Verifying real-world systems

Formal verification of a snapshot-consistent flash translation layer

Yun-Sheng Chang

August 20, 2020

Institute of Information Science, Academia Sinica

Why formal verification?

> ”Can we instead build on the idea that a computer program is a mathematical object that can be understood through logic?”
>
> — Leslie Lamport, The Future of Computing: Logic or Biology

Advantages against conventional approach

- No ambiguity (compared with specification written in natural languages)
- Correct with respect to all allowed inputs and states (compared with testing)

Categorizing verification techniques

Extract-based interactive verification

- Programs, properties and proofs are all written in some functional language which is itself also a consistent logic (e.g., Agda and Coq)
- Examples: CompCert and FSCQ

Import-based interactive verification

- Programs written in some imperative languages (e.g., C and Java); properties and proofs are written in another first-order or higher-order logic
- Examples: seL4 and CertiKOS

Import-based automatic verification

- Programs written in some imperative languages (e.g., C and Java); properties are often written in first-order logic; proofs are done automatically
- Examples: Yxv6 and Ironfleet

Outline

Verifying programs and systems with automatic verification tools

- Definitions of program correctness
- Automatic verification techniques

Formal verification of a snapshot-consistent flash translation layer

Absence of undefined behavior

```
int A[4];
uintptr_t i;
int8_t x;
uint8_t y;

void dummy(void) {
    A[i] = 10;
    x++;
    y++;
}
```

Given what condition does this program have *defined* behaviors (e.g., array out-of-bound access and integer overflow)?

Absence of undefined behavior

```
int A[4];
uintptr_t i;
int8_t x;
uint8_t y;

void dummy(void) {
    A[i] = 10;
    x++;
    y++;
}
```

Given what condition does this program have *defined* behaviors (e.g., array out-of-bound access and integer overflow)?

$$i < 4 \wedge x < 127$$

How about $y < 255$?

Absence of undefined behavior

```
int A[4];
uintptr_t i;
int8_t x;
uint8_t y;

void dummy(void) {
    A[i] = 10;
    x++;
    y++;
}
```

Given what condition does this program have *defined* behaviors (e.g., array out-of-bound access and integer overflow)?

$$i < 4 \wedge x < 127$$

How about $y < 255$? C language requires unsigned integer to ”wrap around”.

Absence of undefined behavior

```
int A[4];
uintptr_t i;
int8_t x;
uint8_t y;

void dummy(void) {
    A[i] = 10;
    x++;
    y++;
}
```

Given what condition does this program have *defined* behaviors (e.g., array out-of-bound access and integer overflow)?

$$i < 4 \land x < 127$$

How about $y < 255$? C language requires unsigned integer to ”wrap around”.

We often call this **precondition**.

But this program is not so useful. We don’t know what this program is *intended to do*.

Algorithmic property

```
int A[4];
uintptr_t i, x;

void findmax(void) {
    x = 0;
    for (i = 0; i < 4; i++)
        if (A[i] >= A[x])
            x = i;
}
```

This program assigns to x the position of the maximal element in array A. We can express this sentence formally:

$$\forall i.i < 4 \implies A[i] \leq A[x]$$

We often call this **postcondition**.

Algorithmic property

```
int A[4];
uintptr_t i, x;

void findmax(void) {
    x = 0;
    for (i = 0; i < 4; i++)
        if (A[i] >= A[x])
            x = i;
}
```

This program assigns to x the position of the maximal element in array A. We can express this sentence formally:

$$\forall i.i < 4 \implies A[i] \leq A[x]$$

We often call this **postcondition**.

In fact, $A[x]$ is the *last* maximal element. Again, formally:

$$(\forall i.i < 4 \implies A[i] \leq A[x]) \wedge (\forall j.x < j < 4 \implies A[j] < A[x])$$

Algorithmic property

```
int A[4];
uintptr_t i, x;

void findmax(void) {
    x = 0;
    for (i = 0; i < 4; i++)
        if (A[i] >= A[x])
            x = i;
}
```

If we know that ”A does not contain repeated elements”, then we can have a stronger postcondition:

$$\forall i.i < 4 \land i \neq x \implies A[i] < A[x]$$

Algorithmic property

```
int A[4];
uintptr_t i, x;

void findmax(void) {
    x = 0;
    for (i = 0; i < 4; i++)
        if (A[i] >= A[x])
            x = i;
}
```

If we know that "A does not contain repeated elements", then we can have a stronger postcondition:

$$\forall i.i < 4 \land i \neq x \implies A[i] < A[x]$$

In this case, "A does not contain repeated elements" is our precondition, and can be expressed formally:

$$\forall i, j.i < 4 \land j < 4 \land i \neq j \implies A[i] \neq A[j]$$

Preservation of invariant

Up to now, we have only seen programs that run to their ends. But a system is more like an infinite loop.

Let's consider a system to be a **state machine** with a set of **states** and a set of **operations** that bring one state to another. E.g., kernel data structures and system calls of an operating system.

An **execution fragment** is a state-operation alternating sequence: $s_0 \alpha_1 s_1 \alpha_2 s_2 \ldots \alpha_n s_n$. We are interested in the **invariants** of a system, which are conditions (on states) satisfied throughout *any* execution fragment.

To show that a system actually has an invariant, we can show:

- The invariant holds for the initial state
- The invariant is *preserved* by every operation

That is, an invariant should be a precondition (*assumed before executing an operation*) but at the same time a postcondition (*established after executing an operation*). Although an invariant may be *violated during execution*.

Preservation of invariant

```
int8_t cnt;

void init(void)
{
    cnt = 0;
}
```

```
void add1(void)
{
    cnt = cnt + 1;
    if (cnt >= 100)
        cnt = cnt - 100;
}
```

```
void add2(void)
{
    cnt = cnt + 2;
    if (cnt >= 100)
        cnt = cnt - 100;
}
```

This simple example has one state variable *cnt* and two operations *add*1 and *add*2.

It has an invariant: $cnt < 100$, which rules out the possibility of signed integer overflow.

State machine refinement

In many cases, we would like to have some *abstraction*. We can define another abstract state machine Spec that has the same set of operations as our concrete state machine Prog, but the states and state transitions of Spec should be simple for people to understand what this system is trying to achieve.

The two state machines are connected by an **abstraction relation** *AR*, which, intuitively, say how to interpret a concrete state as an abstract state.

We say that Prog is a **refinement** of Spec if we can transform every execution fragment $s_0 \alpha_1 s_1 \alpha_2 s_2 \ldots \alpha_n s_n$ of Prog to an execution fragment $t_0 \alpha_1 t_1 \alpha_2 t_2 \ldots \alpha_n t_n$ of Spec such that (1) the corresponding actions are the same and (2) the corresponding states satisfy *AR*. This can be shown if:

- The abstraction relation holds for the initial state of Spec and Prog
- Assuming *AR*, executing the same operation for Spec and Prog establishes *AR*

With the refinement argument, we can reason about, e.g., the state after executing a certain sequence of operations, with the much simpler Spec.

State machine refinement

```
int8_t cnt;

void init(void)
{
    cnt = 0;
}
```

```
void add1(void)
{
    cnt = cnt + 1;
    if (cnt >= 100)
        cnt = cnt - 100;
}
```

```
void add2(void)
{
    cnt = cnt + 2;
    if (cnt >= 100)
        cnt = cnt - 100;
}
```

If we're only interested in whether *cnt* is an odd number, we can define Spec to be a state machine with only one boolean variable *isodd*. The transition relations can be defined as:

$$isodd \xrightarrow{+1} isodd' \triangleq isodd' = \neg isodd$$

$$isodd \xrightarrow{+2} isodd' \triangleq isodd' = isodd$$

The abstraction relation can be defined as:

$$AR \triangleq isodd \iff cnt \,\%\, 2 = 1$$

Note that since Prog is written in a low-level language with undefined behaviors, we still need to find a proper **representation invariant**, e.g., $cnt < 100$, to exclude undefined behaviors.

Expressing correctness definitions with Hoare triples

We can express all previous correctness definitions with *Hoare triples*.

