same control-flow structure as P, but only |E| boolean variables.
- For any path p feasible in P, there is a corresponding feasible path in B.

Suitable abstraction for checking control-dominated properties (such as SLIC rules).

- Models control flow in P precisely.
- Models only a few predicates about data relevant to each rule being checked (so limits state explosion).

Predicate abstraction by example: 5+ steps

```
void example() {
  do {
   KeAcquireSpinLock();
   KeAcquireSpinLock_return();

   nOld = nPackets;

   if (request) {
     request = request->next;
     KeReleaseSpinLock();
     KeReleaseSpinLock_return();
     nPackets++;
   }
  } while (nPackets != nOld);

  KeReleaseSpinLock();
  KeReleaseSpinLock_return();
}
```

```
enum {Locked=0, Unlocked=1}
   state = Unlocked;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if (state == Locked)
    slic_abort();
  else
    state = Locked; }

void KeReleaseSpinLock_return {
  if (state == Unlocked)
    slic_abort();
  else
    state = Unlocked; }
```

Program P'

Step 1: extract initial predicates from SLIC rules

```
void example() {
  do {
   KeAcquireSpinLock();
   KeAcquireSpinLock_return();

   nOld = nPackets;

   if (request) {
     request = request->next;
     KeReleaseSpinLock();
     KeReleaseSpinLock_return();
     nPackets++;
   }
  } while (nPackets != nOld);

  KeReleaseSpinLock();
  KeReleaseSpinLock_return();
}
```

```
enum {Locked=0, Unlocked=1}
   state = Unlocked;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if (state == Locked)
    slic_abort();
  else
    state = Locked; }

void KeReleaseSpinLock_return {
  if (state == Unlocked)
    slic_abort();
  else
    state = Unlocked; }
```

Program P'

```
(state == Locked)
(state == Unlocked)
```

Step 2: introduce boolean variables for E

```
void example() {
  do {
   KeAcquireSpinLock();
   KeAcquireSpinLock_return();

   nOld = nPackets;

   if (request) {
     request = request->next;
     KeReleaseSpinLock();
     KeReleaseSpinLock_return();
     nPackets++;
   }
  } while (nPackets != nOld);

  KeReleaseSpinLock();
  KeReleaseSpinLock_return();
}
```

```
b(state==Locked), b(state==Unlocked) := F, T;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if b(state==Locked)
    slic_abort();
  else
    state = Locked;
}

void KeReleaseSpinLock_return {
  if b(state==Unlocked)
    slic_abort();
  else
    state = Unlocked; }
```

```
(state == Locked)
(state == Unlocked)
```

Step 3: skip statements with no effect on E

```
void example() {
  do {
   skip;
   KeAcquireSpinLock_return();

   skip;

   if (request) {
     skip;
     skip;
     KeReleaseSpinLock_return();
     skip;
   }
  } while (nPackets != nOld);

  skip;
  KeReleaseSpinLock_return();
}
```

```
b(state==Locked), b(state==Unlocked) := F, T;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if b(state==Locked)
    slic_abort();
  else
    state = Locked;
}

void KeReleaseSpinLock_return {
  if b(state==Unlocked)
    slic_abort();
  else
    state = Unlocked; }
```

```
(state == Locked)
(state == Unlocked)
```

Step 4: encode the effects of assignments on E

```
void example() {
  do {
   skip;
   KeAcquireSpinLock_return();

   skip;

   if (request) {
     skip;
     skip;
     KeReleaseSpinLock_return();
     skip;
   }
  } while (nPackets != nOld);

  skip;
  KeReleaseSpinLock_return();
}
```

```
b(state==Locked), b(state==Unlocked) := F, T;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if b(state==Locked)
    slic_abort();
  else
    b(state==Locked),
    b(state==Unlocked) := T, F; }

void KeReleaseSpinLock_return {
  if b(state==Unlocked)
    slic_abort();
  else
    b(state==Locked),
    b(state==Unlocked) := F, T; }
```

```
(state == Locked)
(state == Unlocked)
```

