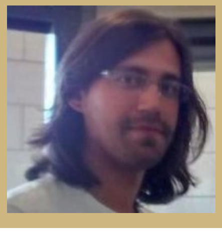
SMT-based Function Summarization for Software Verification



Leonardo Alt
Ethereum

Sepideh Asadi
USI

Martin Blicha
USI



Grigory Fedyukovich
Princeton

Antti Hyvärinen
USI

Natasha Sharygina
USI

University of Lugano(USI), Switzerland

Formal Verification in Lugano, Switzerland

Model checking software (HiFrog, FunFrog, eVolCheck, LoopFrog), ANSI-C programs

Interpolation-based Bounded Model Checking:

- Propositional and First-order Interpolation [TACAS'19],[LPAR'13],[FMCAD'17], [CAV'15]
- Function summarization [TACAS'17],[ATVA'12]
- Theory and Summary Refinement [SAT'17], [LPAR'18]

Formal Verification in Lugano, Switzerland

Boolean and Theory Reasoning (SAT/SMT):

Solver, ***OpenSMT***, combines MiniSAT2 SAT-Solver with state-of-the-art decision procedures for QF EUF, LRA, LIA, BV, RDL, IDL

Extensible: the SAT-to-theory interface facilitates design and plug-in of new decision procedures

Incremental*:* suitable for incremental verification

Open-source*:* available under MIT license

Parallelized*:* efficient search space partitioning

Efficient*:* competitive open-source SMT Solver according to SMT-Comp.

Formal Verification in Lugano, Switzerland

Efficient and adoptable-to-the-task decision procedures as computational engines of verification

SMT-based *Gas consumption* estimation for smart contracts [LPAR'18]

Incremental verification, Upgrade checking [STTT'17],[FMCAD'14],[TACAS'13]

Integrated dynamic and static analysis [ISSTA'14]

Model checking *Ethereum smart contracts* and mobile programs [ongoing]

More info at:

www.verify.inf.usi.ch

USI

The cost of poor software

From https://raygun.com/
"**11 of the most costly software errors in history**":

Bitcoin Mt. Gox Hack:
In 2011, the world's largest bitcoin exchange, after being hacked, lost over 800,000 bitcoins – worth around half a billion dollars!

- **Testing** is not sufficient to find the bug (not exhaustive!)
- The strongest tool to defend against hacks is formal verification. [Makerdao white paper]

Program correctness

Can we prove some properties **ALWAYS** hold in the program?

Program correctness

Can we prove some properties **ALWAYS** hold in the program?

In general, program verification is
undecidable,
but …
under some conditions/restrictions, it can be turned into a decidable problem!

Automated formal verification: Model Checking [Clarke & Emerson 1981, Queille & Sifakis 1982]

Pros

+ Mathematical and algorithmic way to verify the program
+ Exhaustive search on the state space
+ Fully automatic
+ Can guarantee the absence of bugs

Cons

- Computationally expensive
- State space explosion problem

Advances in model checking

Hardware

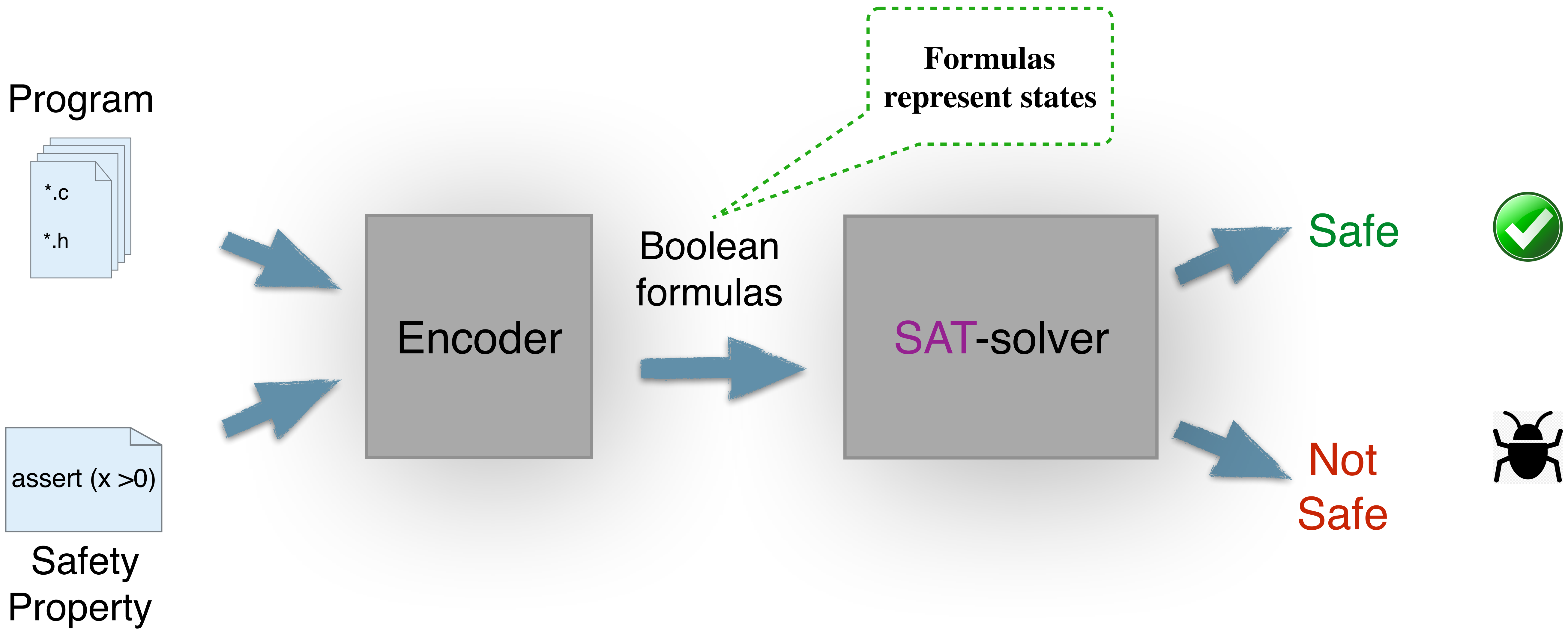
- Well-established techniques
  - Finite size model
  - Based on bit-precise encoding

Software

- Open Challenges!
  - Large bit-widths
  - Dynamic memory management
  - Unbounded recursion
  - Domain-specific languages
  - Long development history
  - …

Symbolic model checking [McMillan 1993]

SAT-based Model Checking [Biere et al. 1999]



Symbolic model checking [McMillan 1993]

SAT-based Model Checking [Biere et al. 1999]

SAT-based Model Checking

- ✓ An excellent tool for many problem domains
- ✓ Very efficient SAT-solvers exist
- ✗ Very low-level language —> large formulations
- ✗ Makes search space larger
- ✗ Sometimes even prevent from termination

Abstraction-based model checking [kurshan1994, Clarke et al. 2000]

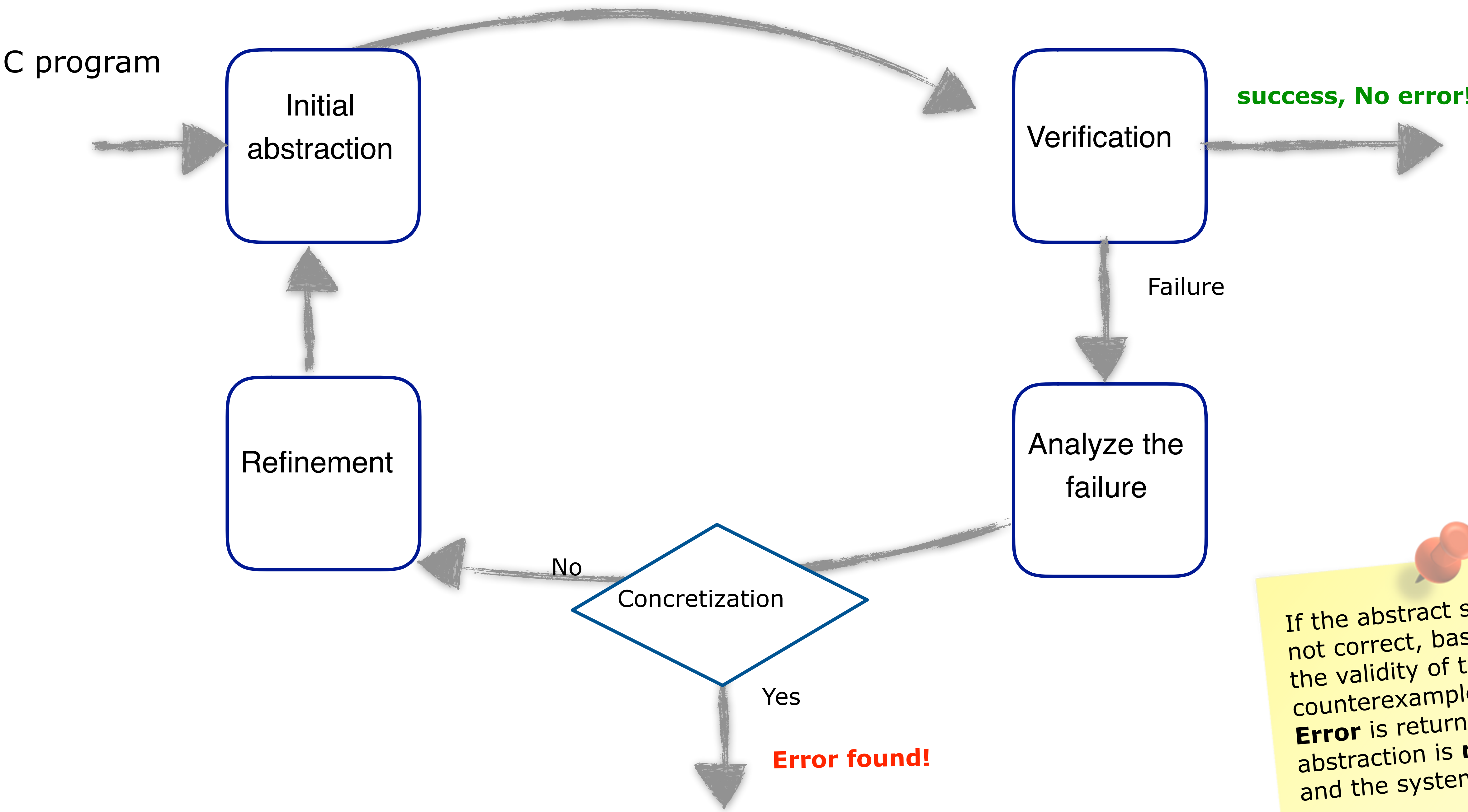
- **Problem**: High complexity of software model checking

- **Solution:**

  ➡ **Abstraction** : Removes or **simplifies** details of the system that are irrelevant to the property under consideration

The paradigm of abstract-check-refine (CEGAR) [Clarke et al. 2000]

The paradigm of abstract-check-refine (CEGAR) [Clarke et al. 2000]

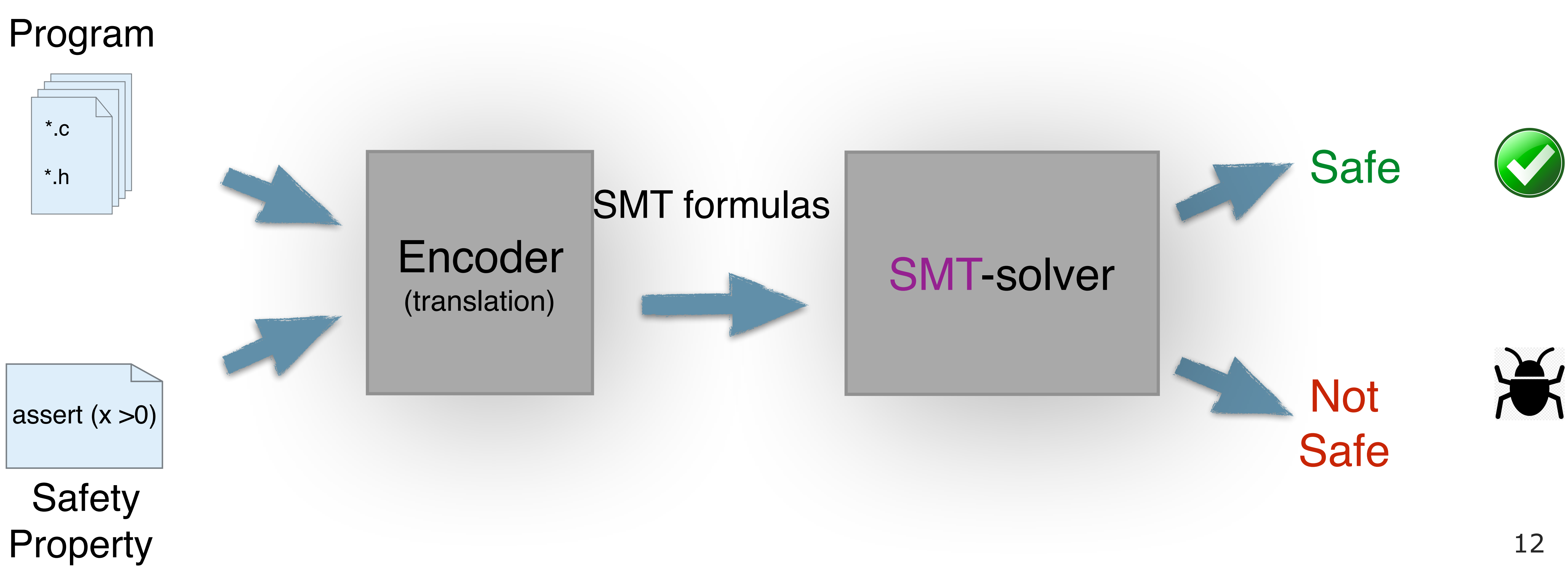


If the abstract system is not correct, based on the validity of the counterexample, either **Error** is returned or the abstraction is **refined** and the system iterates.