Absence of undefined behavior

$$\{Pre\}C\{True\}$$

Algorithmic property

$$\{Pre\}C\{Post\}$$

Preservation of invariant

$$\{Inv\}C\{Inv\}$$

State machine refinement

$$\{RI \wedge AR\}C\{RI \wedge (\exists t'.t \rightarrow t' \wedge AR[t'/t])\}$$

Decorated program

To verify this program using Hoare logic, we have to:

```
uint8_t x, y, z;

if (z >= y)

    x = z;

else

    x = y;

x = x + 1;
```

Decorated program

To verify this program using Hoare logic, we have to:

1. Specify pre and post conditions

```
uint8_t x, y, z;
```

$\{True\}$

```
if (z >= y)
```

```
    x = z;
```

```
else
```

```
    x = y;
```

```
x = x + 1;
```

$\{x > y \wedge x > z\}$

Decorated program

To verify this program using Hoare logic, we have to:

1. Specify pre and post conditions
2. Generate intermediate assertions

```
uint8_t x, y, z;
```

$\{True\}$

```
if (z >= y)
```

```
    x = z;
```

```
else
```

```
    x = y;
```

```
x = x + 1;
```

$\{x > y \wedge x > z\}$

Decorated program

To verify this program using Hoare logic, we have to:

1. Specify pre and post conditions
2. Generate intermediate assertions

```
uint8_t x, y, z;
```

$\{True\}$

```
if (z >= y)
```

```
    x = z;
```

```
else
```

```
    x = y;
```

$\{x + 1 > y \wedge x + 1 \geq z\}$

```
x = x + 1;
```

$\{x > y \wedge x > z\}$

Decorated program

To verify this program using Hoare logic, we have to:

1. Specify pre and post conditions
2. Generate intermediate assertions

```
uint8_t x, y, z;
```

$\{True\}$

```
if (z >= y)
```

```
    x = z;
```

$\{x + 1 > y \wedge x + 1 \geq z\}$

```
else
```

```
    x = y;
```

$\{x + 1 > y \wedge x + 1 \geq z\}$
$\{x + 1 > y \wedge x + 1 \geq z\}$

```
x = x + 1;
```

$\{x > y \wedge x > z\}$

Decorated program

To verify this program using Hoare logic, we have to:

1. Specify pre and post conditions
2. Generate intermediate assertions

```
uint8_t x, y, z;
```

$\{True\}$

```
if (z >= y)
```

$\{z + 1 > y \land z + 1 \geq z\}$

```
    x = z;
```

$\{x + 1 > y \land x + 1 \geq z\}$

```
else
```

$\{y + 1 > y \land y + 1 \geq z\}$

```
    x = y;
```

$\{x + 1 > y \land x + 1 \geq z\}$
$\{x + 1 > y \land x + 1 \geq z\}$

```
x = x + 1;
```

$\{x > y \land x > z\}$

Decorated program

To verify this program using Hoare logic, we have to:

1. Specify pre and post conditions
2. Generate intermediate assertions

```
uint8_t x, y, z;
```

$\{True\}$

```
if (z >= y)
```

$\{True \wedge z \geq y\}$
$\{z + 1 > y \wedge z + 1 \geq z\}$

```
    x = z;
```

$\{x + 1 > y \wedge x + 1 \geq z\}$

```
else
```

$\{True \wedge z < y\}$
$\{y + 1 > y \wedge y + 1 \geq z\}$

```
    x = y;
```

$\{x + 1 > y \wedge x + 1 \geq z\}$
$\{x + 1 > y \wedge x + 1 \geq z\}$

```
x = x + 1;
```

$\{x > y \wedge x > z\}$

Decorated program

To verify this program using Hoare logic, we have to:

1. Specify pre and post conditions
2. Generate intermediate assertions
3. Reason about logical and arithmetic formulae

```
uint8_t x, y, z;
```

$$\{True\}$$

```
if (z >= y)
```

$$\{True \wedge z \geq y\} \implies \{z + 1 > y \wedge z + 1 \geq z\}$$

```
    x = z;
```

$$\{x + 1 > y \wedge x + 1 \geq z\}$$

```
else
```

$$\{True \wedge z < y\} \implies \{y + 1 > y \wedge y + 1 \geq z\}$$

```
    x = y;
```

$$\{x + 1 > y \wedge x + 1 \geq z\}$$
$$\{x + 1 > y \wedge x + 1 \geq z\}$$

```
x = x + 1;
```

$$\{x > y \wedge x > z\}$$

Decorated program

To verify this program using Hoare logic, we have to:

1. Specify pre and post conditions
2. Generate intermediate assertions
3. Reason about logical and arithmetic formulae

```
uint8_t x, y, z;
```

$\{y < 255 \land z < 255\}$

```
if (z >= y)
```

$\{ y < 255 \land z < 255 \land z \geq y\} \implies$
$\{z + 1 > y \land z + 1 \geq z\}$

```
    x = z;
```

$\{x + 1 > y \land x + 1 \geq z\}$

```
else
```

$\{ y < 255 \land z < 255 \land z < y\} \implies$
$\{y + 1 > y \land y + 1 \geq z\}$

```
    x = y;
```

$\{x + 1 > y \land x + 1 \geq z\}$
$\{x + 1 > y \land x + 1 \geq z\}$

```
x = x + 1;
```

$\{x > y \land x > z\}$

Decorated program

To verify this program using Hoare logic, we have to:

1. Specify pre and post conditions
2. Generate intermediate assertions
3. Reason about logical and arithmetic formulae

We'll see how to automate step 2 with **symbolic execution** and step 3 with **SMT solving**.

```
uint8_t x, y, z;
```

$$\{y < 255 \land z < 255\}$$

```
if (z >= y)
```

$$\{ y < 255 \land z < 255 \land z \geq y\} \implies \{z + 1 > y \land z + 1 \geq z\}$$

```
    x = z;
```

$$\{x + 1 > y \land x + 1 \geq z\}$$

```
else
```

$$\{ y < 255 \land z < 255 \land z < y\} \implies \{y + 1 > y \land y + 1 \geq z\}$$

```
    x = y;
```

$$\{x + 1 > y \land x + 1 \geq z\}$$
$$\{x + 1 > y \land x + 1 \geq z\}$$

```
x = x + 1;
```

$$\{x > y \land x > z\}$$

Symbolic execution

Let $[\![P]\!]$ be the set of states characterized by the **state predicate** P. For example:

$$[\![x > -2 \wedge x < 3]\!] = [\![x > -2]\!] \cap [\![x < 3]\!] = \{-1, 0, 1, 2\} \text{ , if } x \text{ is an integer}$$

$\{P\}C\{Q\}$ says: "C (if terminates) brings any state in $[\![P]\!]$ to some state in $[\![Q]\!]$."

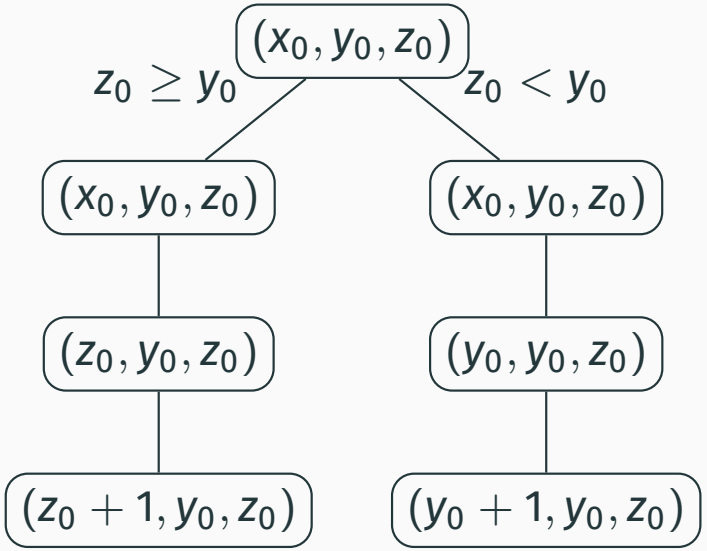
When mentally executing a program, we often consider a single state. But Hoare logic tells us that, in fact, we can consider a program as a *transformer* from a set of states to another set of states.

We use a **symbolic value** to denote "a set of values (of a certain type)".

We maintain two data structures, **symbolic state** and **path conditions**, during symbolic execution of a program. A symbolic state is like a normal state, except it can map variables to symbolic values (or symbolic expressions). A path condition limits the possible values that variables on a certain path can take.

Symbolic execution

```
uint8_t x, y, z;
if (z >= y)
    x = z;
else
    x = y;
x = x + 1;
```



A **path** is formed by conjoining all path conditions from the root node to a leaf node, and the relations between old and new values obtained via the symbolic state.

The **transition relation** of a program is the disjunction of all its paths.

$$(x_0, y_0, z_0) \xrightarrow{C} (x_1, y_1, z_1) \triangleq$$

$$(z_0 \geq y_0 \wedge x_1 = z_0 + 1 \wedge y_1 = y_0 \wedge z_1 = z_0)$$

$$\vee$$

$$(z_0 < y_0 \wedge x_1 = y_0 + 1 \wedge y_1 = y_0 \wedge z_1 = z_0)$$

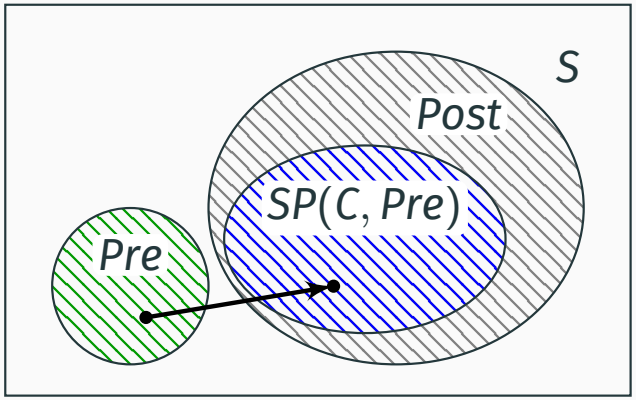
Strongest postcondition

Note that the transition relation generated by symbolic execution *precisely* captures the behavior of a program. (In contrast, $\{P\}C\{True\}$ is valid but *imprecise*.)