Step 5: use non-determinism for conditions

```
void example() {
  do {
   skip;
   KeAcquireSpinLock_return();

   skip;

   if (*) {
     skip;
     skip;
     KeReleaseSpinLock_return();
     skip;
   }
  } while (*);

  skip;
  KeReleaseSpinLock_return();
}
```

```
b(state==Locked), b(state==Unlocked) := F, T;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if b(state==Locked)
    slic_abort();
  else
    b(state==Locked),
    b(state==Unlocked) := T, F; }

void KeReleaseSpinLock_return {
  if b(state==Unlocked)
    slic_abort();
  else
    b(state==Locked),
    b(state==Unlocked) := F, T; }
```

```
(state == Locked)
(state == Unlocked)
```

Step 5: use non-determinism for conditions

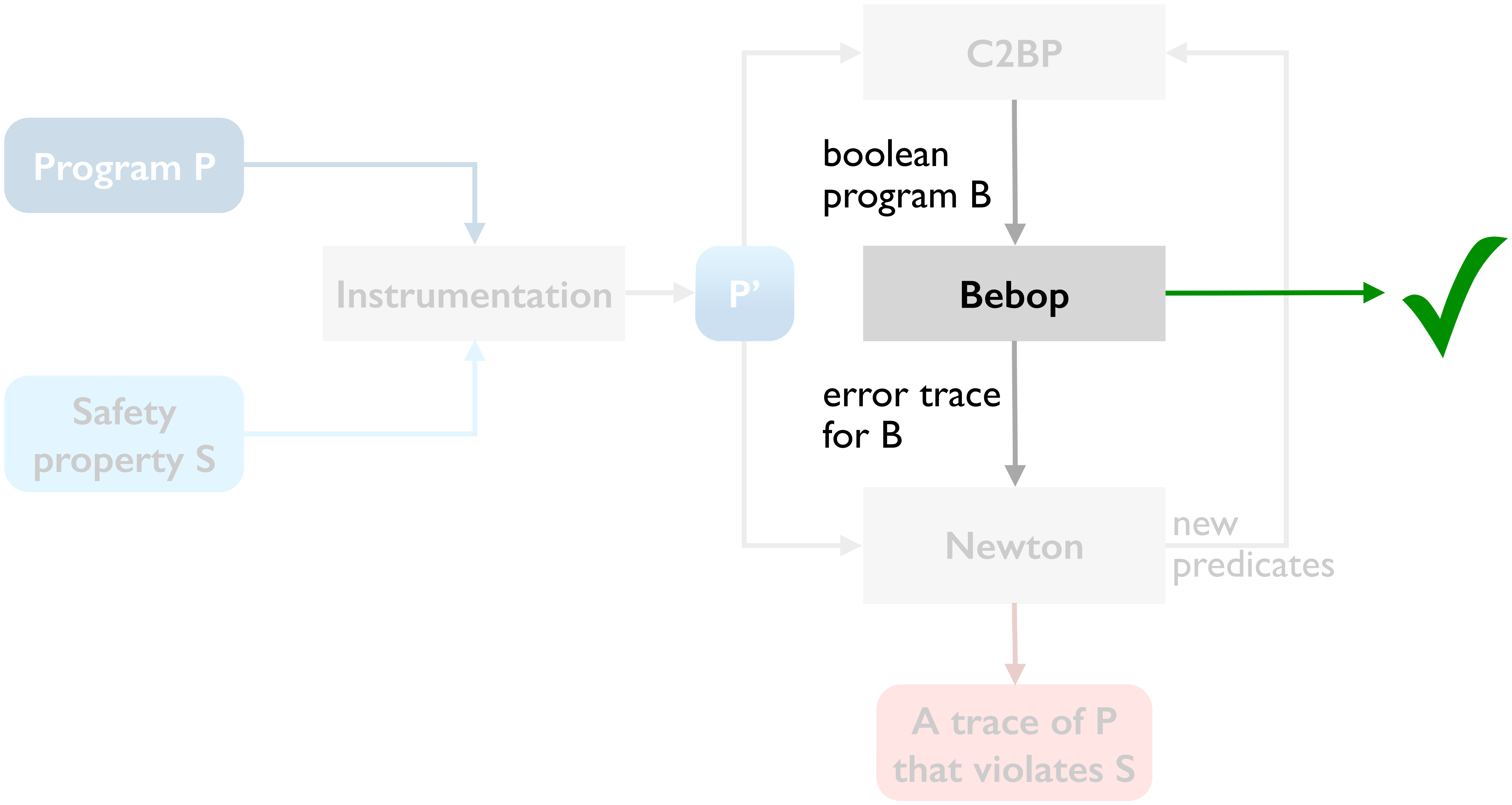
```
void example() {
  do {
   skip;
   KeAcquireSpinLock_return();

   skip;

   if (*) {
     skip;
     skip;
     KeReleaseSpinL
     skip;
   }
  } while (*);

  skip;
  KeReleaseSpinLock_return();
}
```

```
b(state==Locked)

void
  SLIC_ERROR: ; }

();

void

    slic_abort();

    b
    b
```