SMT

- Satisfiability Modulo Theory (SMT)
- Deciding the satisfiability of a **first-order logic over different theories**
- SMT can create verification engines that can reason natively at a higher level of abstraction

The focus of this talk



SMT vs. SAT encoding

```c
int
inc(int n)
{
    return n + 1;
}

int nondet();

int
main()
{
    int n = nondet();
    if(n >= 0 && n < 1000)
    {
        n = inc(n);
        assert(n > 0);
    }
    return 0;
}
```

SAT encoding: 4212 lines

SMT encoding: $(inc(n_0) = n_0 + 1) \wedge$
$(n_0 >= 0 \wedge n_0 < 1000) \rightarrow$
$(n_1 = inc(n_0) \wedge \neg(n_1 > 0)) \wedge$
$\neg(n_0 >= 0 \wedge n_0 < 1000) \rightarrow \top$

More expressive

More compact

More light-weight

Efficient solving procedure

Hierarchy of different theories

- Equality Logic & Uninterpreted Functions (EUF)
  - Example: $(f(x, y) \neq f(u, v)) \wedge (x = u) \wedge (y = v)$

Hierarchy of different theories

- Equality Logic & Uninterpreted Functions (EUF)

  - Example: $(f(x,y) \neq f(u,v)) \wedge (x = u) \wedge (y = v)$

- Linear Real Arithmetic (LRA)
  - Example: $(x + y \leq 0) \wedge (x = 0) \wedge (\neg a \vee (x = 1) \vee (y \geq 0))$

Hierarchy of different theories

- Equality Logic & Uninterpreted Functions (EUF)

  - Example: $(f(x,y) \neq f(u,v)) \wedge (x = u) \wedge (y = v)$

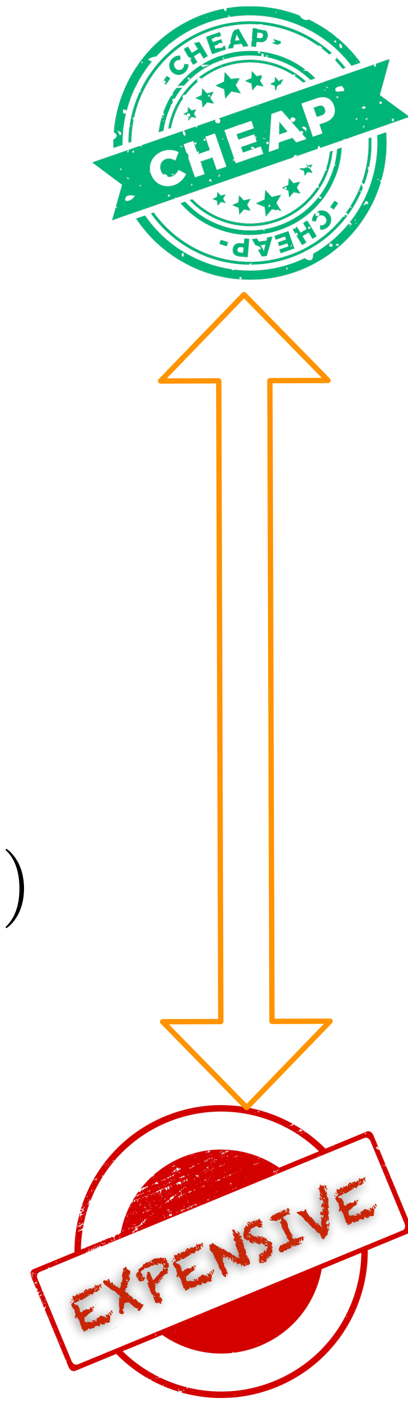
- Linear Real Arithmetic (LRA)
  - Example: $(x + y \leq 0) \wedge (x = 0) \wedge (\neg a \vee (x = 1) \vee (y \geq 0))$

- Theory of Bit-Vectors (BV)

  - Example: $\left[(a + b) \,\%\, 2 \neq ((a \,\%\, 2) + (b \,\%\, 2)) \,\%\, 2\right]$

Hierarchy of different theories

- Equality Logic & Uninterpreted Functions (EUF)
  - Example: $(f(x,y) \neq f(u,v)) \wedge (x = u) \wedge (y = v)$
- Linear Real Arithmetic (LRA)
  - Example: $(x + y \leq 0) \wedge (x = 0) \wedge (\neg a \vee (x = 1) \vee (y \geq 0))$
- Theory of Bit-Vectors (BV)
  - Example: $\left[(a+b) \,\%\, 2 \neq ((a \,\%\, 2) + (b \,\%\, 2)) \,\%\, 2\right]$



Formal Verification in Lugano, Switzerland

Efficient and adoptable to the task decision procedures as computational engines of verification

Gas consumption estimation for smart contracts

Incremental verification

Integrated dynamic and static analysis

Model checking mobile programs

Motivation

- Need for incremental analysis
  - To avoid repetition of same tasks while checking multiple properties of the same code
- Incremental verification
  - Reuse information from one verification run to another
  - Speed-up in consecutive verification runs

HiFrog [TACAS'17]

HiFrog [TACAS'17]

A bounded model checker

- Uses **function summaries** based on interpolation

HiFrog [TACAS'17]

A bounded model checker

- Uses **function summaries** based on interpolation
- With different theory reasoning (SMT-based)

HiFrog [TACAS'17]

A bounded model checker

- Uses **function summaries** based on interpolation
- With different theory reasoning (SMT-based)
- SMT interpolation system w.r.t different first order theories
  - Compact and readable summaries

HiFrog [TACAS'17]

A bounded model checker

- Uses **function summaries** based on interpolation
- With different theory reasoning (SMT-based)
- SMT interpolation system w.r.t different first order theories
  - Compact and readable summaries
- Controllable interpolation system for SMT-theories
  - flexible in Size & Strength

HiFrog [TACAS'17]

A bounded model checker

- Uses **function summaries** based on interpolation
- With different theory reasoning (SMT-based)
- SMT interpolation system w.r.t different first order theories
  - Compact and readable summaries
- Controllable interpolation system for SMT-theories
  - flexible in Size & Strength
- Additional features:
  - User-defined summaries and Assertion optimization

Foundations

Bounded model checking [Biere et al. 1999]

- only look for bugs up to specific depth

– The BMC formula is then checked by using a SAT/SMT procedure

Task: Satisfiability check by a SAT/SMT procedure

SAT : Error found!

- Satisfying assignment identifies an error trace

UNSAT : Program is safe

Function summarization

Function summarization: A technique to create and use over-approximation of the function behavior

- Contains only relevant information to prove properties
- Expressed using function's in/out parameters

Usage

- Same code, different properties
    - To approximate the corresponding functions

Example of summaries in a C program with assertions

```
void main() {
  int y = 1;
  int x = nondet();
  if (x > 0)
    y = f(x);

  assert(y >= 0);
  assert(y >= 1);
}


int f(int a) {
  if (a < 10)
    return a;
  return a – 10;
}
```

USI

Example of summaries in a C program with assertions

```
void main() {
  int y = 1;
  int x = nondet();
  if (x > 0)
    y = f(x);

  assert(y >= 0);
  assert(y >= 1);
}


int f(int a) {
  if (a < 10)
    return a;
  return a – 10;
}
```

Summary

=>

```
(a > 0) ->
(f_return >= 0)
```

Over-approximates real behavior!

Example of summaries in a C program with assertions

```c
void main() {
  int y = 1;
  int x = nondet();
  if (x > 0)
    y = f(x);

  assert(y >= 0);
  assert(y >= 1);
}


int f(int a) {
  if (a < 10)
    return a;
  return a – 10;
}
```

=>

```c
void main() {
  int y = 1;
  int x = nondet();

  if (x > 0){
    assume(y >= 0);
  }
  assert(y >= 0);
  assert(y >= 1);
}
```

Use of Summary

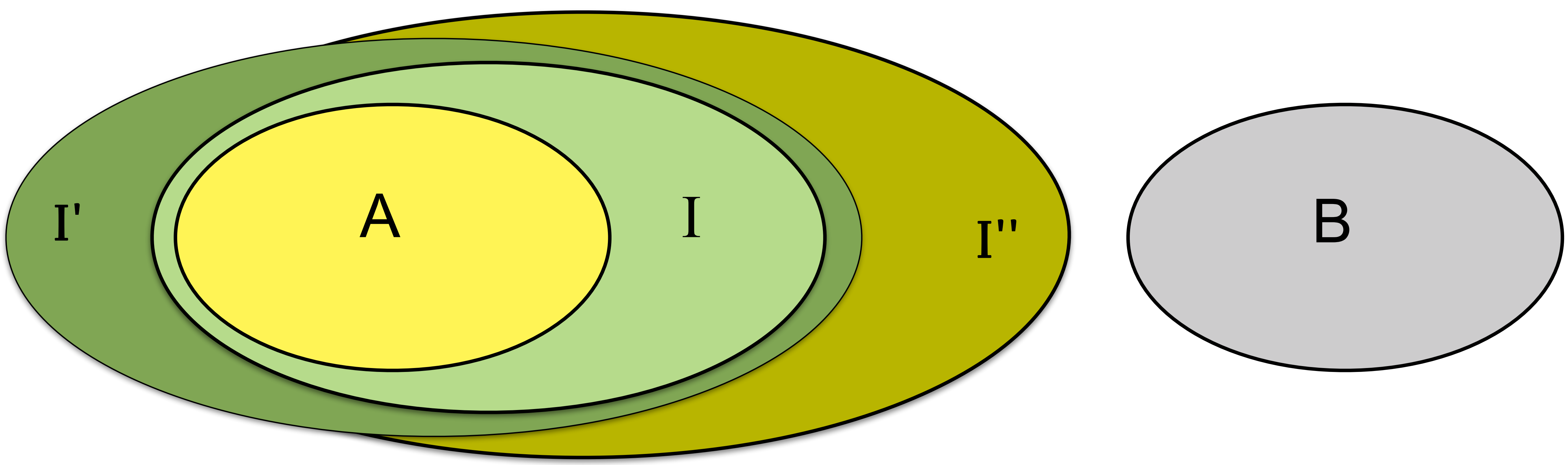
=>

```
(a > 0) ->
(f_return >= 0)
```

Craig interpolation [Craig '57]

Definition:

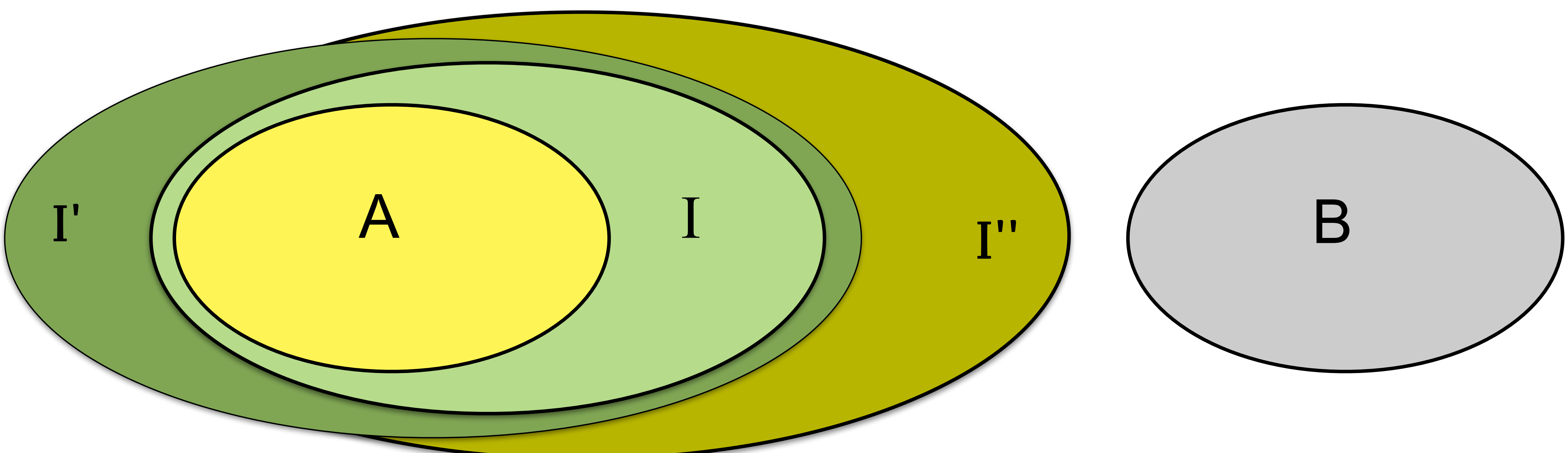
- Given mutually unsatisfiable formulas A and B, an *Interpolant* is a formula I such that
  - $A \rightarrow I$
  - $I \wedge B$ is unsatisfiable
  - I is defined over common symbols of both A and B



Craig interpolation [Craig '57]

Definition:

- Given mutually unsatisfiable formulas A and B, an *Interpolant* is a formula I such that
  - $A \rightarrow I$
  - $I \wedge B$ is unsatisfiable
  - I is defined over common symbols of both A and B



I is over-approximation of A, still unsatisfiable with B

Interpolation-based function summarization

Apply Craig interpolation after SMT-solver returns UNSAT

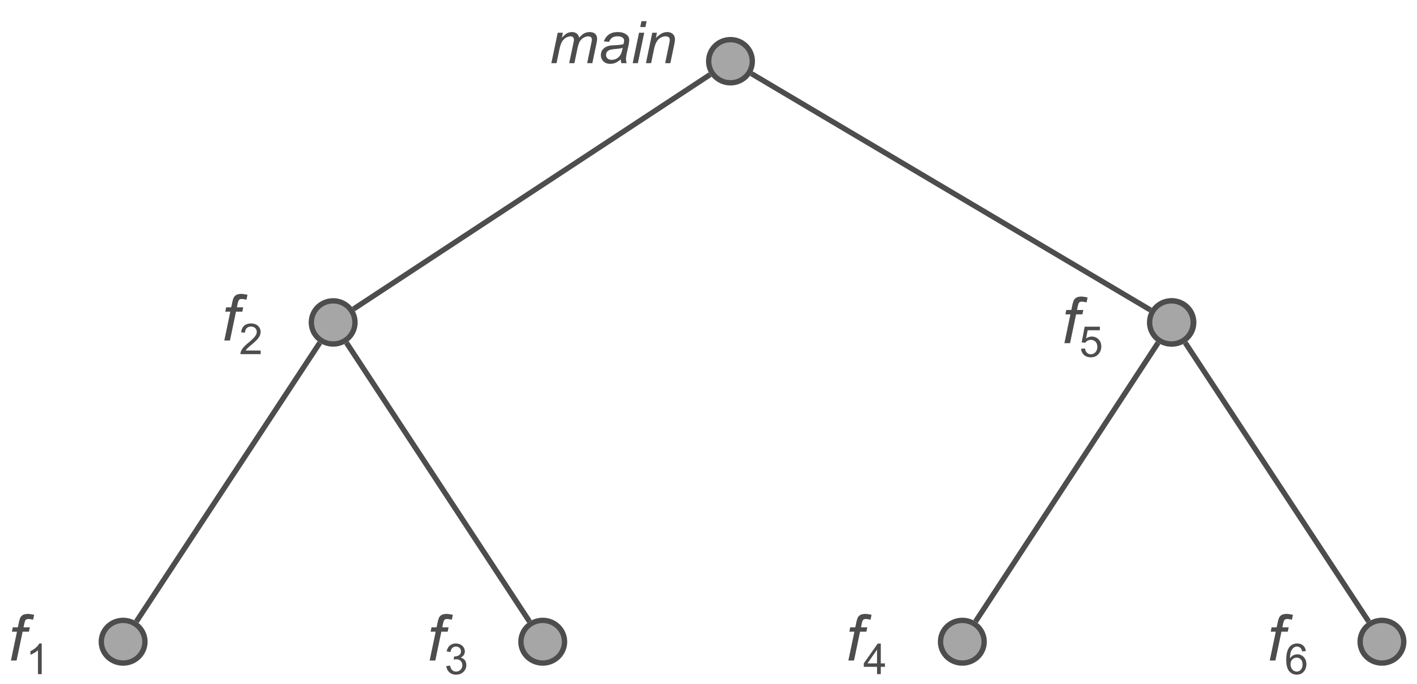
- Iterative procedure over the set of function calls