We can thus calculate the *minimal* set of the reachable states starting from a state in $[\![Pre]\!]$ by conjoining the precondition with the transition relation. We call this formula the **strongest postcondition**, written as $SP(C, Pre)$. For example:

$$SP(C, Pre) \triangleq (y_0 < 255 \wedge z_0 < 255) \wedge \left(\begin{array}{c} (z_0 \geq y_0 \wedge x_1 = z_0 + 1 \wedge y_1 = y_0 \wedge z_1 = z_0) \\ \vee \\ (z_0 < y_0 \wedge x_1 = y_0 + 1 \wedge y_1 = y_0 \wedge z_1 = z_0) \end{array} \right)$$

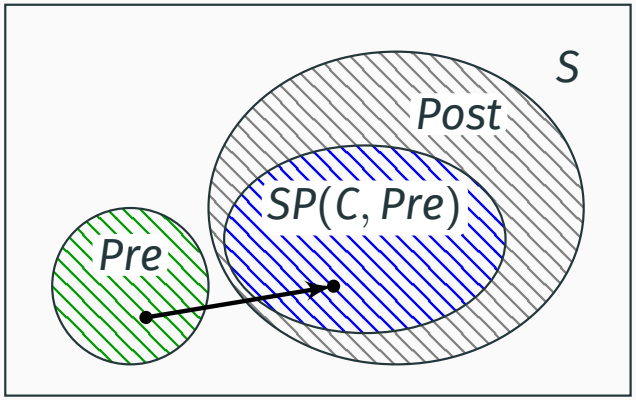
Proving Hoare triples with the strongest postcondition

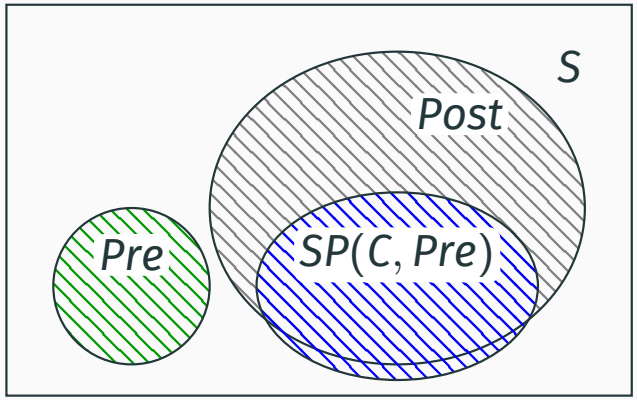


To prove $\{Pre\}C\{Post\}$, we can show that every state in $[\![SP(C, Pre)]\!]$ is also in $[\![Post]\!]$, namely

$$SP(C, Pre) \implies Post$$

Proving Hoare triples with the strongest postcondition



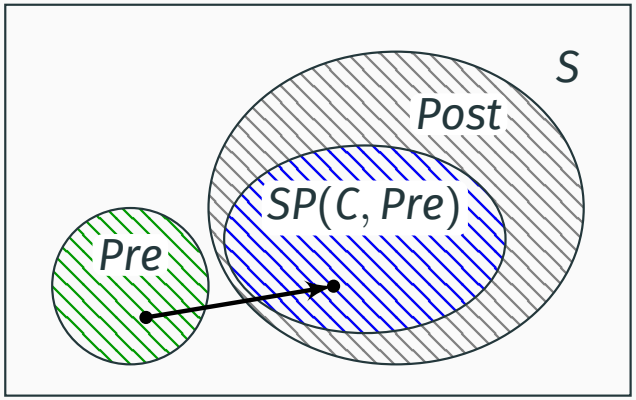


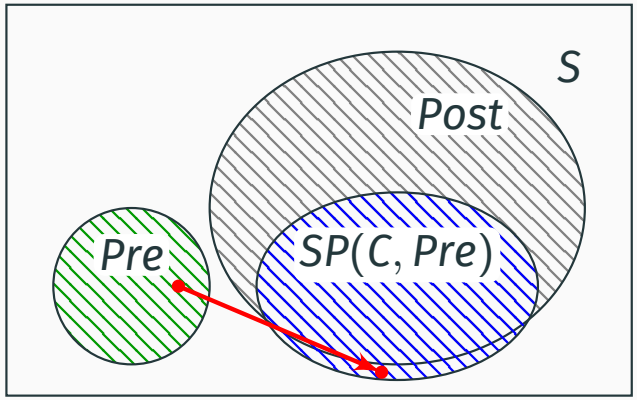
To prove $\{Pre\}C\{Post\}$, we can show that every state in $[\![SP(C, Pre)]\!]$ is also in $[\![Post]\!]$, namely

$$SP(C, Pre) \implies Post$$

One interesting fact about being the strongest is that if the above implication does not hold, then there must be a state in $[\![Pre]\!]$ such that C can bring the state to a state outside $[\![Post]\!]$. We then know *exactly what is wrong* with our program.

Proving Hoare triples with the strongest postcondition





To prove $\{Pre\}C\{Post\}$, we can show that every state in $[\![SP(C, Pre)]\!]$ is also in $[\![Post]\!]$, namely

$$SP(C, Pre) \implies Post$$

One interesting fact about being the strongest is that if the above implication does not hold, then there must be a state in $[\![Pre]\!]$ such that C can bring the state to a state outside $[\![Post]\!]$. We then know *exactly what is wrong* with our program.

Automatic reasoning: SMT solving

The satisfiability problem (SAT) asks whether a propositional formula F has a **model**, which is a valuation of variables satisfying F.

Examples.

- $(P \wedge \neg Q) \vee R$ is sat because it has a model $\{P = \mathit{true}, Q = \mathit{false}, R = \mathit{false}\}$
- $P \wedge \neg P$ is unsat because it does not have a model

The satisfiability modulo theories (SMT) extends SAT with quantifiers (i.e., $\forall$ and $\exists$), and more useful data types such as integers, bit vectors and uninterpreted functions.

Examples.

- $\forall z.f(z) = y$ is sat because it has a model $\{y = 4, f = \lambda x.4\}$
- $x > 7 \wedge x < 3$ is unsat because it does not have a model

There are efficient algorithms to decide satisfiability. SAT/SMT itself are also actively (and independently) developed by their research communities.

Validity vs. satisfiability

Validity. A logical formula F is valid if *all* valuations of variables satisfy F.

Satisfiability. A logical formula F is satisfiable if *some* valuation of variables satisfy F.

Theorem. A logical formula F is valid *iff* its negation $\neg F$ is unsatisfiable.

When the SMT solver says $\neg F$ is satisfiable, it also returns a **counterexample**. A counterexample is a valuation of variables satisfying $\neg F$ (and thus violates F).

Summary

We have introduced several definitions of program correctness, from as simple as saying my program does not have undefined behaviors, to finding an abstract specification to replace our concrete representation.

Even if we're not doing verification, knowing what our programs/systems are supposed to do is still good.

Symbolic execution can be used to generate the strongest postcondition, and SMT solving can do the automatic reasoning.

Formal verification of a snapshot-consistent FTL

Crash and file system inconsistency

Crashes have two sources of non-determinism:

1. A crash can occur at any point
2. Reordering due to scheduling and concurrent operations

A write operation would:

1. Allocate a new block
2. Append the newly allocated block to the written file
3. Write the contents to the block

What would happen if step 2 succeeds but step 1 fails? The same block can be allocated to two distinct files.

Or if step 1 succeeds but step 2 fails? Memory leak.

Such file system inconsistency can be avoided by ensuring **the atomicity of multiple disk writes**.

Snapshot-consistent flash translation layer

Conventional designs often use a **write-ahead log** to provide the atomicity of multiple disk writes.

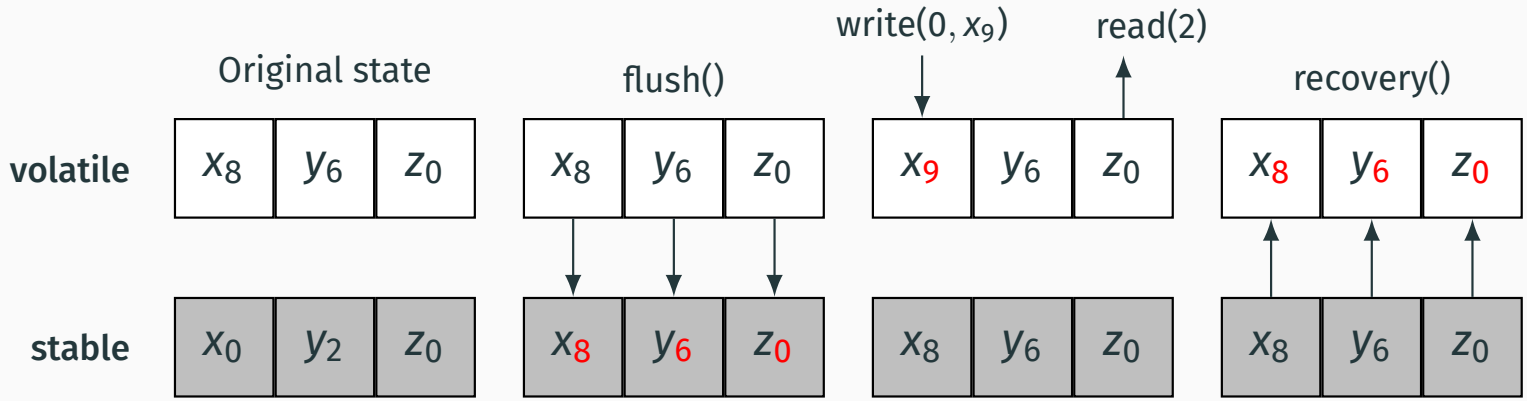
Problem: Data are written twice, resulting in poor performance

We want to solve this issue at the flash translation layer (FTL) because flash disks have some features we can exploit.