This is a highly simplified example of predicate abstraction. The process is much more complex in reality. For details, see **Automatic predicate abstraction of C programs**.

```
(state == Locked)
(state == Unlocked)
```

The SLAM process: model checking



Model checking of boolean programs

Model checking of boolean programs

Given a boolean program B and a statement s in B, Bebop determines if s is reachable in B.

- Produces a shortest trace in B (if any) leading to S.

Model checking of boolean programs

Given a boolean program B and a statement s in B, Bebop determines if s is reachable in B.

- Produces a shortest trace in B (if any) leading to S.

Performs symbolic reachability analysis using BDDs.

- Adapts the interprocedural dataflow analysis of Reps, Horwitz and Sagiv (RHS) to decide the reachability of s in B.
- Uses BDDs to represent the procedure summaries in RHS, which are binary relations between sets of states.

Model checking of boolean programs

Given a boolean program B and a statement s in B, Bebop determines if s is reachable in B.

- Produces a shortest trace in B (if any) leading to S.

Performs symbolic reachability analysis using BDDs.

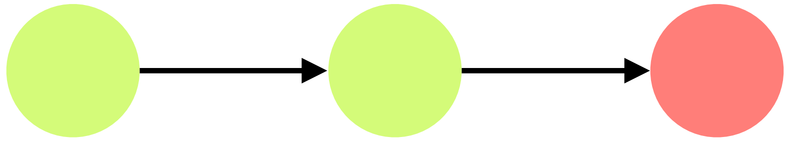
- Adapts the interprocedural dataflow analysis of Reps, Horwitz and Sagiv (RHS) to decide the reachability of s in B.
- Uses BDDs to represent the procedure summaries in RHS, which are binary relations between sets of states.