How to use interpolation for extracting function summarization

unwound program → partitioned bounded model checking (PBMC)

Partitioning BMC

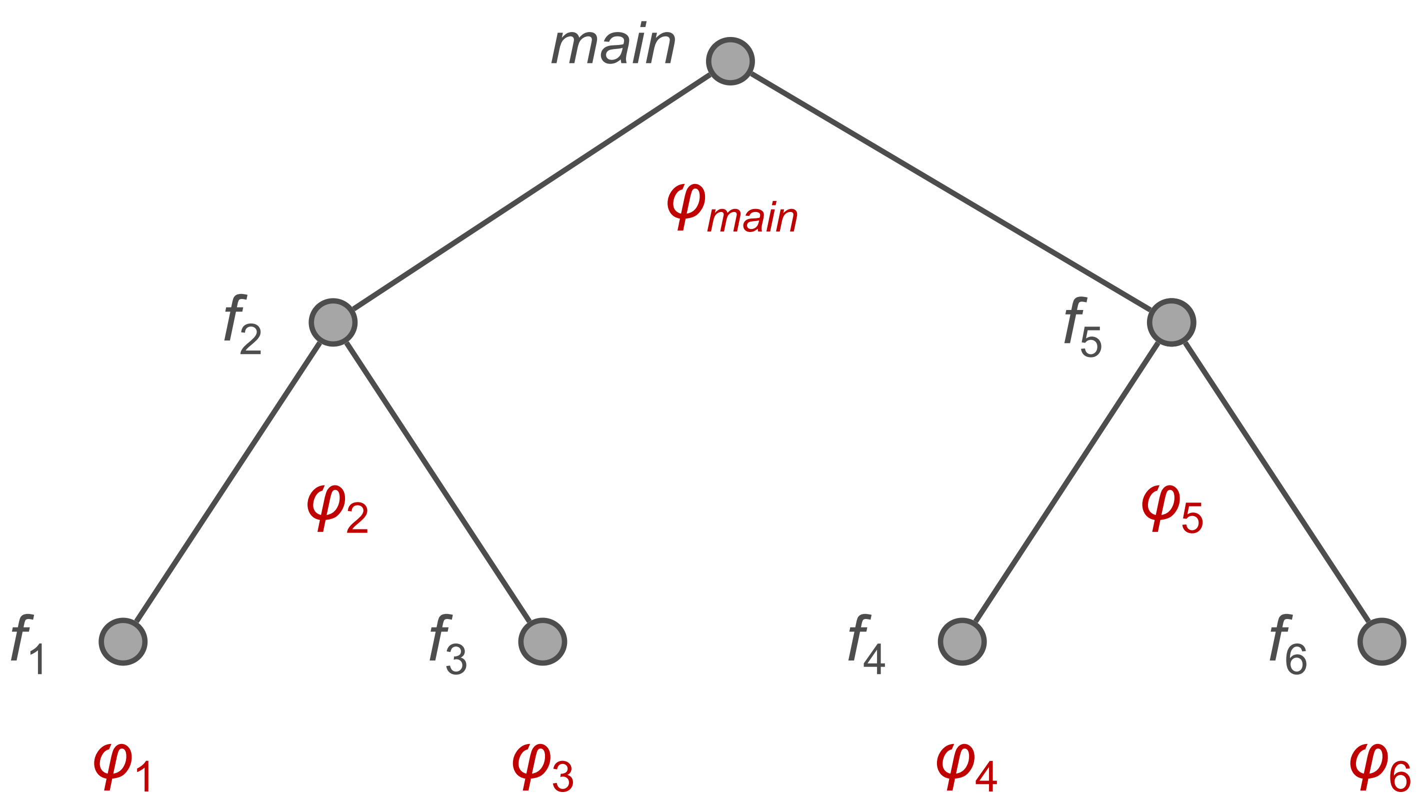
Formula construction



BMC formula created in a partitioned way: each partition represents the body of a function

Partitioning BMC

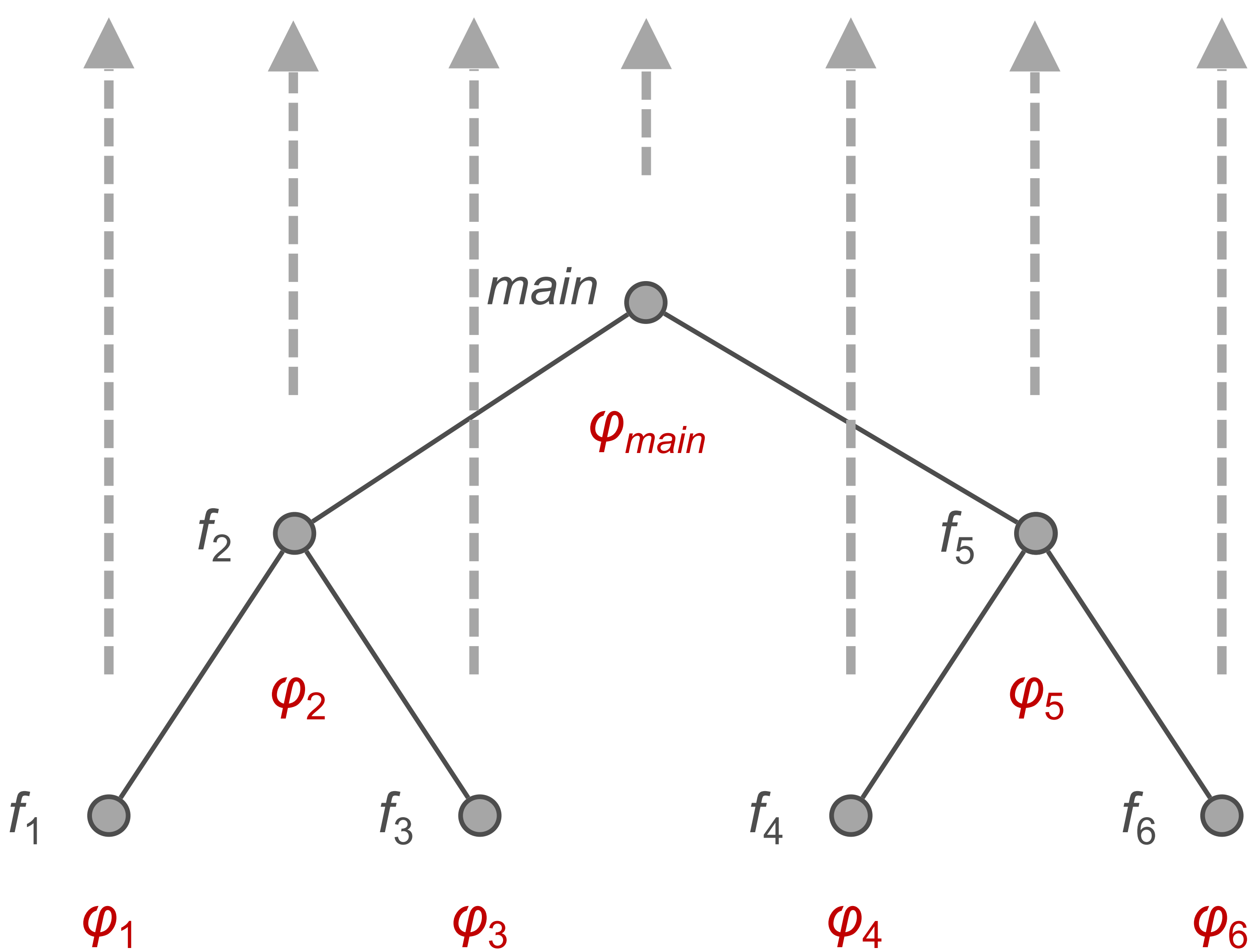
Formula construction



BMC formula created in a partitioned way: each partition represents the body of a function

Partitioning BMC

Formula construction

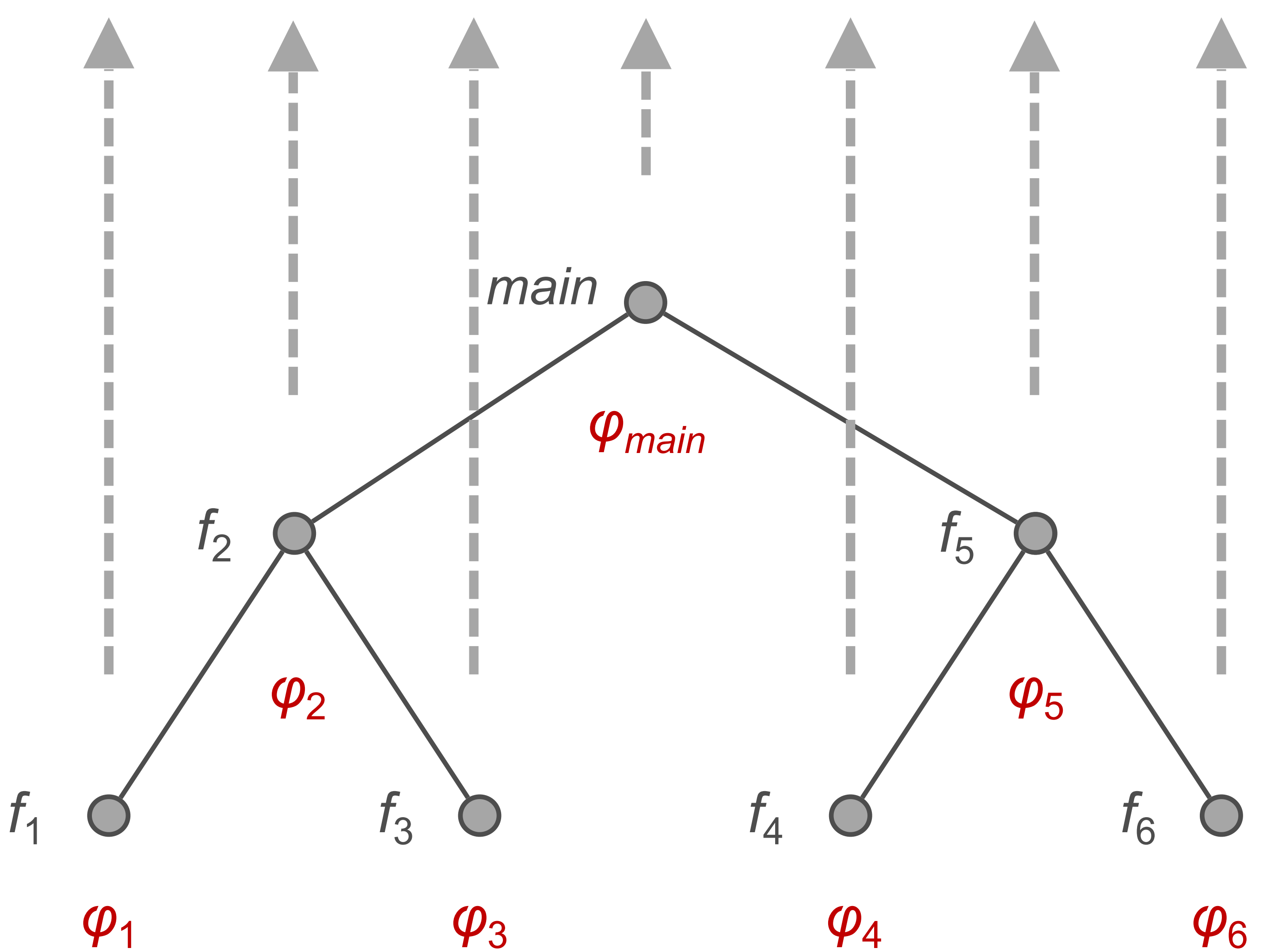


BMC formula created in a partitioned way: each partition represents the body of a function

Partitioning BMC

Formula construction

$$\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_{main} \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge error_{main}$$