Our design goal:

- Guarantee the atomicity of multiple disk writes
- Don’t sacrifice performance

Specification



This specification has the following features:

- A flush copies the volatile array to the stable array
- A write and a read only access the volatile array
- A recovery copies the stable array to the volatile array (opposite of flush)
- The volatile array is allowed to have any value upon a crash, but the stable one must remain unchanged

This specification is **snapshot-consistent** because invoking a flush is like creating a snapshot. This naturally ensures the atomicity of multiple disk writes between two consecutive flushes. Next, we give a formal version of this specification.

Operational specification for successful operations

Write (successful)

$$t \xrightarrow{w_{a,d}} t' \triangleq t'.volatile = t.volatile[a \mapsto d] \wedge t'.stable = t.stable$$

Flush (successful)

$$t \xrightarrow{f} t' \triangleq t'.volatile = t.volatile \wedge t'.stable = t.volatile$$

GC (successful)

$$t \xrightarrow{g} t' \triangleq t'.volatile = t.volatile \wedge t'.stable = t.stable$$

Recovery (successful)

$$t \xrightarrow{r} t' \triangleq t'.volatile = t.stable \wedge t'.stable = t.stable$$

Read (successful)

$$read(a) \triangleq volatile[a]$$

Operational specification for crashed operations

Write (crashed)

$$t \xrightarrow{w^c_{a,d}} t' \triangleq t'.stable = t.stable$$

Flush (crashed)

$$t \xrightarrow{f^c} t' \triangleq t'.stable = t.stable \lor t'.stable = t.volatile$$

GC (crashed)

$$t \xrightarrow{g^c} t' \triangleq t'.stable = t.stable$$

Recovery (crashed)

$$t \xrightarrow{r^c} t' \triangleq t'.stable = t.stable$$

Crashed flush becomes **non-deterministic** as we cannot know whether a crash occurs *before* or *after stable* is overwritten with *volatile*.

In fact, these crashed operations are all non-deterministic as we allow *volatile* to be assigned *any* value.

Formalizing correctness

We use the *state machine refinement argument* as our correctness definition. One condition we have to show is **per-operation correctness**:

$$\forall s, s'.\ s \xrightarrow{op} s' \land RI(s) \implies RI(s')$$

$$\forall s, s', t.\ s \xrightarrow{op} s' \land RI(s) \land AR(s, t) \implies \exists t'.t \xrightarrow{op} t' \land AR(s', t')$$

However, *RI* and *AR* would be too strong for us to prove when a crash occurs.

Reason: A crash tears down all in-memory contents.

Solution: Defining a weaker representation invariant *CI* and a weaker abstraction relation *CR*. *CI* and *CR* should be *weak enough to be established by crashed operations*, but *strong enough to allow successful recovery*.

We reformulate our per-operation correctness to incorporate *CI* and *CR*.

Per-operation correctness

Functional correctness (for successful write, flush and GC)

$$\forall s, s'.\ s \xrightarrow{op} s' \wedge RI(s) \implies RI(s')$$

$$\forall s, s', t.\ s \xrightarrow{op} s' \wedge RI(s) \wedge AR(s,t) \implies \exists t'.t \xrightarrow{op} t' \wedge AR(s', t')$$

Crash invariance (for crashed write, flush and GC)

$$\forall s, s'.\ s \xrightarrow{op^c} s' \wedge RI(s) \implies CI(s')$$

$$\forall s, s', t.\ s \xrightarrow{op^c} s' \wedge RI(s) \wedge AR(s,t) \implies \exists t'.t \xrightarrow{op^c} t' \wedge CR(s', t')$$

Recovery correctness (for successful recovery)

$$\forall s, s'.\ s \xrightarrow{r} s' \wedge CI(s) \implies RI(s')$$

$$\forall s, s', t.\ s \xrightarrow{r} s' \wedge CI(s) \wedge CR(s,t) \implies \exists t'.t \xrightarrow{r} t' \wedge AR(s', t')$$

Recovery idempotence (for crashed recovery)

$$\forall s, s'.\ s \xrightarrow{r^c} s' \wedge CI(s) \implies CI(s')$$

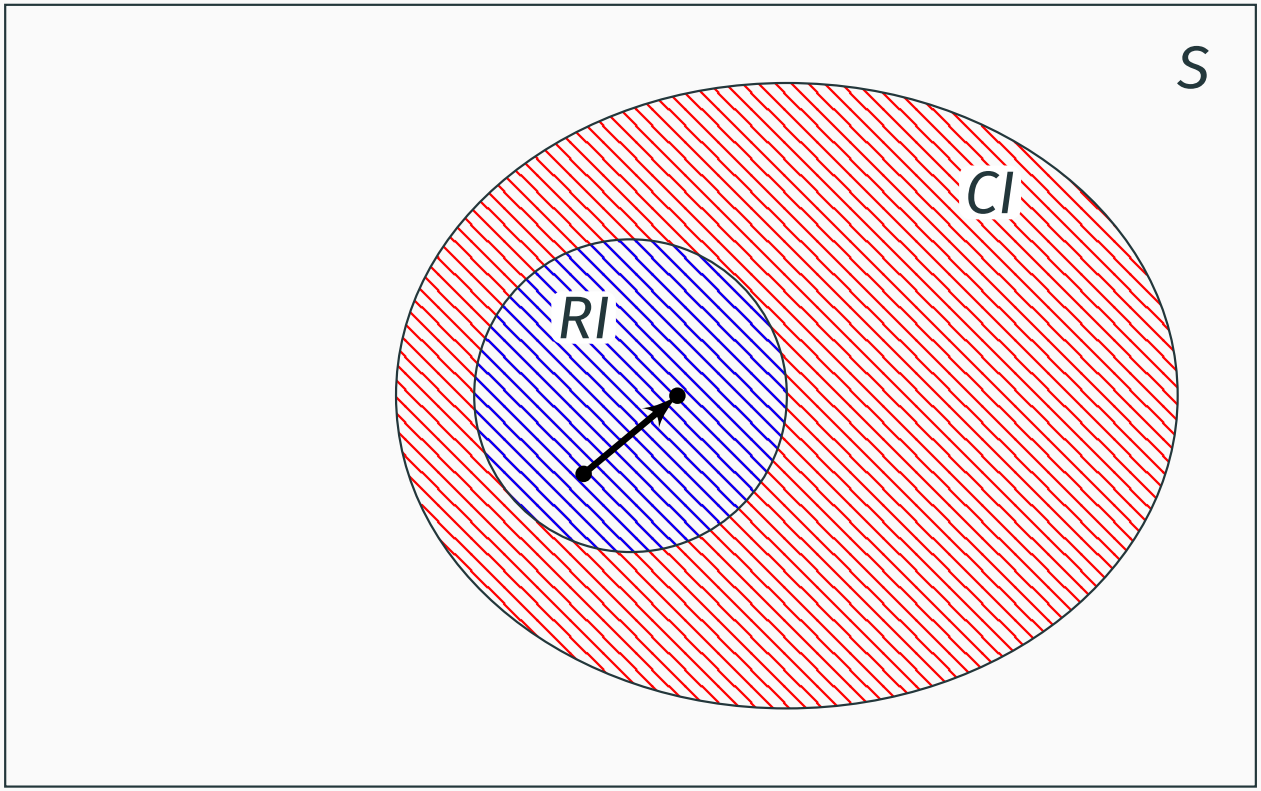
$$\forall s, s', t.\ s \xrightarrow{r^c} s' \wedge CI(s) \wedge CR(s,t) \implies \exists t'.t \xrightarrow{r^c} t' \wedge CR(s', t')$$

Observational equivalence (for read)

$$\forall s, t.\ RI(s) \wedge AR(s,t) \implies \forall a \in Addr.\ read(s) = read(t)$$