For details, see Bebop: A Symbolic Model Checker for Boolean Programs**.**

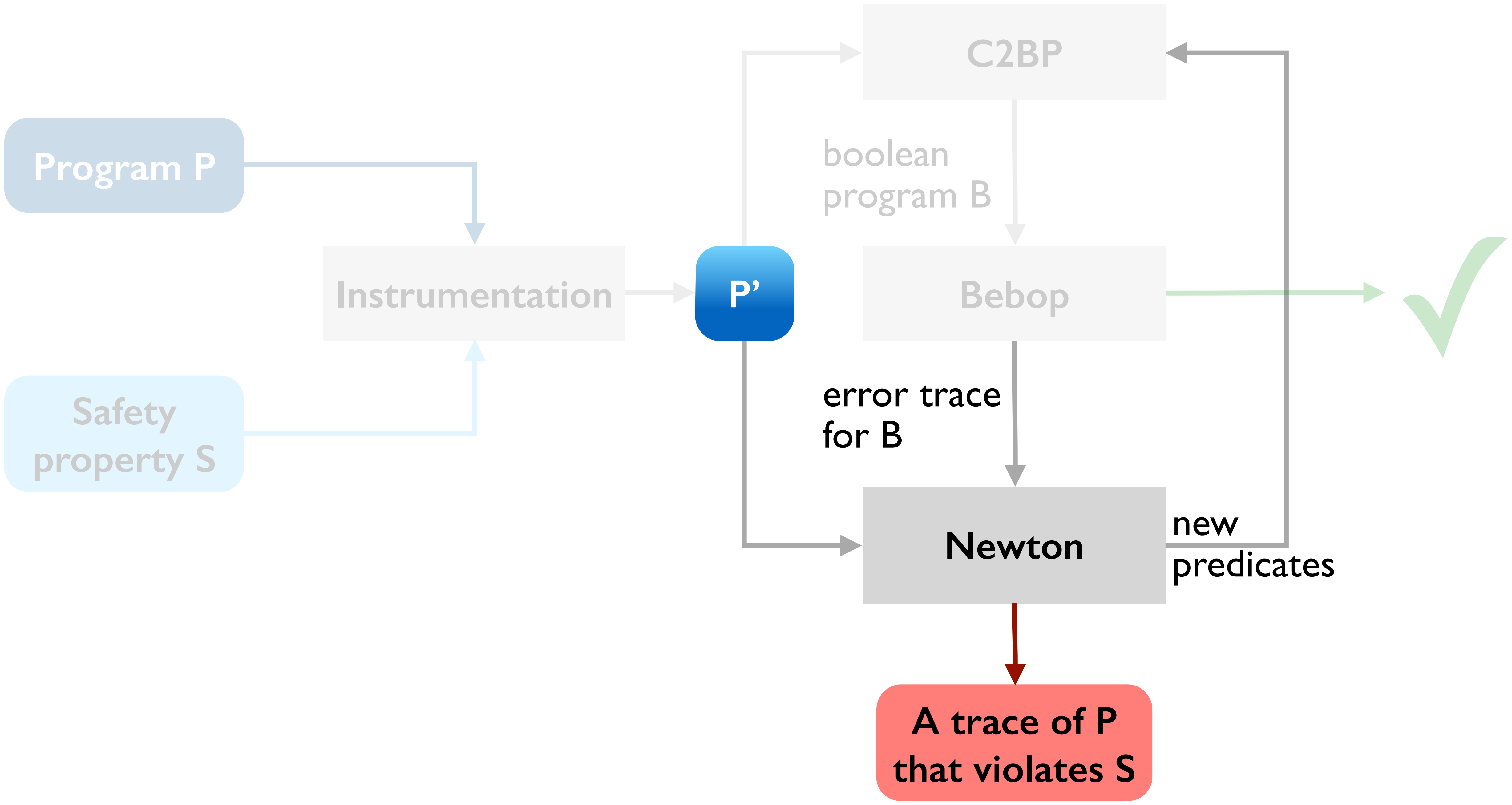
Model checking of the example program

```
void example() {
  do {
   skip;
   KeAcquireSpinLock_return();

   skip;

   if (*) {
     skip;
     skip;
     KeReleaseSpinLock_return();
     skip;
   }
  } while (*);

  skip;
  KeReleaseSpinLock_return();
}
```

```
b(state==Locked), b(state==Unlocked) := F, T;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if b(state==Locked)
    slic_abort();
  else
    b(state==Locked),
    b(state==Unlocked) := T, F; }

void KeReleaseSpinLock_return {
  if b(state==Unlocked)
    slic_abort();
  else
    b(state==Locked),
    b(state==Unlocked) := F, T; }
```



The SLAM process: trace validation



Error trace validation & abstraction refinement

Error trace validation & abstraction refinement

Given a program P' and a candidate error trace, Newton determines if the trace is feasible.

- Uses verification condition generation for feasibility checking.
- If feasible, the error trace corresponds to a real bug.
- If not, returns a small set of predicates that explain why the path is infeasible. Based on greedy minimal unsatisfiable core computation.

Error trace validation & abstraction refinement

Given a program P' and a candidate error trace, Newton determines if the trace is feasible.

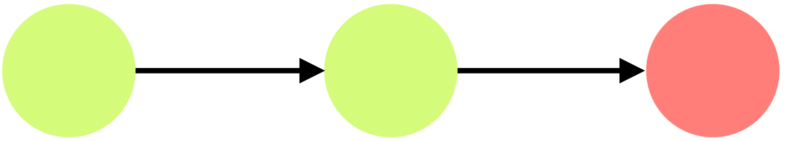
- Uses verification condition generation for feasibility checking.
- If feasible, the error trace corresponds to a real bug.
- If not, returns a small set of predicates that explain why the path is infeasible. Based on greedy minimal unsatisfiable core computation.