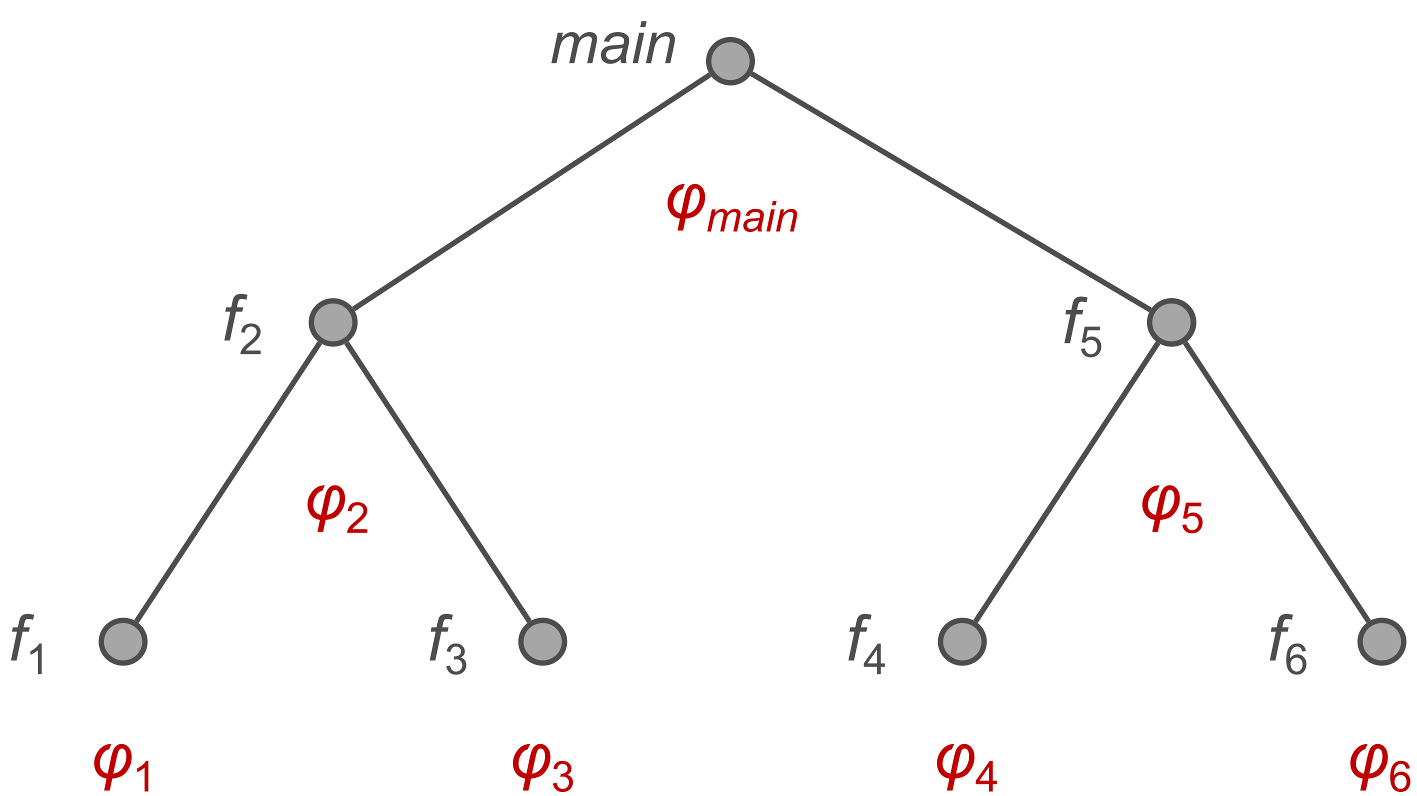


BMC formula created in a partitioned way: each partition represents the body of a function

Partitioning BMC

Formula construction

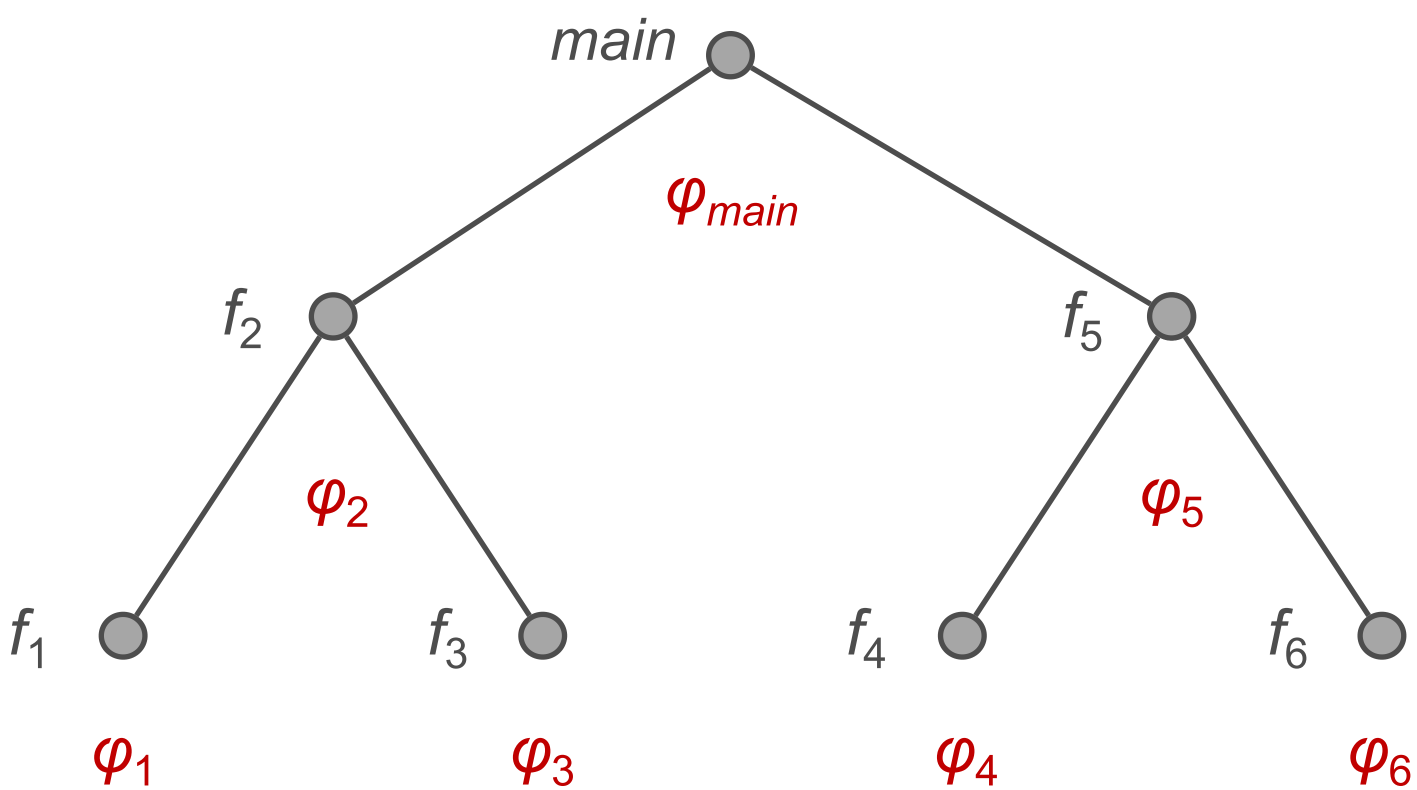
$\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_{main} \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge error_{main}$ ← UNSAT



BMC formula created in a partitioned way: each partition represents the body of a function

Partitioning BMC

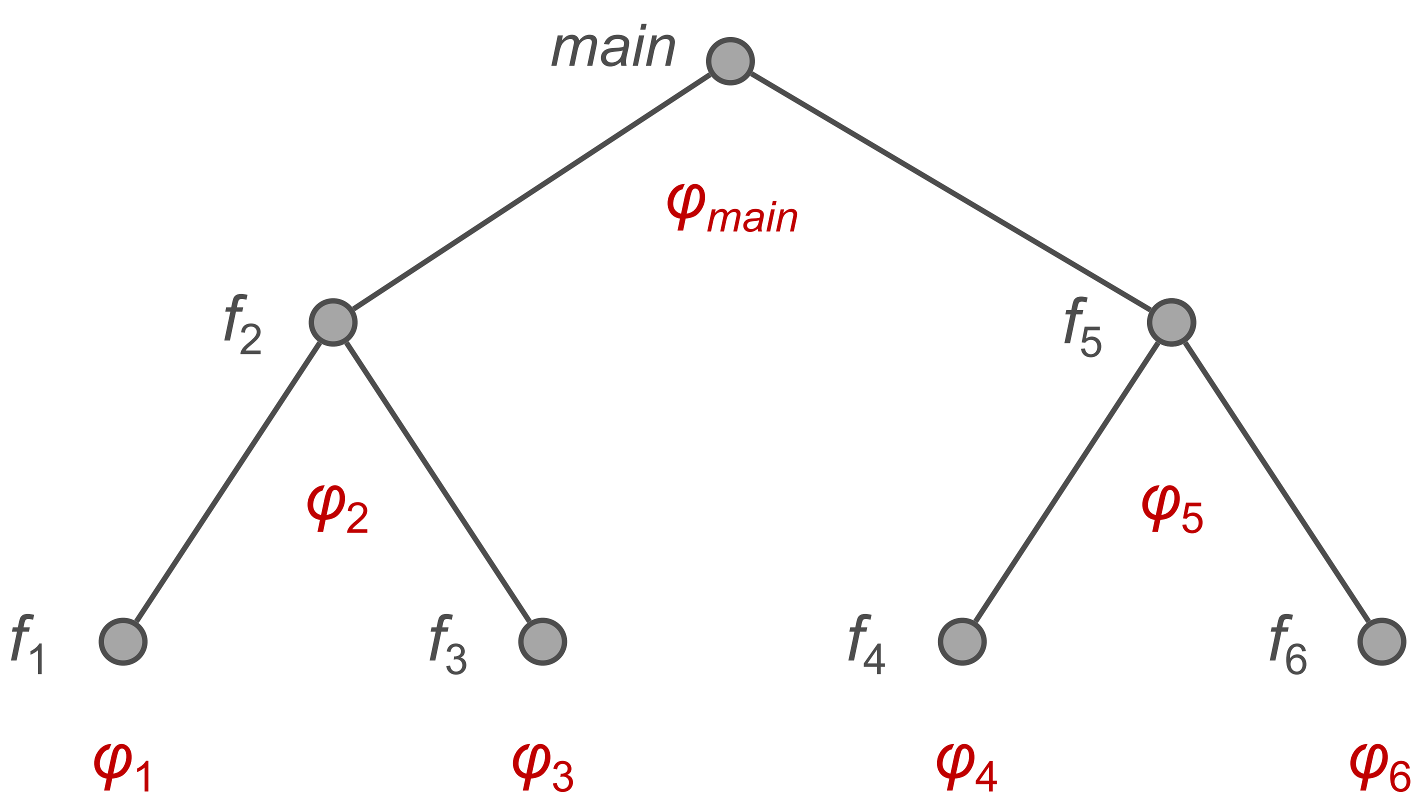
$\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_{main} \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge error_{main}$ ← UNSAT



Partitioning BMC

Generation of summaries

$\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_{main} \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge error_{main}$ ← UNSAT