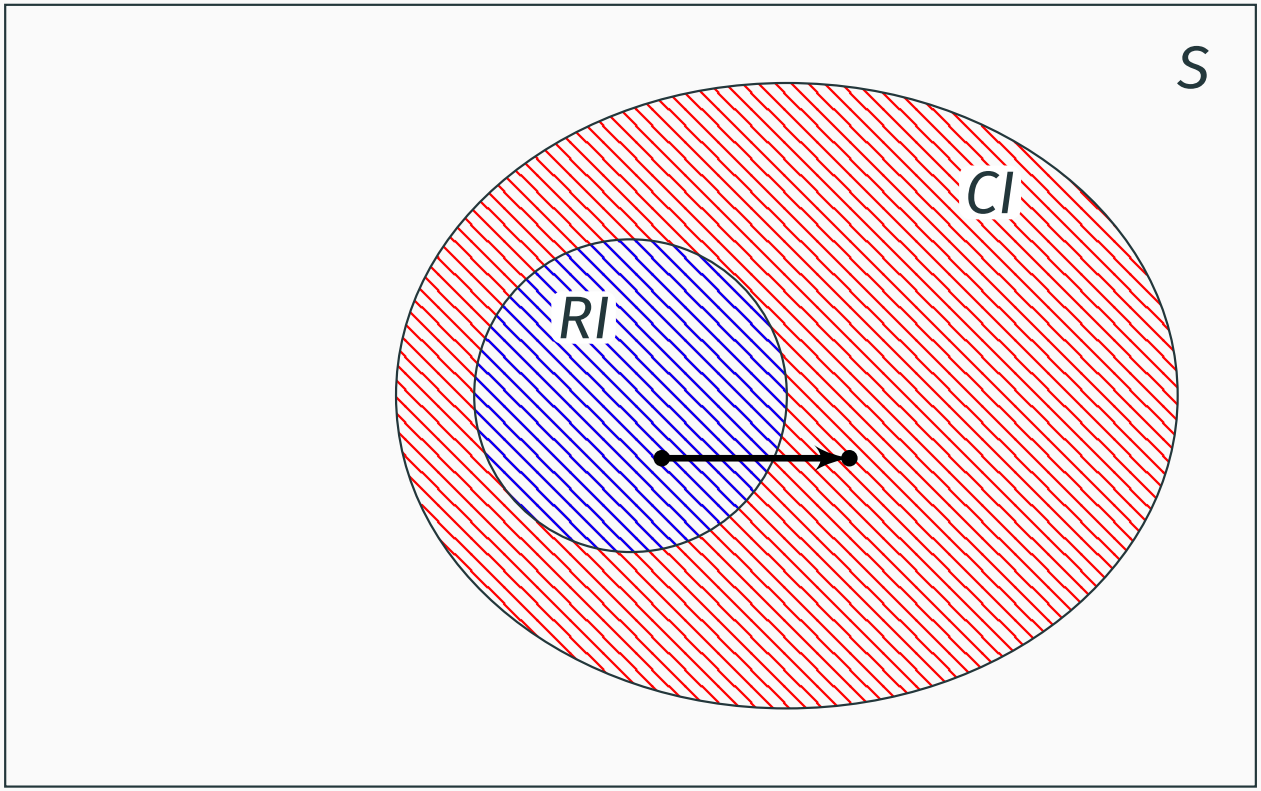
Per-operation correctness

Functional correctness



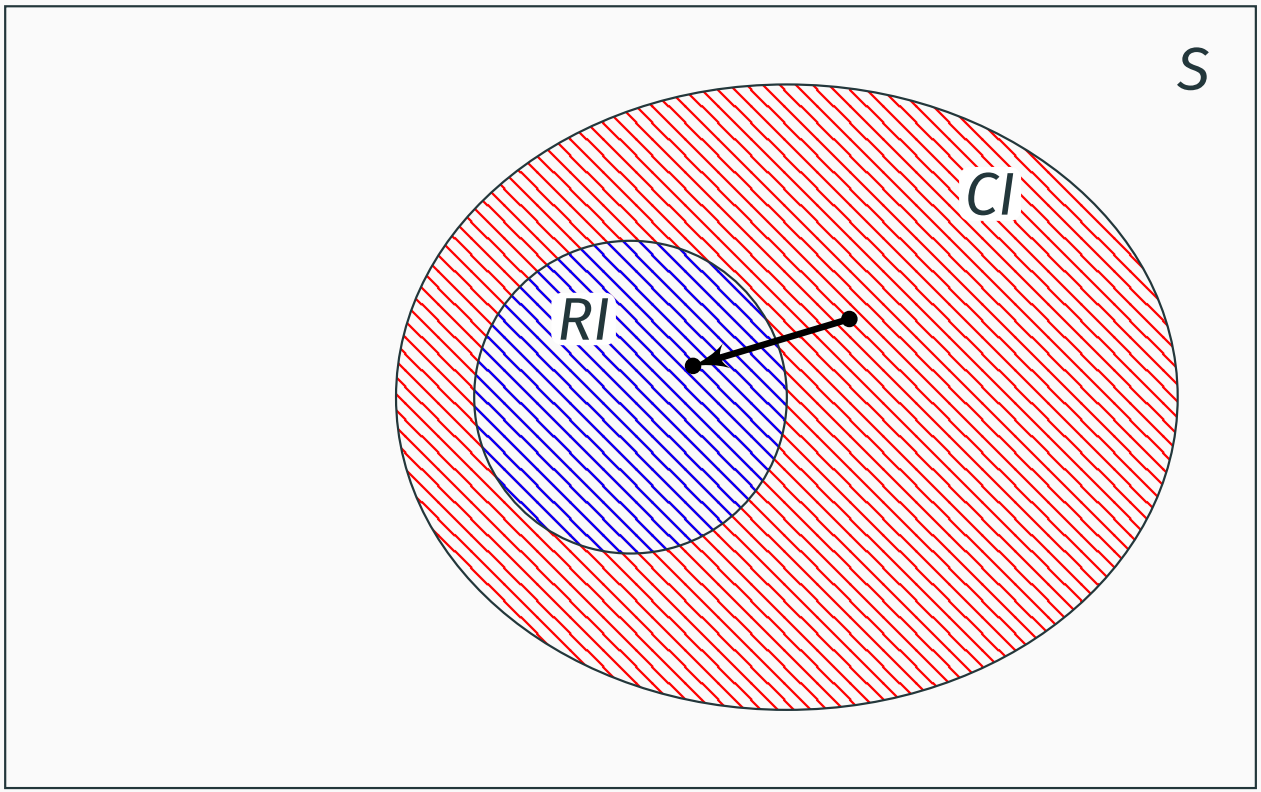
Per-operation correctness

Crash invariance



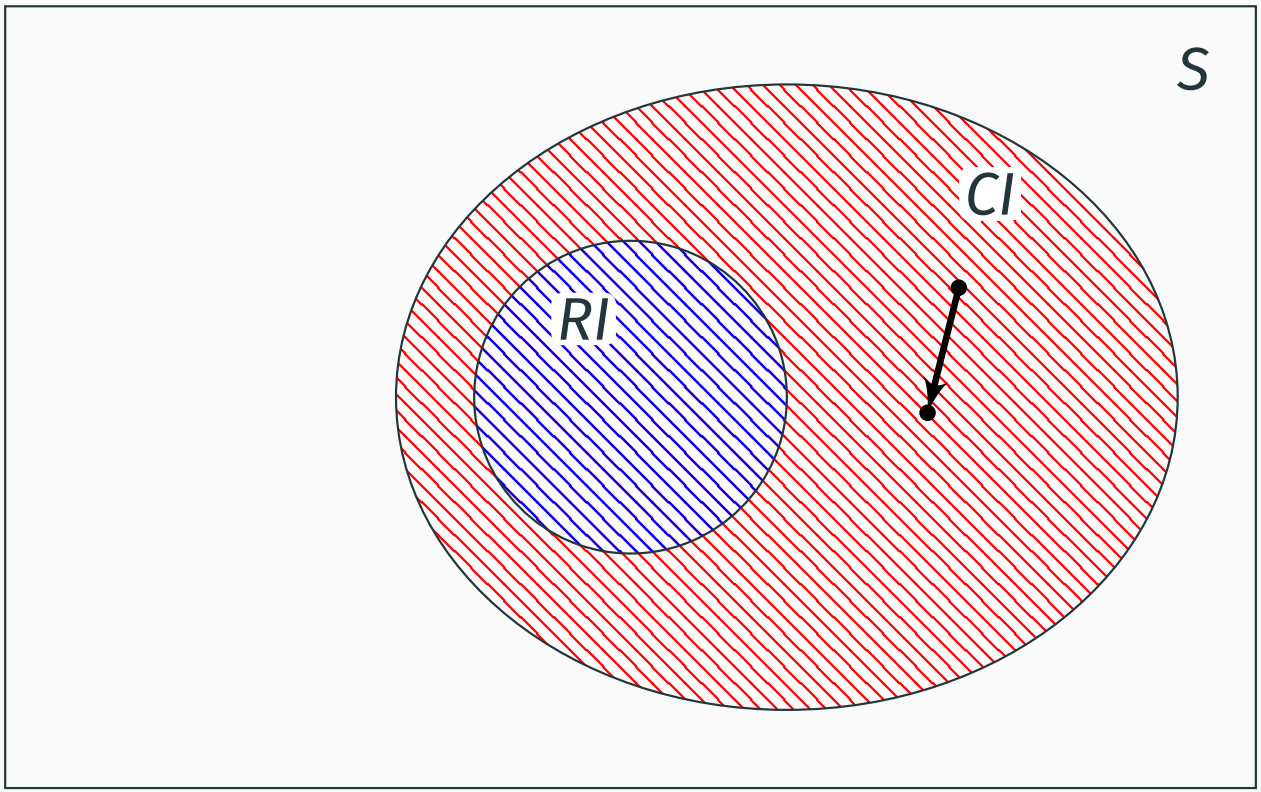
Per-operation correctness

Recovery correctness



Per-operation correctness

Recovery idempotence



Are we all prepared to verify our program?

Functional correctness (for successful write, flush and GC)

$$\forall s, s'.\ s \xrightarrow{op} s' \wedge RI(s) \implies RI(s')$$

$$\forall s, s', t.\ s \xrightarrow{op} s' \wedge RI(s) \wedge AR(s,t) \implies \exists t'.t \xrightarrow{op} t' \wedge AR(s', t')$$

Crash invariance (for crashed write, flush and GC)

$$\forall s, s'.\ s \xrightarrow{op^c} s' \wedge RI(s) \implies CI(s')$$

$$\forall s, s', t.\ s \xrightarrow{op^c} s' \wedge RI(s) \wedge AR(s,t) \implies \exists t'.t \xrightarrow{op^c} t' \wedge CR(s', t')$$

Transition relations of successful operations and specification operations are automatically generated by standard symbolic execution.

Representation invariants and abstraction relations are manually found by us.

How about transition relations of crashed operations? We need to formalize **crash behaviors**.