For details, see Generating Abstract Explanations of Spurious Counterexamples in C Programs**.**

Validation & refinement for the example

```
void example() {
  do {
   KeAcquireSpinLock();
   KeAcquireSpinLock_return();

   nOld = nPackets;

   if (request) {
     request = request->next;
     KeReleaseSpinLock();
     KeReleaseSpinLock_return();
     nPackets++;
   }
  } while (nPackets != nOld);

  KeReleaseSpinLock();
  KeReleaseSpinLock_return();
}
```

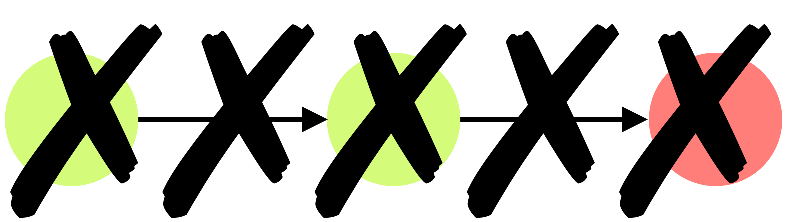
```
enum {Locked=0, Unlocked=1}
   state = Unlocked;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if (state == Locked)
    slic_abort();
  else
    state = Locked; }

void KeReleaseSpinLock_return {
  if (state == Unlocked)
    slic_abort();
  else
    state = Unlocked; }
```



```
(state == Locked)
(state == Unlocked)
```

Validation & refinement for the example

```
void example() {
  do {
   KeAcquireSpinLock();
   KeAcquireSpinLock_return();

   nOld = nPackets;

   if (request) {
     request = request->next;
     KeReleaseSpinLock();
     KeReleaseSpinLock_return();
     nPackets++;
   }
  } while (nPackets != nOld);

  KeReleaseSpinLock();
  KeReleaseSpinLock_return();
}
```



```
enum {Locked=0, Unlocked=1}
   state = Unlocked;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if (state == Locked)
    slic_abort();
  else
    state = Locked; }

void KeReleaseSpinLock_return {
  if (state == Unlocked)
    slic_abort();
  else
    state = Unlocked; }
```

```
(nPackets == nOld)
(state == Locked)
(state == Unlocked)
```

Back to C2BP and Bebop ...

```
void example() {
  do {
   skip;
   KeAcquireSpinLock_return();

   b(nOld==nPackets) := T;

   if (*) {
     skip;
     skip;
     KeReleaseSpinLock_return();
     b(nOld==nPackets) :=
      b(nOld==nPackets) ? F : *;
   }
  } while (!b(nOld==nPackets));

  skip;
  KeReleaseSpinLock_return();
}
```

```
b(state==Locked), b(state==Unlocked) := F, T;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if b(state==Locked)
    slic_abort();
  else
    b(state==Locked),
    b(state==Unlocked) := T, F; }

void KeReleaseSpinLock_return {
  if b(state==Unlocked)
    slic_abort();
  else
    b(state==Locked),
    b(state==Unlocked) := F, T; }
```

```
(nPackets == nOld)
(state == Locked)
(state == Unlocked)
```

Back to C2BP and Bebop …

```
void example() {
  do {
   skip;
   KeAcquireSpinLock_return();

   b(nOld==nPackets) := T;

   if (*) {
     skip;
     skip;
     KeReleaseSpinLock_return();
     b(nOld==nPackets) :=
      b(nOld==nPackets) ? F : *;
   }
  } while (!b(nOld==nPackets));

  skip;
  KeReleaseSpinLock_return();
}
```

✓

```
b(state==Locked), b(state==Unlocked) := F, T;

void slic_abort() {
  SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
  if b(state==Locked)
    slic_abort();
  else
    b(state==Locked),
    b(state==Unlocked) := T, F; }

void KeReleaseSpinLock_return {
  if b(state==Unlocked)
    slic_abort();
  else
    b(state==Locked),
    b(state==Unlocked) := F, T; }
```

Summary

Today

- Software model checking with SLAM
  - Predicate abstraction of C programs
  - Model checking of boolean programs
  - Trace validation and abstraction refinement

Next lecture

- Guest lecture by Zach Tatlock!
- Verifying compiler optimizations with SMT solvers