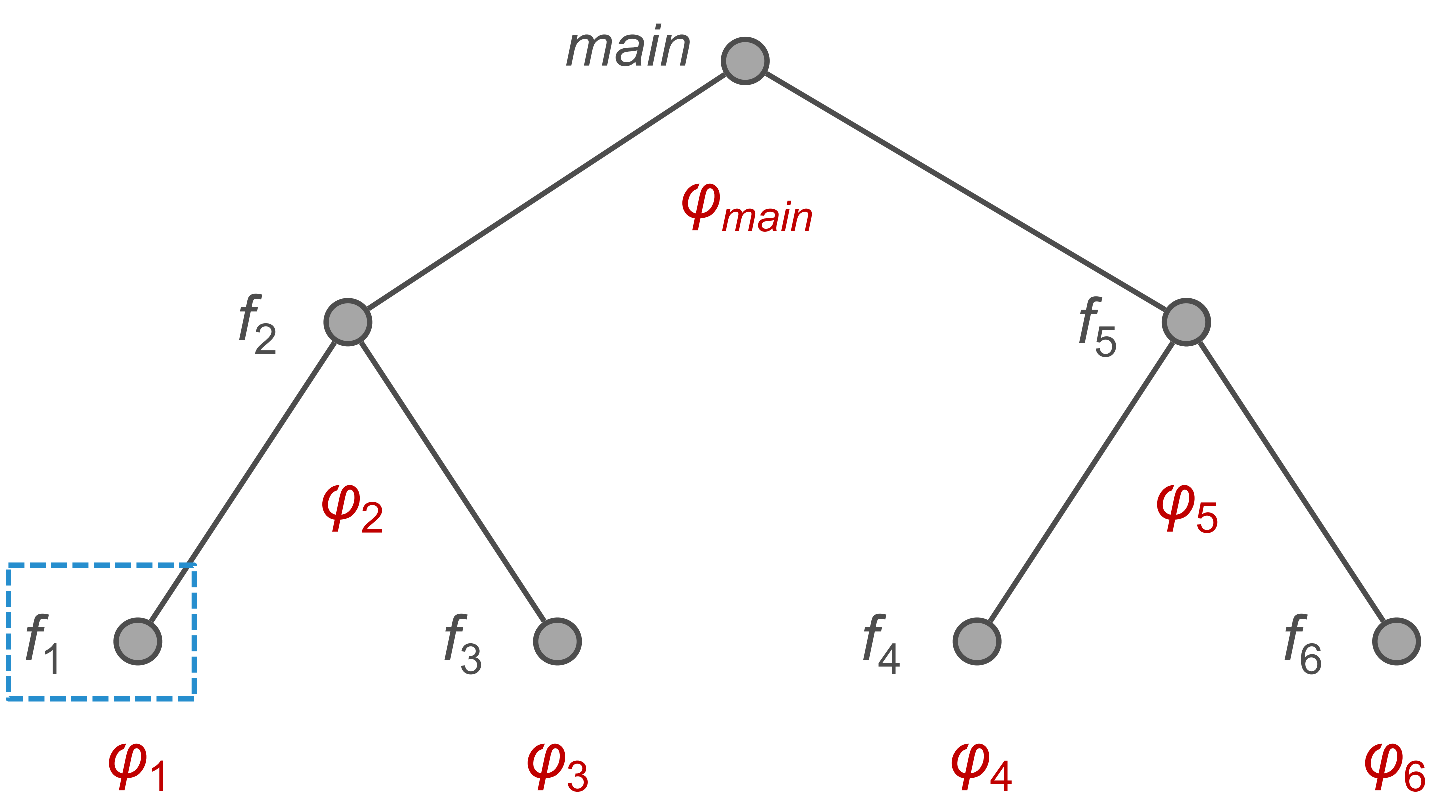


USI

Partitioning BMC

Generation of summaries

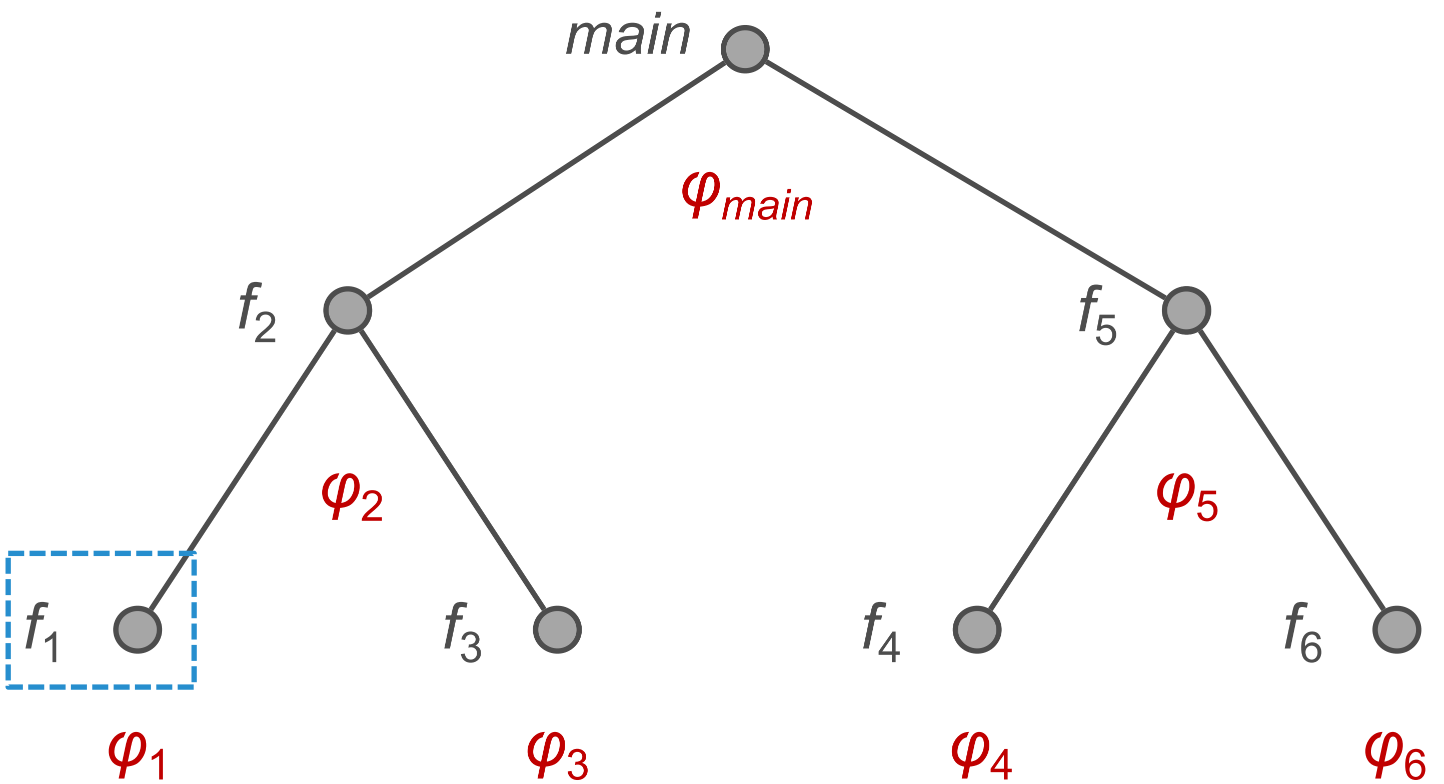
$\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_{main} \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge error_{main}$ ← UNSAT



Partitioning BMC

Generation of summaries

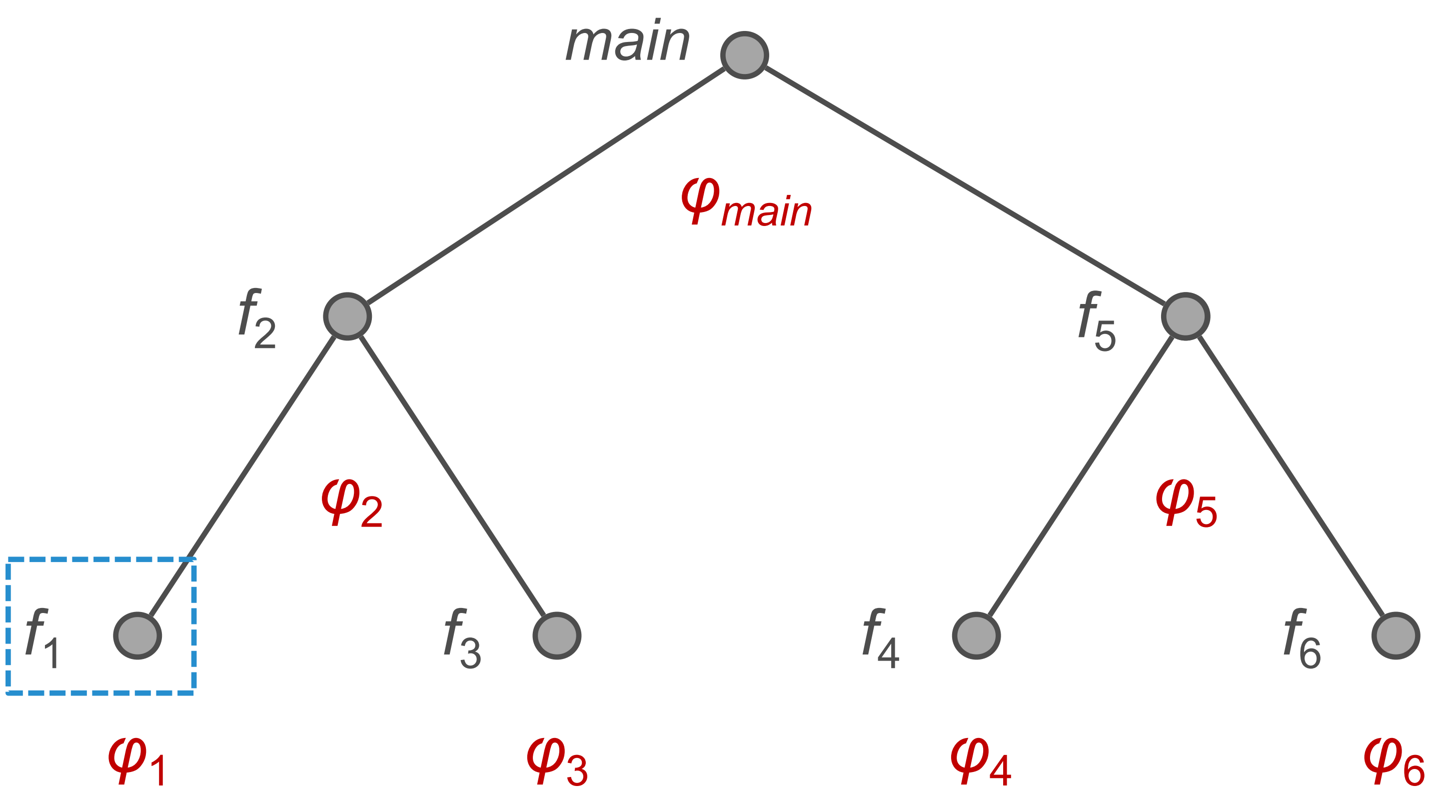
$$\underbrace{\varphi_1}^{A} \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_{main} \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge error_{main} \longleftarrow \text{UNSAT}$$



Partitioning BMC

Generation of summaries

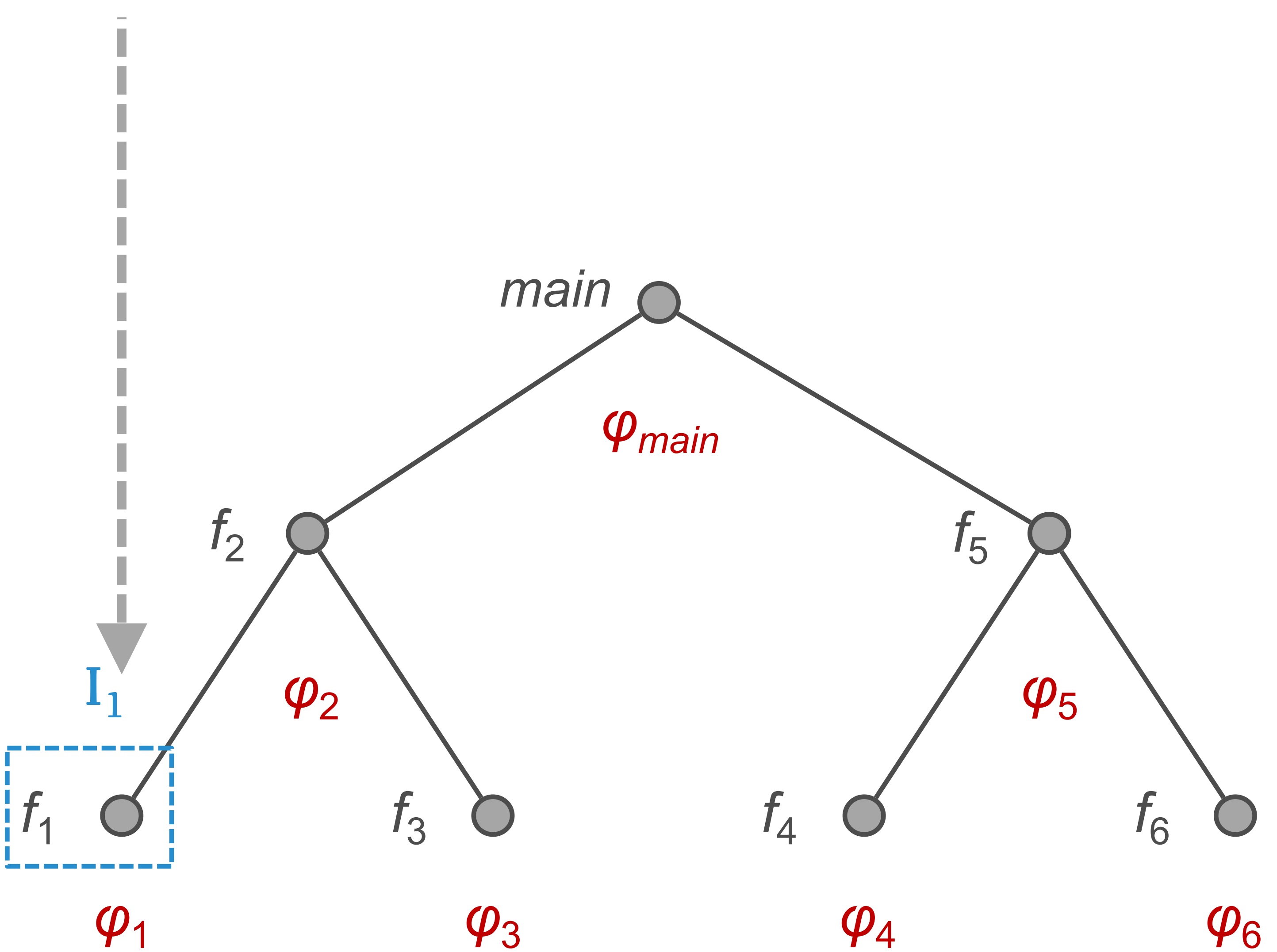
$$\overbrace{\varphi_1}^{A} \wedge \overbrace{\varphi_2 \wedge \varphi_3 \wedge \varphi_{main} \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge error_{main}}^{B}$$ UNSAT



Partitioning BMC

Generation of summaries

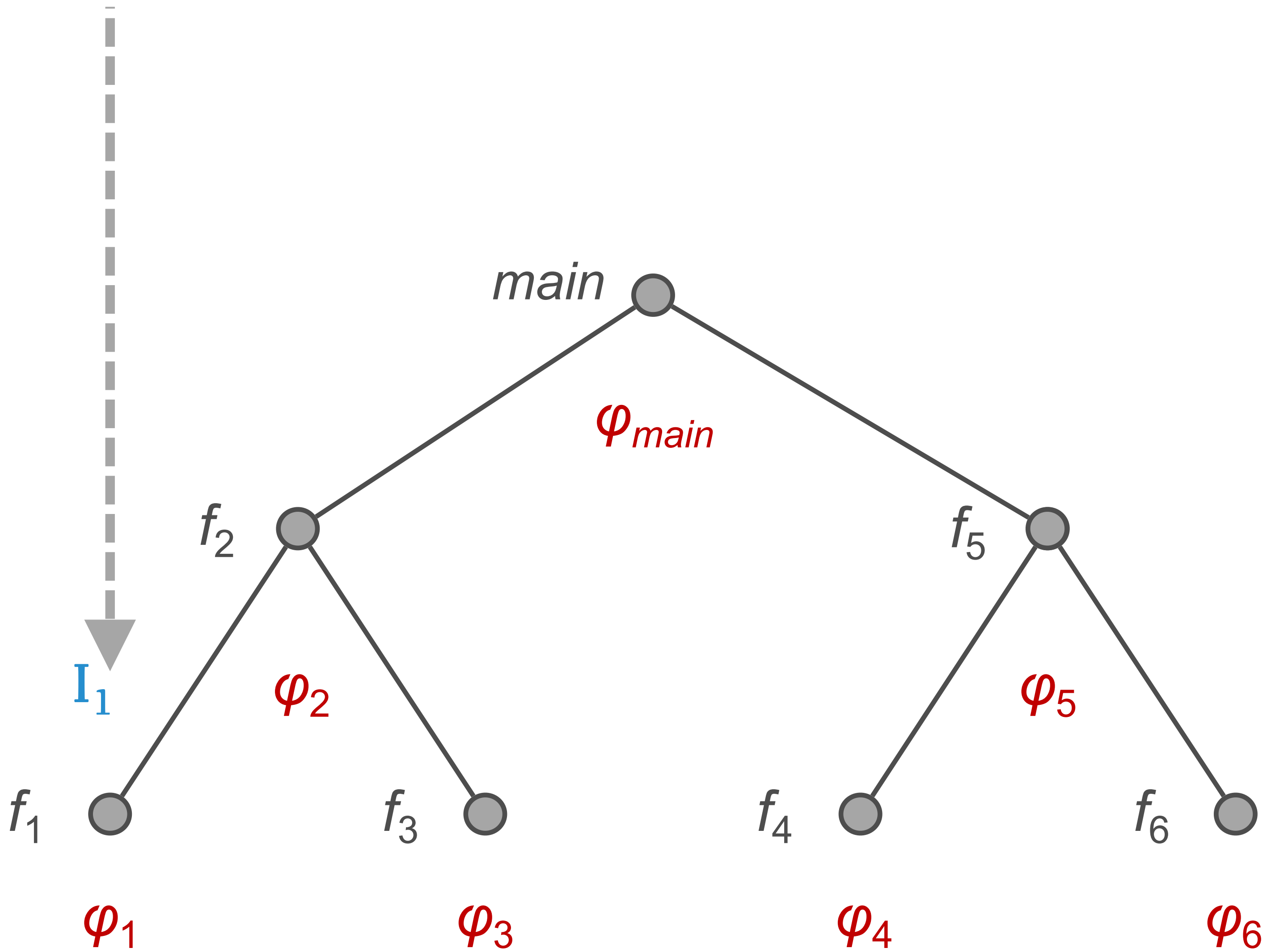
$$\underbrace{\varphi_1}_{A} \wedge \underbrace{\varphi_2 \wedge \varphi_3 \wedge \varphi_{main} \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge error_{main}}_{B}$$ UNSAT