Modeling crash behaviors

Crash behavioral models should capture all states that can be observed after a crash.

First attempt (Yxv6's approach)

- Using *ite*s and fresh boolean variables: Can be efficiently solved by SMT solvers
- Only allow single old value: Concurrency is limited in one single operation

Second attempt (FSCQ's approach)

- Using lists: Lists can't be efficiently solved by SMT solvers
- Allow multiple old values: Concurrency is allowed across operations

Final attempt

- Using a *synced* function to record which parts of the disk a contain single value, and which parts contain multiple values (although we have no idea what the values actually are)
- On crash, one value remains unchanged; multiple values can become *any* value
- Functions can be efficiently solved by SMT solvers and concurrency is allowed across operations

Over-approximation: If we can prove our program correct for any value; then we have proved our program correct for multiple values.

Finding invariant

We have defined our formal specification, formalized our correctness definition, and modeled crash behaviors. Finally, we need to find the proper representation invariant and abstraction relation. This is often the most time-consuming part.

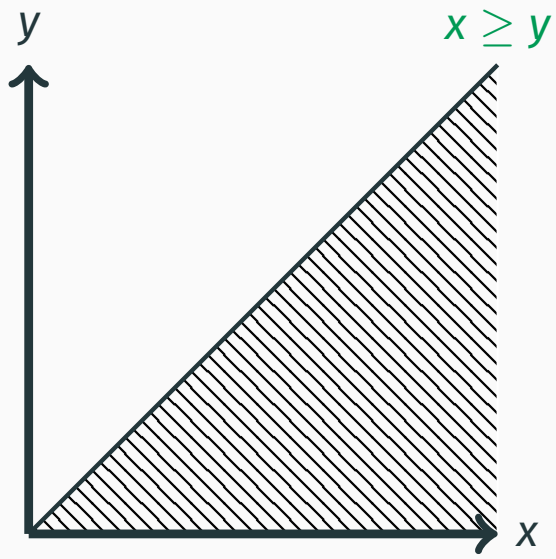
$$\forall s, s'.\ s \xrightarrow{op} s' \wedge C_0(s) \implies C_0(s')$$

$$\forall s, s'.\ s \xrightarrow{op} s' \wedge C_0(s) \wedge C_1(s) \implies C_0(s') \wedge C_1(s')$$

$$\ldots$$

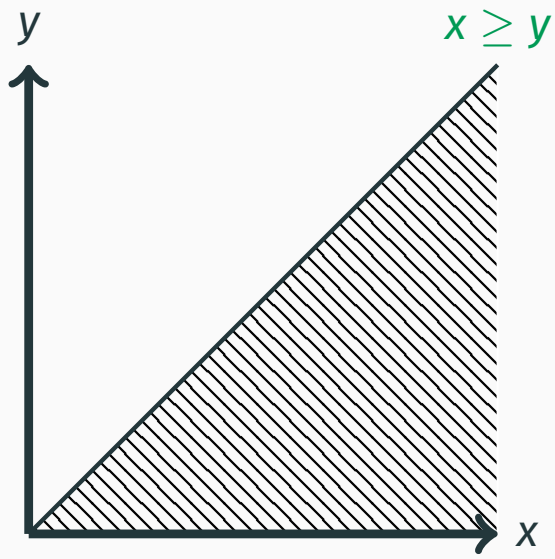
$$\forall s, s'.\ s \xrightarrow{op} s' \wedge C_0(s) \wedge C_1(s) \wedge \ldots \wedge C_n(s) \implies C_0(s') \wedge C_1(s') \wedge \ldots \wedge C_n(s')$$