Partitioning BMC

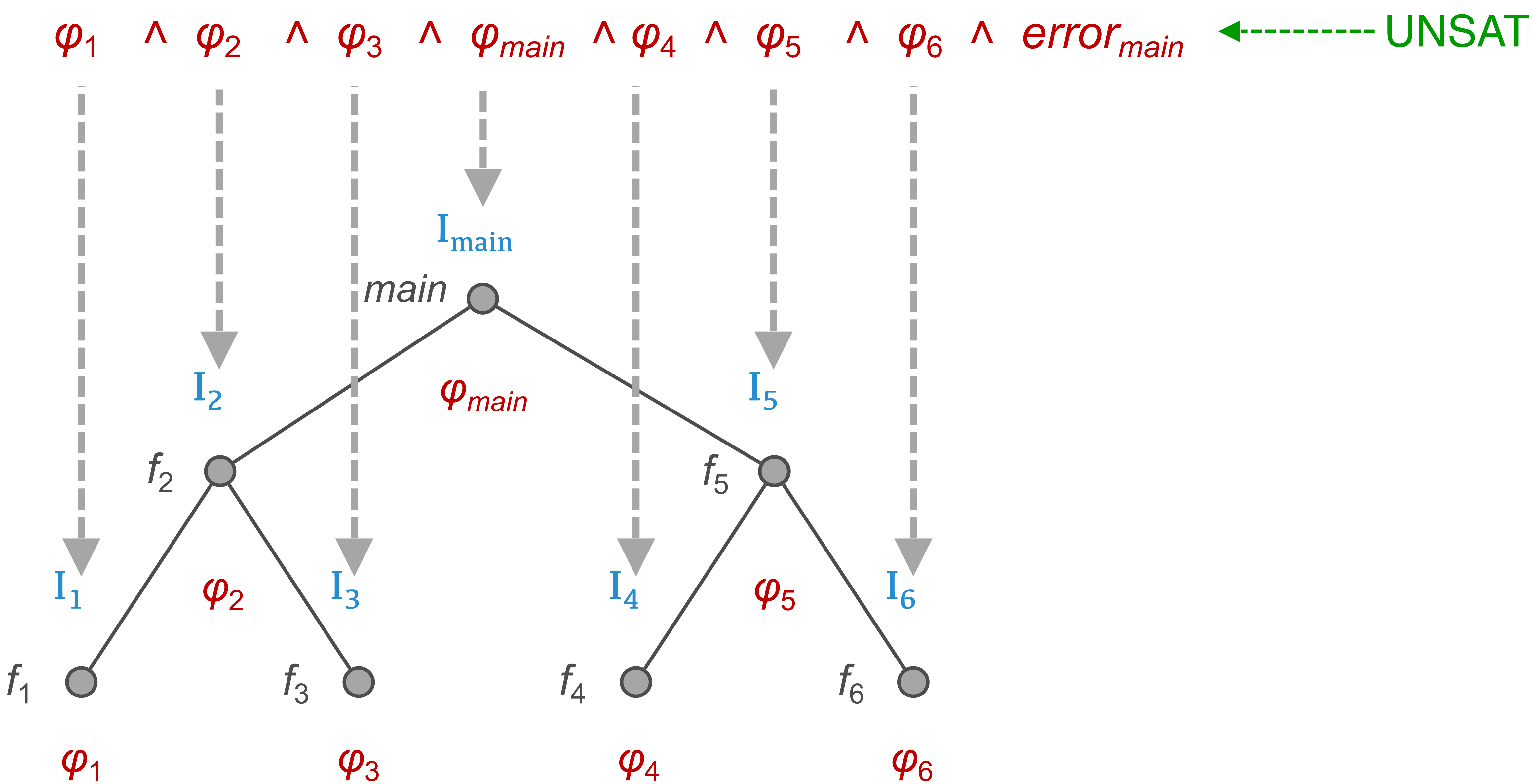
Generation of summaries

$\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_{main} \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge error_{main}$ ← UNSAT



Partitioning BMC

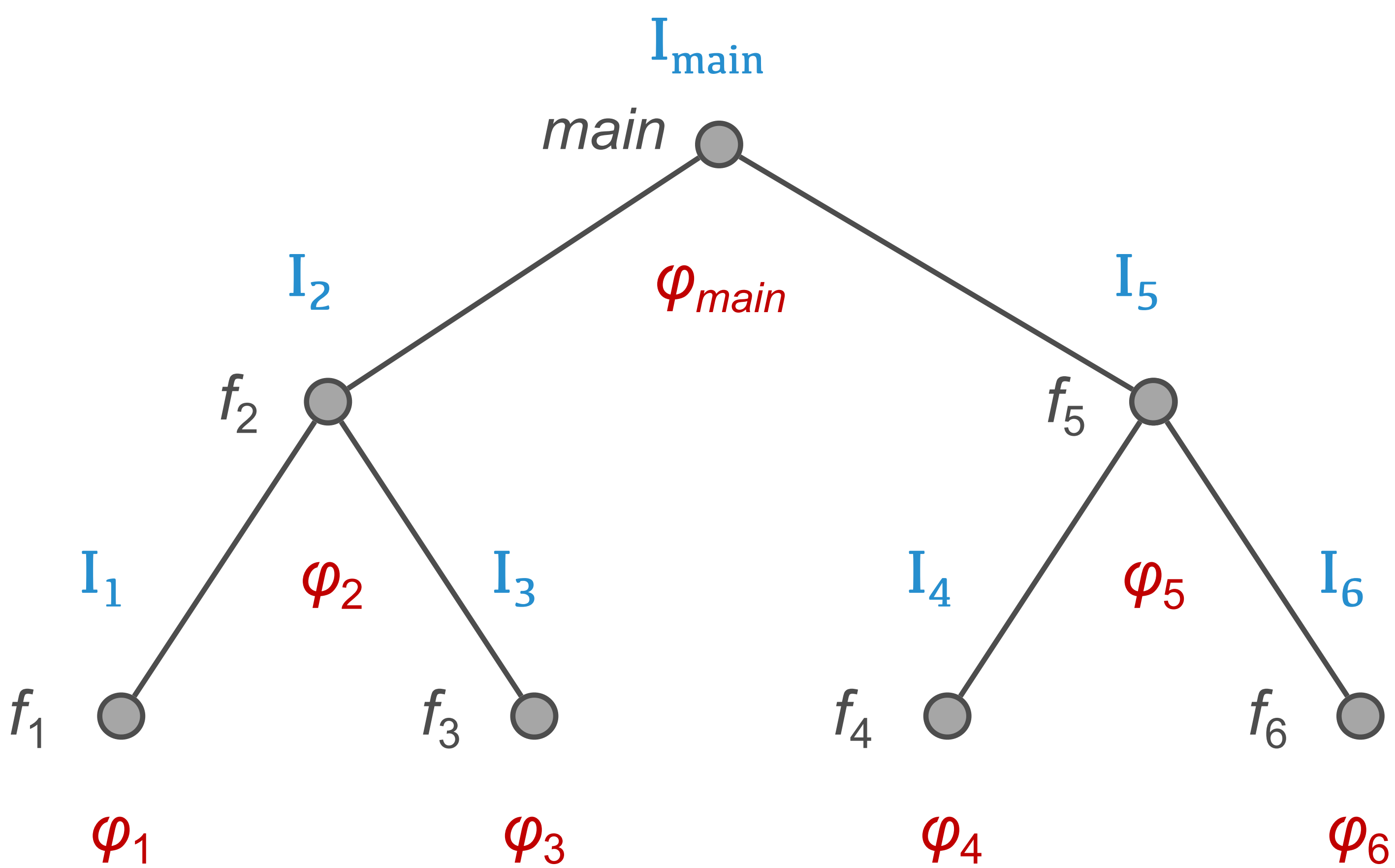
Generation of summaries



Partitioning BMC

Generation of summaries

$\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_{main} \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge error_{main}$ ← UNSAT



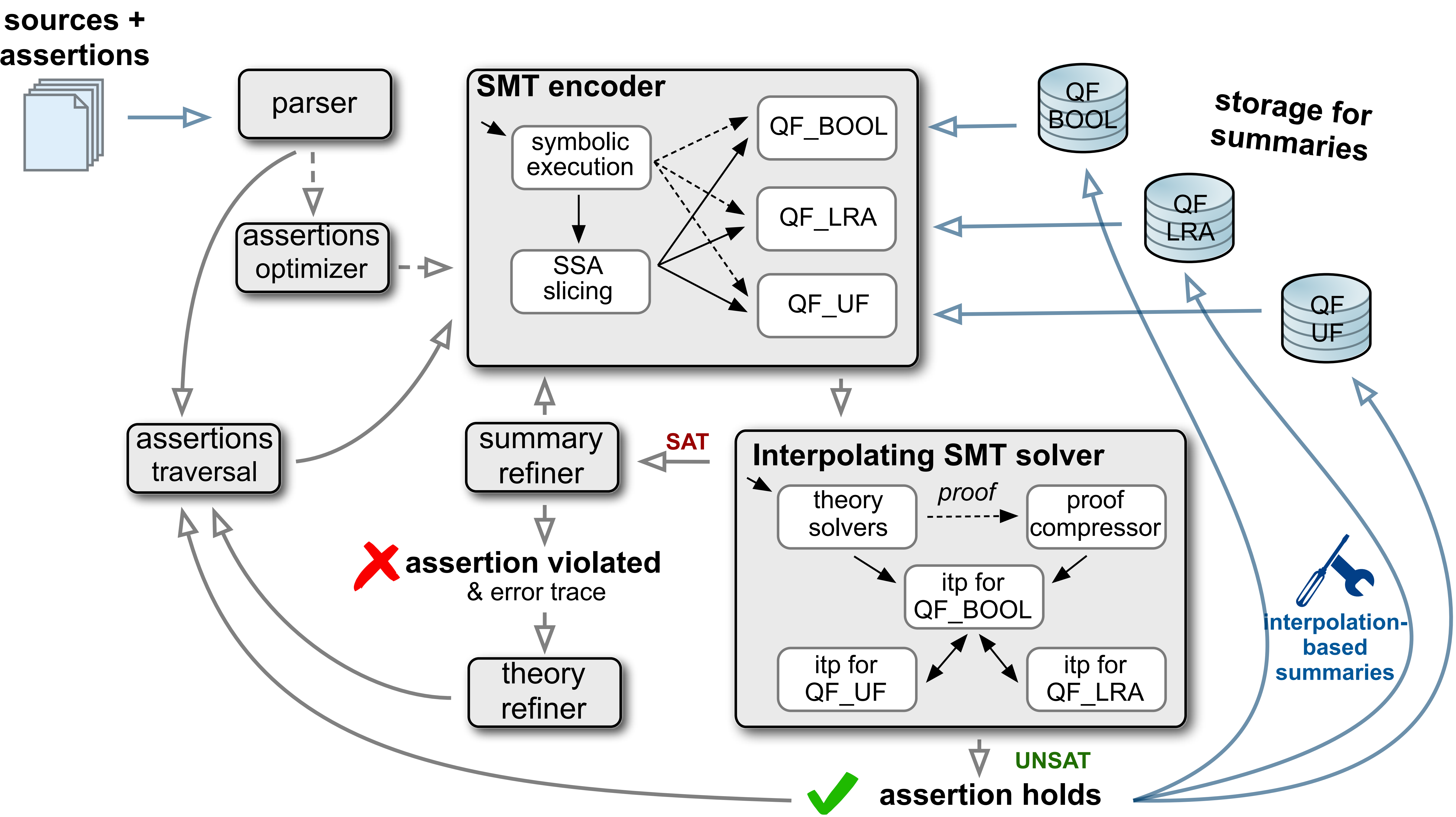
Underlying technology

- HiFrog - model checker for C
  - https://scm.ti-edu.ch/projects/hifrog

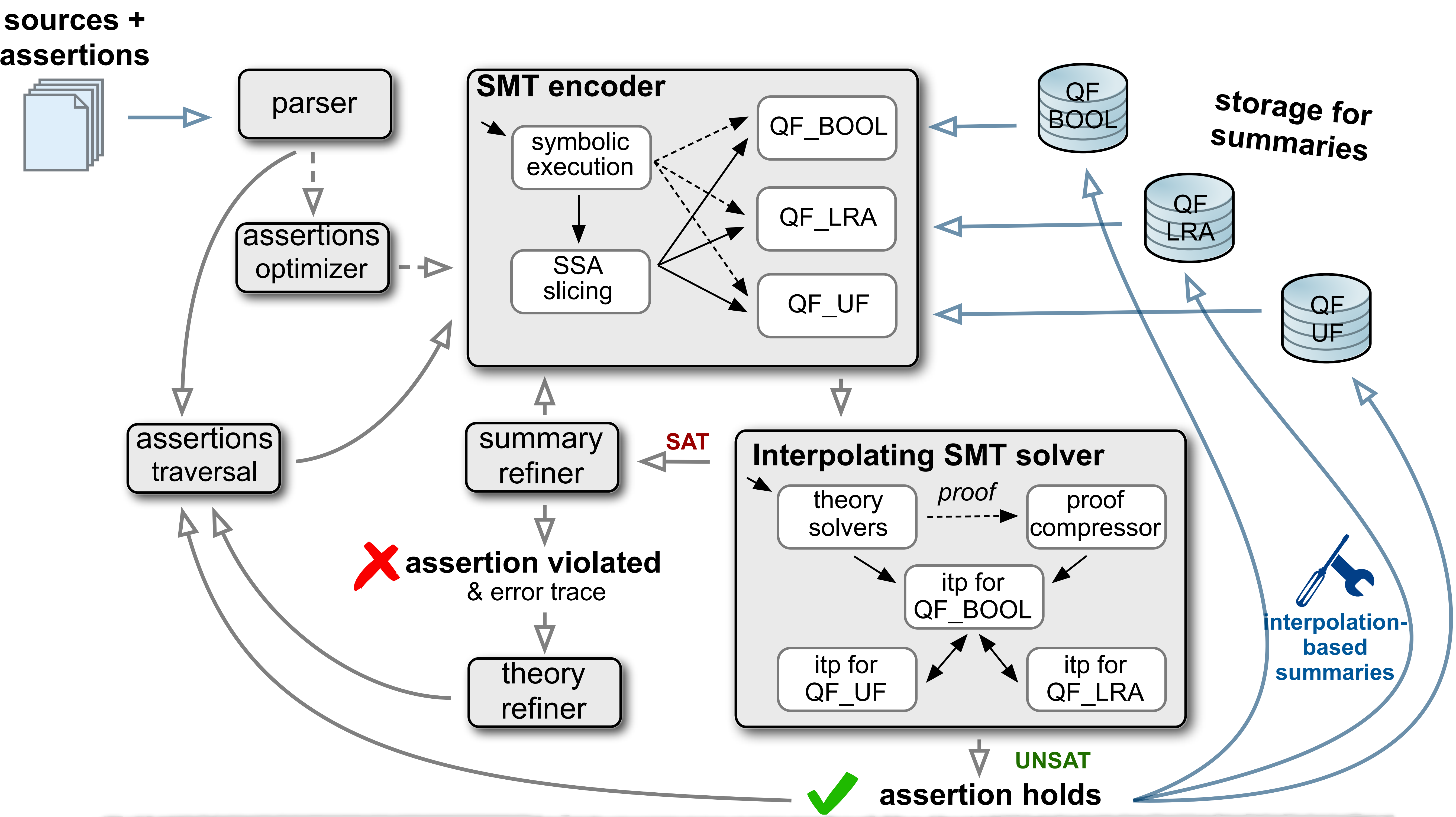
- Uses CProver framework for symbolic encoding of C programs
  - http://cprover.org [Kroening et al.]

- Employs our open-source SMT-solver OpenSMT2
  - For SMT checks & interpolation http://verify.inf.usi.ch/opensmt

HiFrog Architecture



HiFrog Architecture



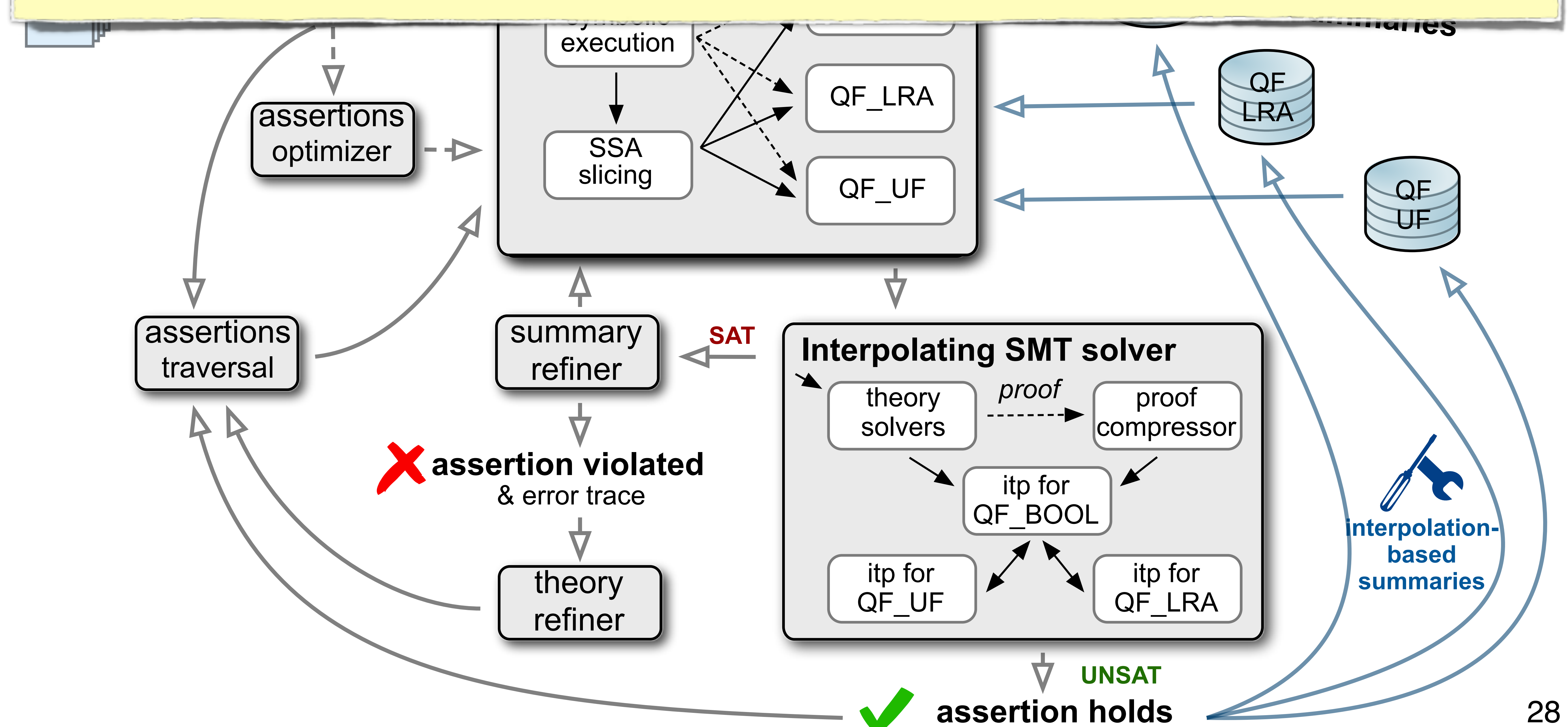
If successful, HiFrog updates function summaries for next checks
If unsuccessful, after refinement HiFrog reports violation + an error trace

If the SMT formula is SAT

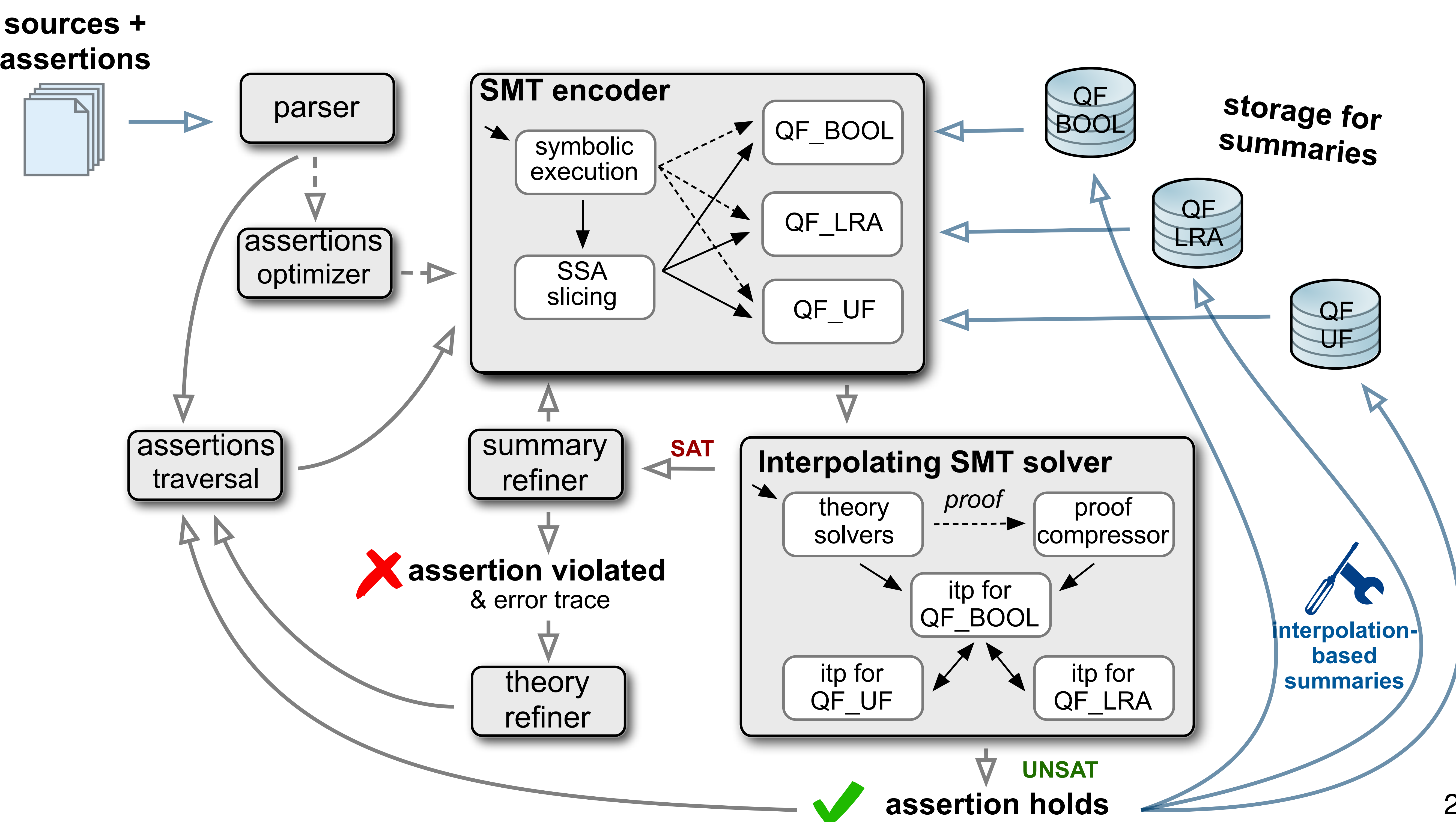
— Maybe the reachable error is spurious due to over-approximation of summaries

Solution: Refine the abstraction!

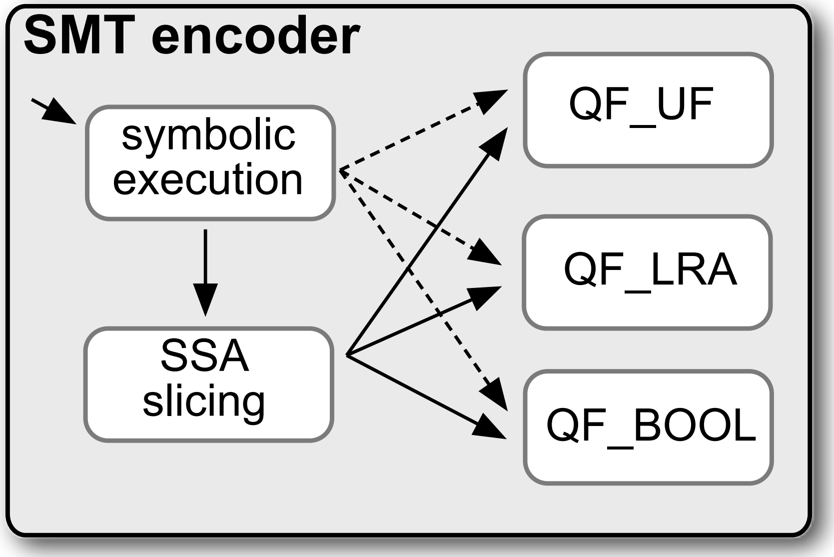
- By Error trace analysis identify summaries that appears along the error trace
- Replace summaries by precise representation



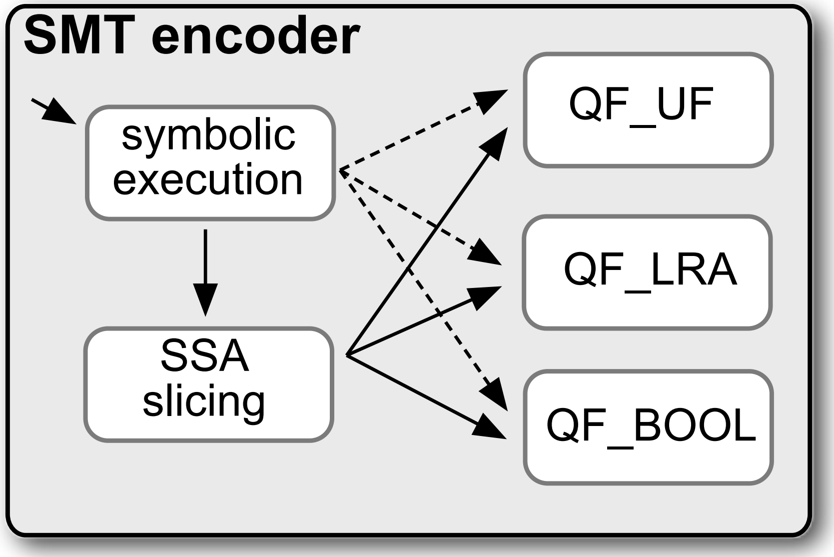
HiFrog Architecture



Different encoding precisions through SMT theories



Different encoding precisions through SMT theories

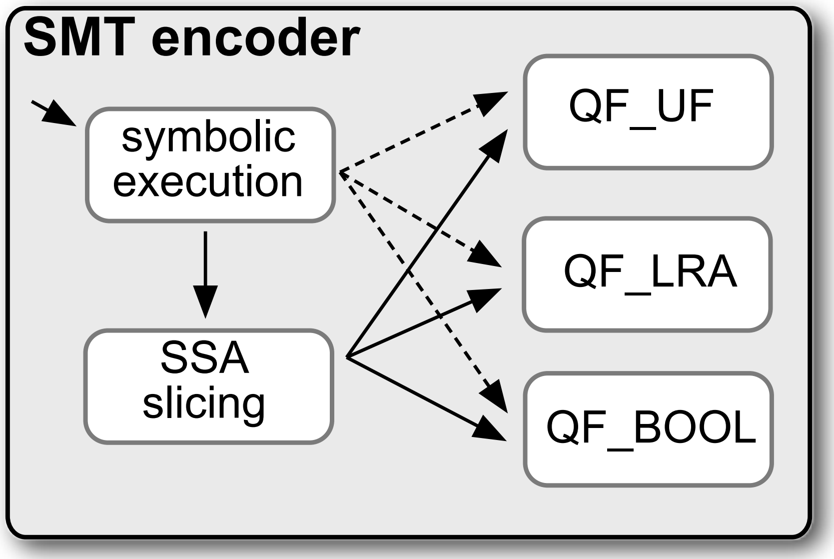


Uninterpreted Functions (QF_UF)