Finding invariant



Start from the property of interest

Finding invariant

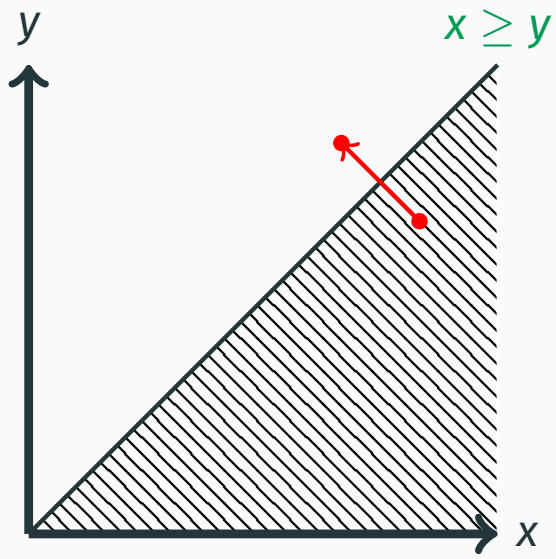


Start from the property of interest

↓

Ask SMT solver to verify the formula

Finding invariant



Start from the property of interest

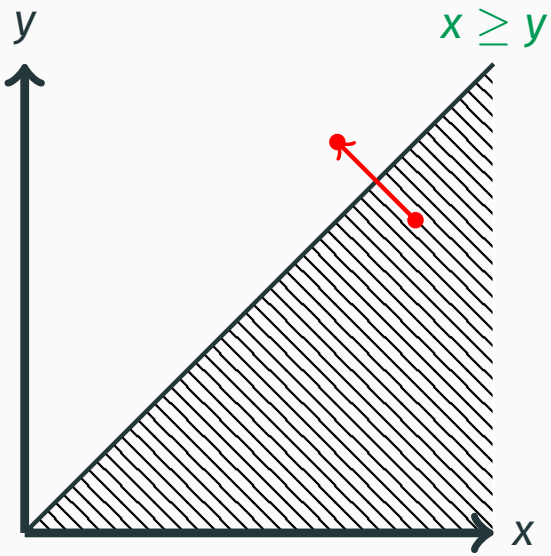
↓

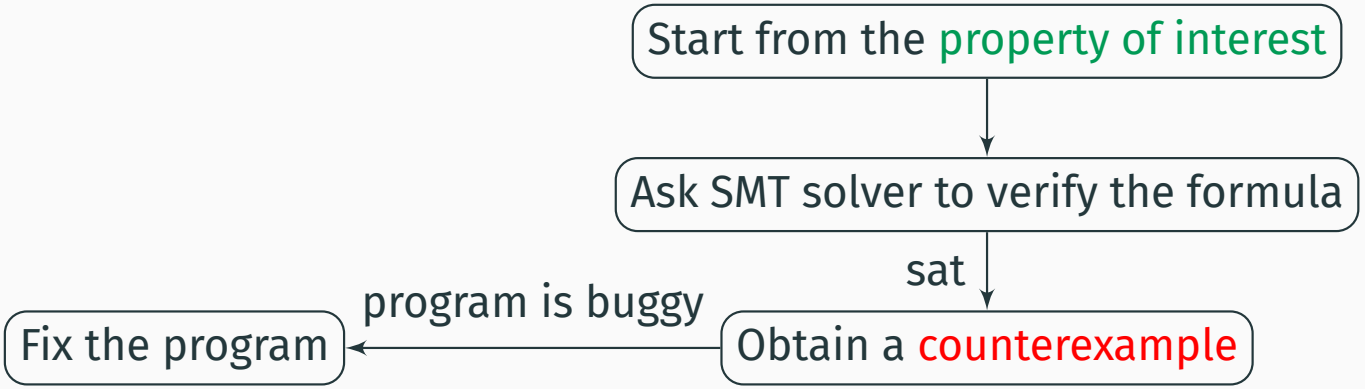
Ask SMT solver to verify the formula

↓ sat

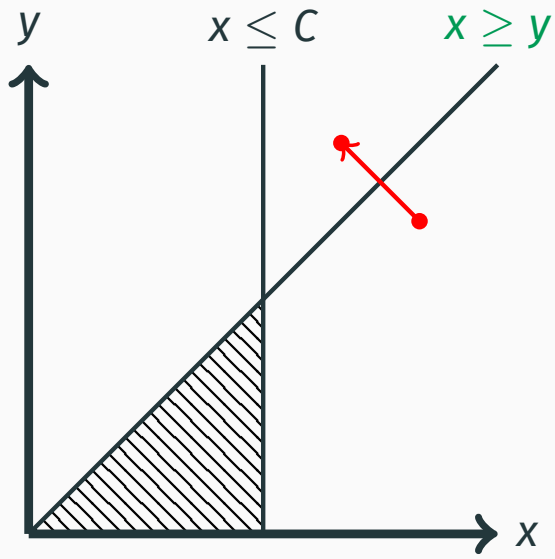
Obtain a counterexample

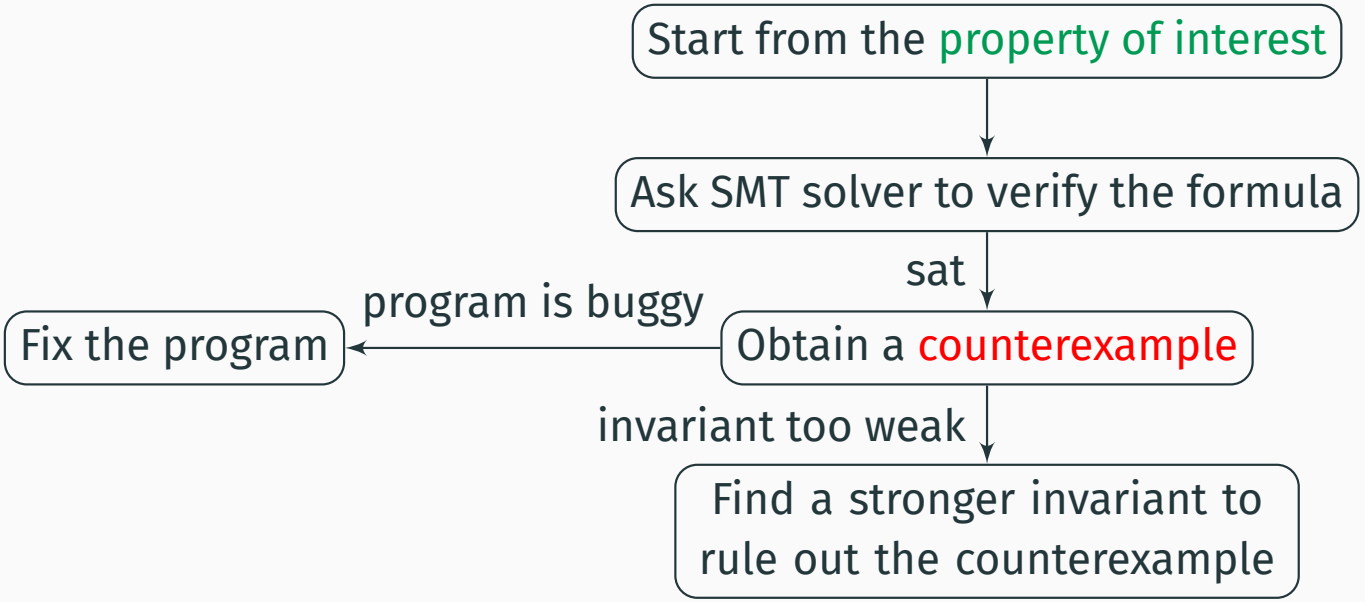
Finding invariant



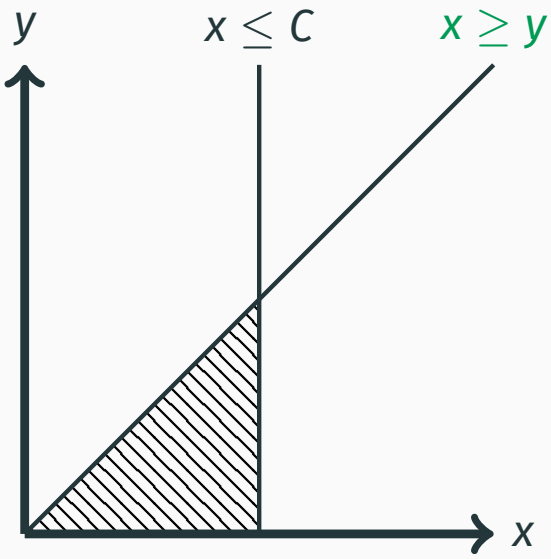


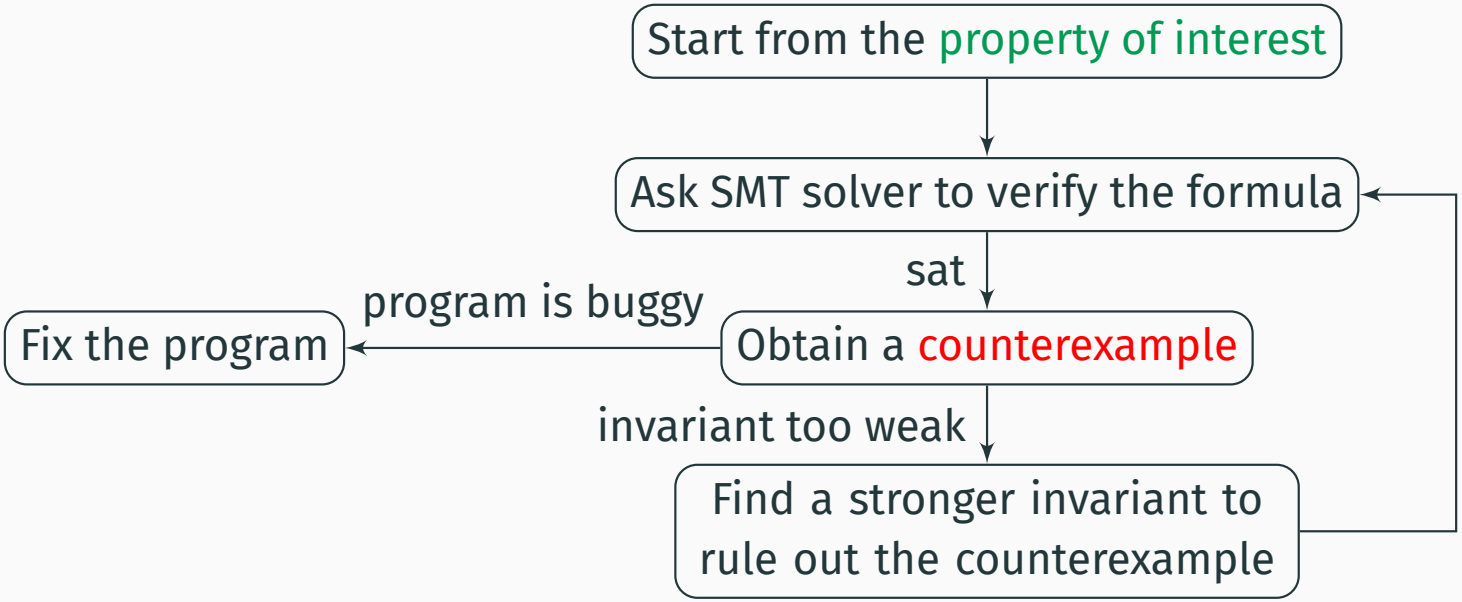
Finding invariant



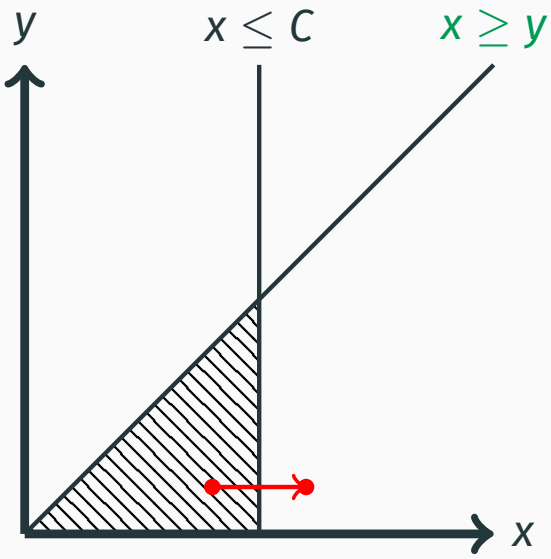


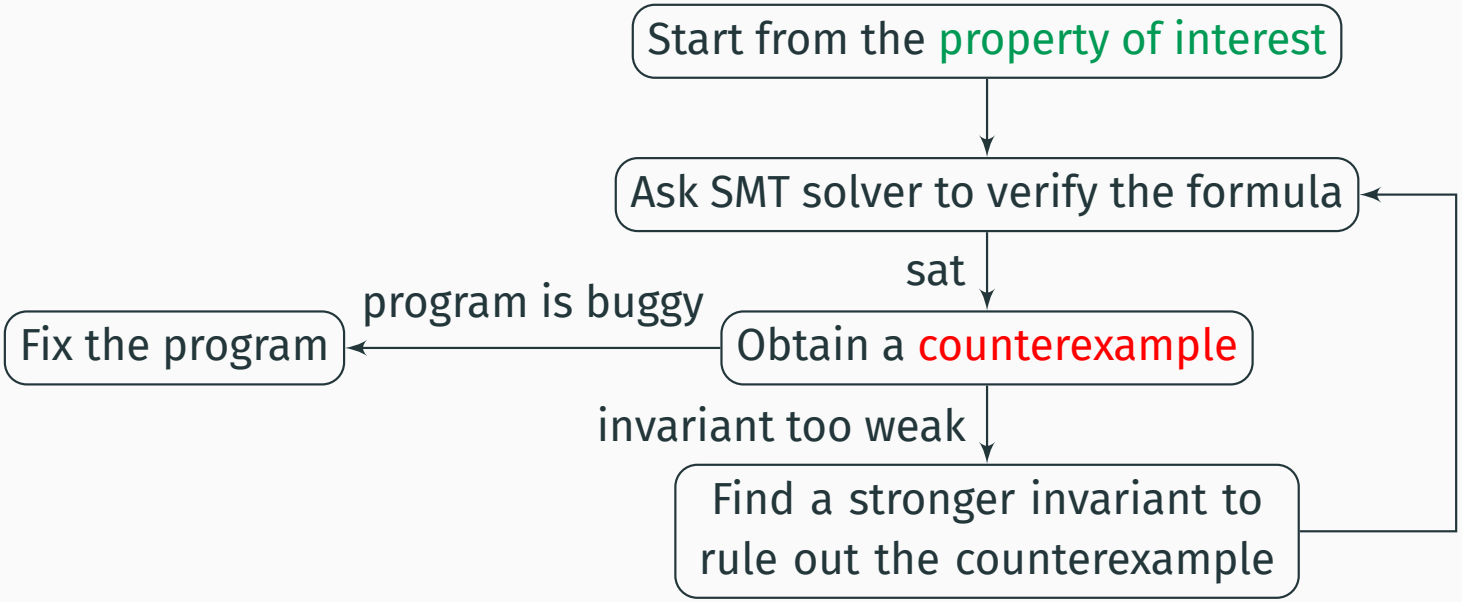
Finding invariant



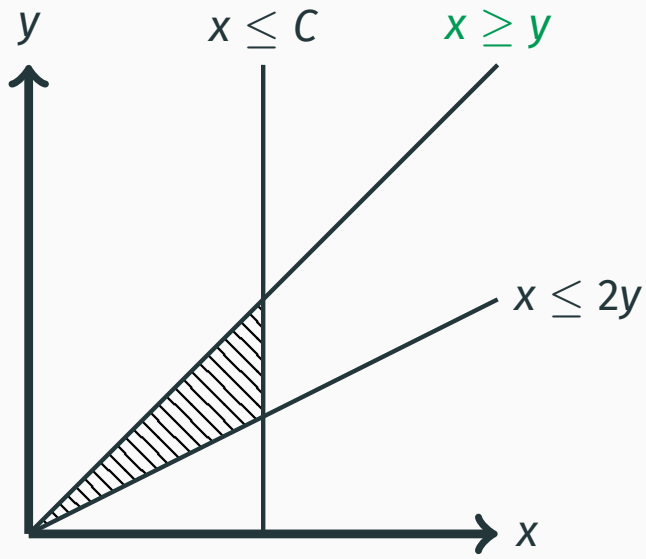


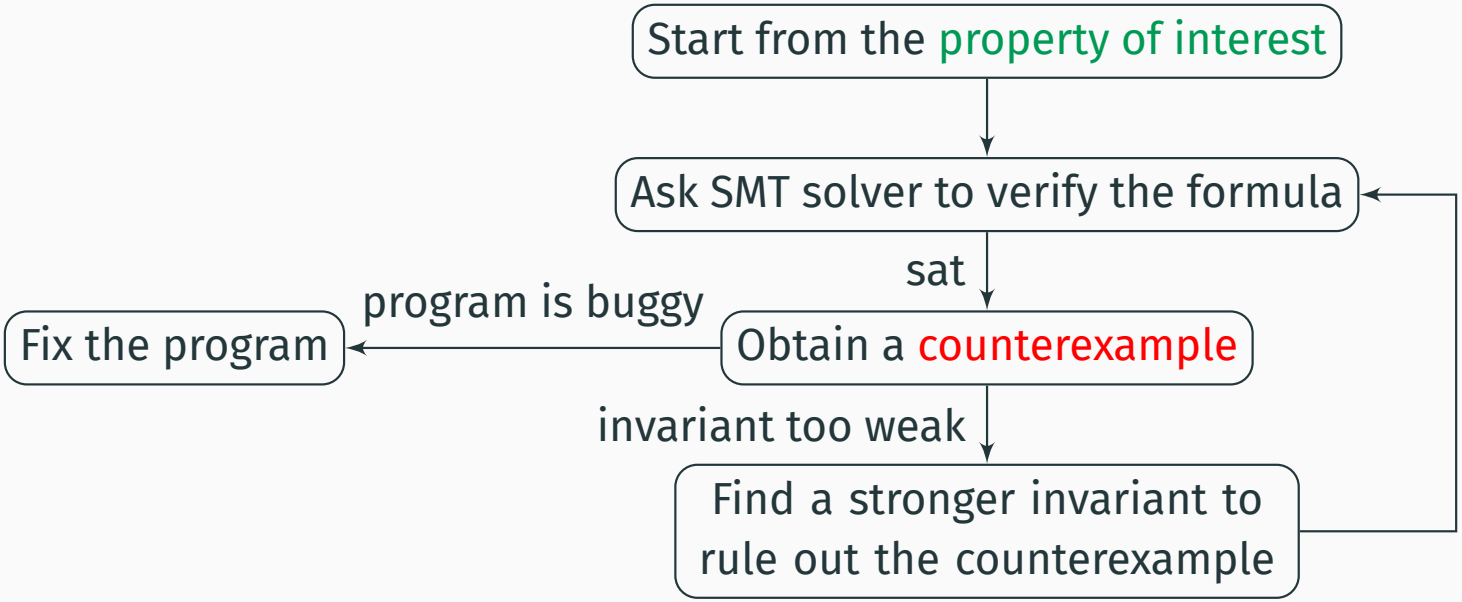
Finding invariant



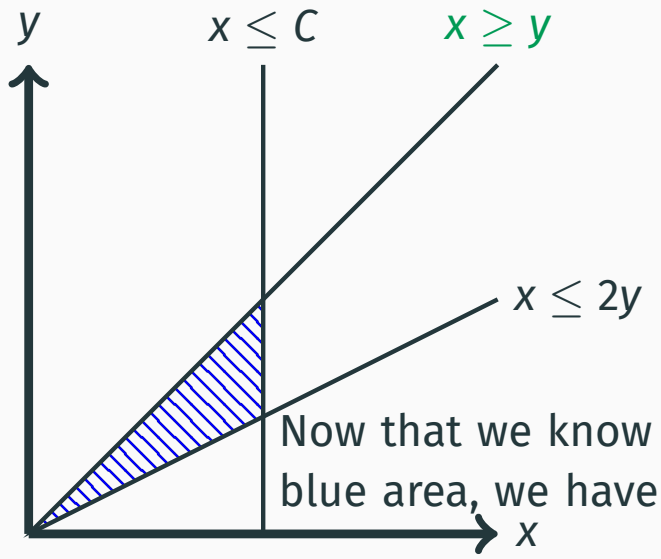


Finding invariant

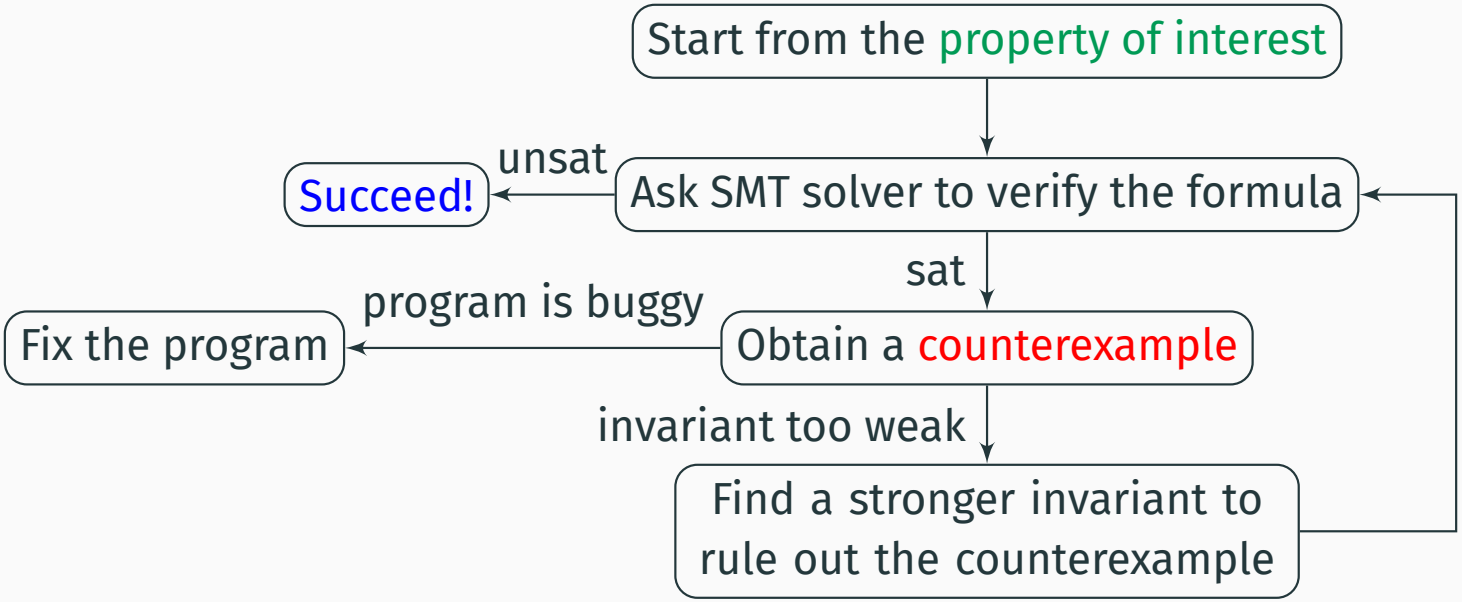




Finding invariant



Now that we know the states of our system will never leave this blue area, we have verified that our system indeed satisfies $x \geq y$.



Scaling automatic reasoning

There are certain forms of formulae that SMT solvers are particularly bad at, e.g.,

- quantifier alternation ($\forall x.\exists y.\forall z....$)
- deeply nested function application ($f_1(f_2(f_3(...(f_n(x))))))$
- non-deterministic transition

We can ...

- find a more efficient encoding of invariants and relations
- rewrite formulae
- modify specification
- tune the parameters of the SMT solver
- do anything to make the SMT happy (as long as it's correct!)

Conclusion

I have covered our experience on verifying a flash translation layer:

- Design the specification based on what your system is trying to achieve
- Choose a proper correctness definition
- Several aspects should be considered when modeling behaviors
- Automatic verification is an iterated process of finding invariants, fixing the program, and solving automatic reasoning bottleneck