Linear Real Arithmetics(QF_LRA)

Propositional logic (QF_BOOL)

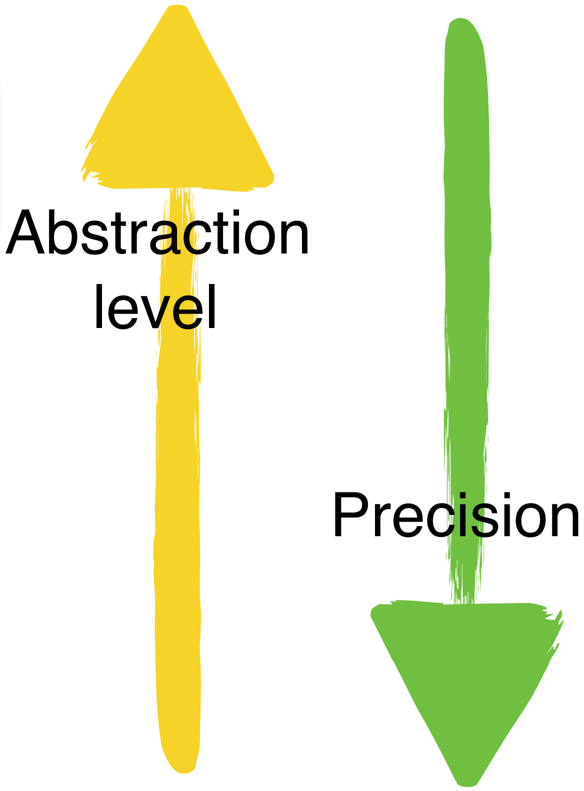
Different encoding precisions through SMT theories



Uninterpreted Functions (QF_UF)

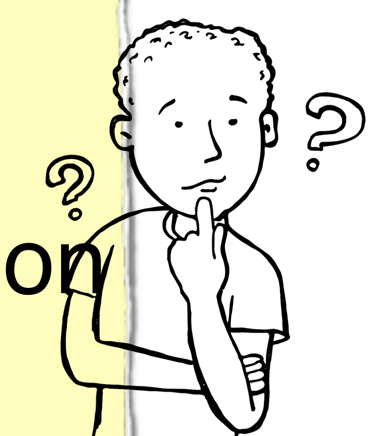
Linear Real Arithmetics(QF_LRA)

Propositional logic (QF_BOOL) 

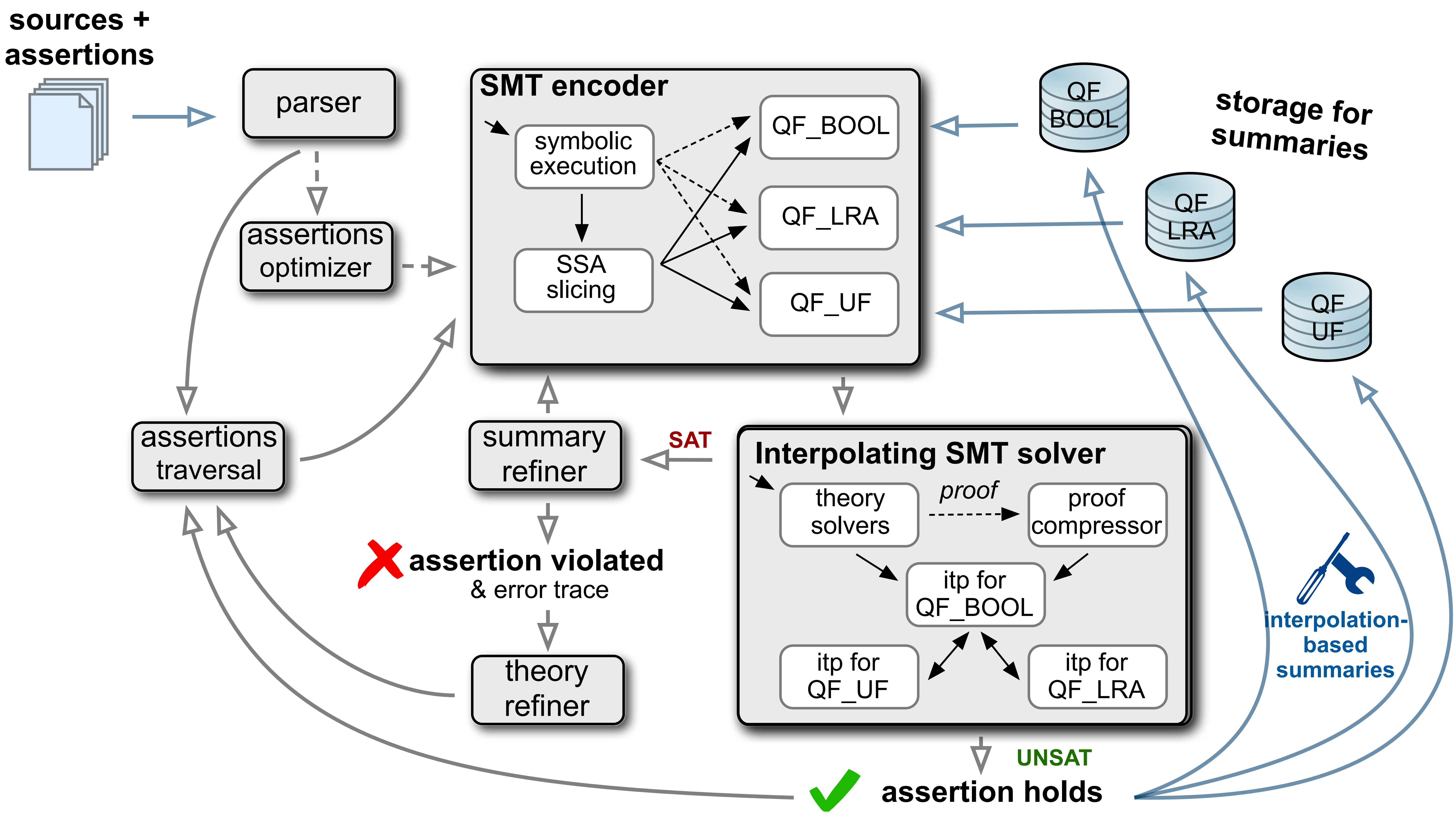


Trade off between level of abstraction and precision!

A key factor for success is to find a level of abstraction that is sufficiently precise but not too expensive to reason on

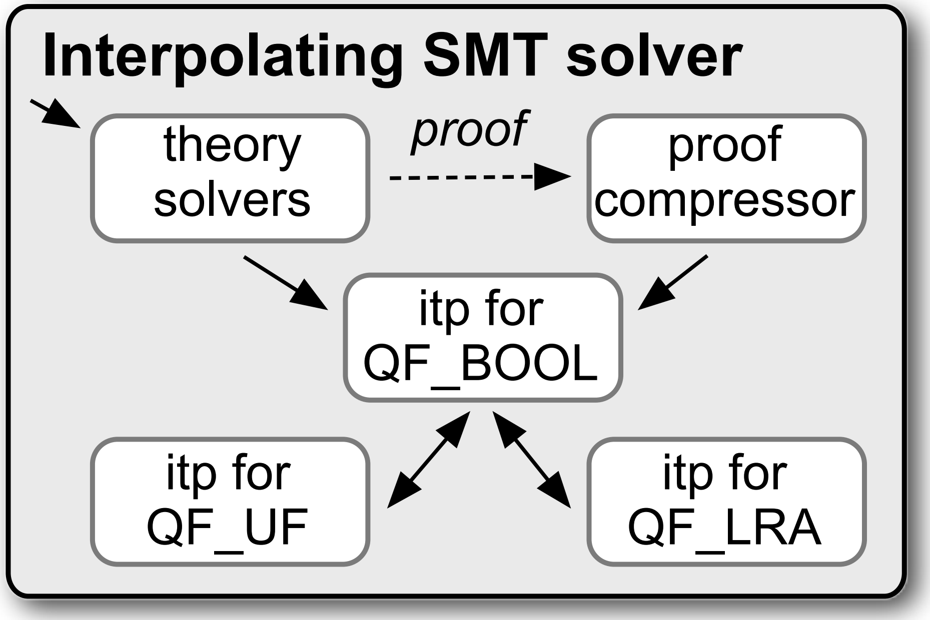


HiFrog Architecture



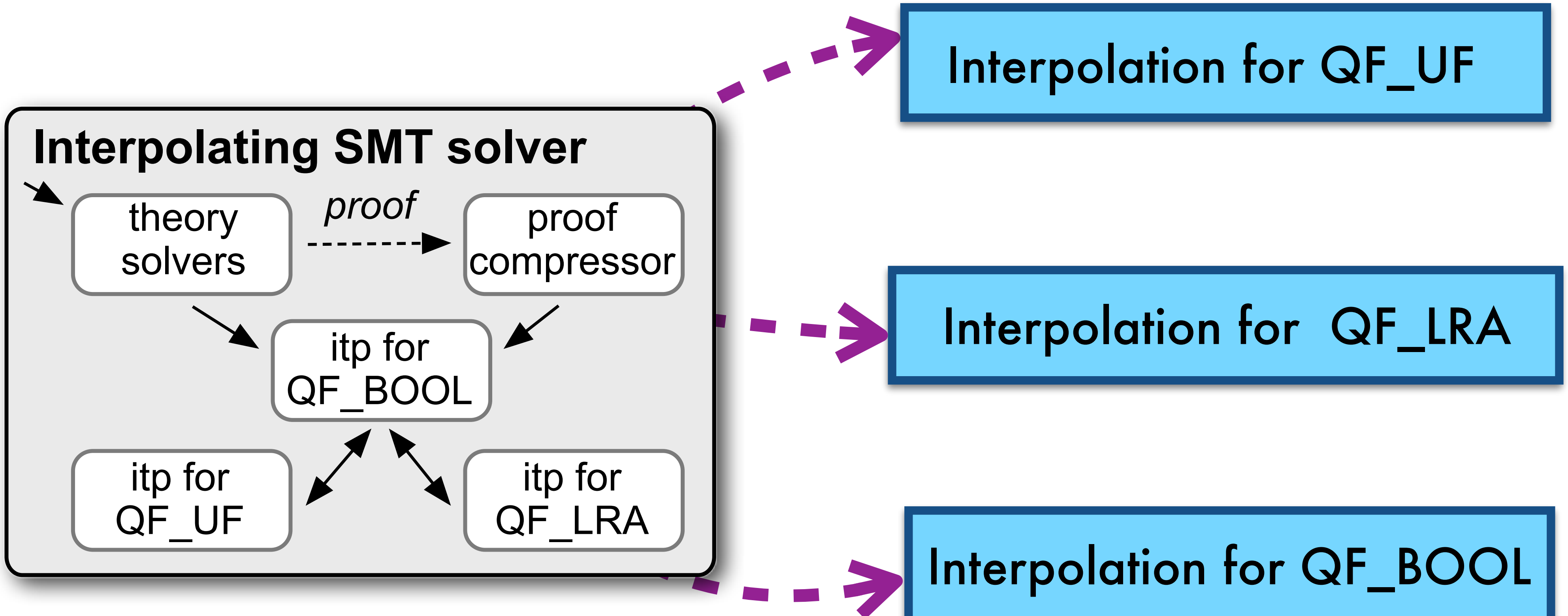
Interpolation for various theories

Each theory has its own interpolation procedure



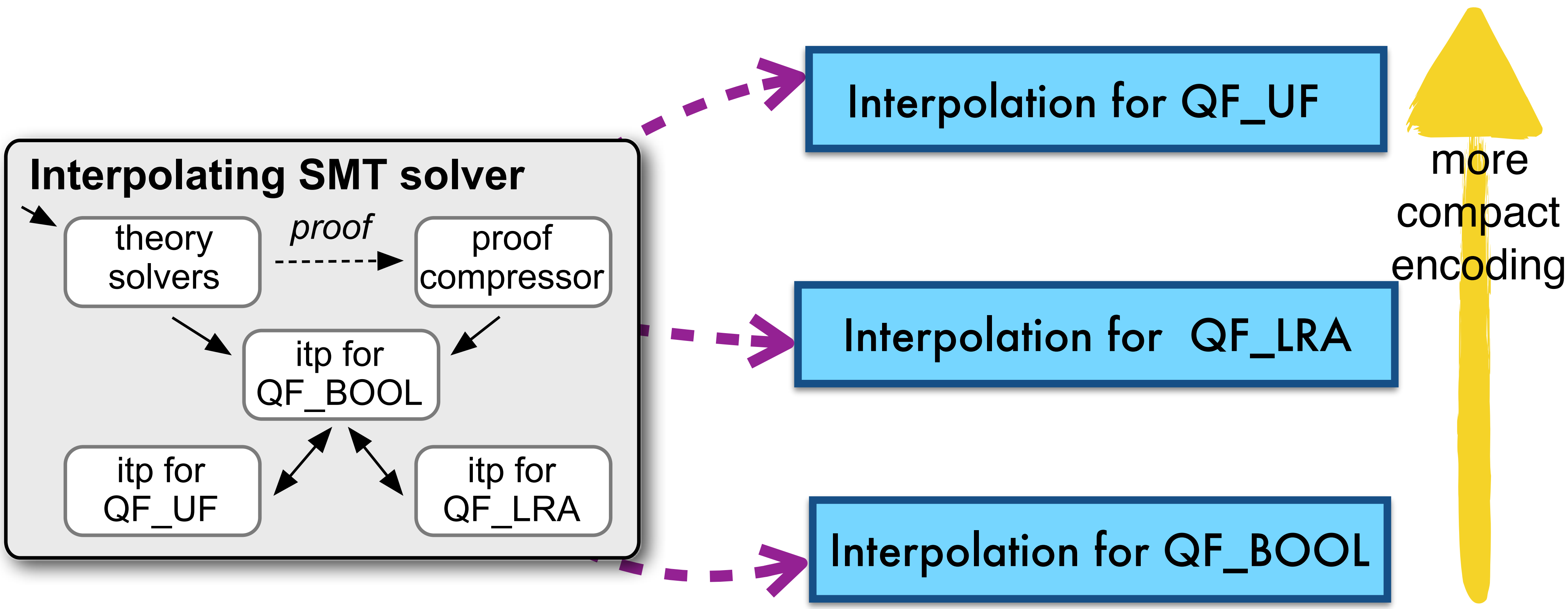
Interpolation for various theories

Each theory has its own interpolation procedure



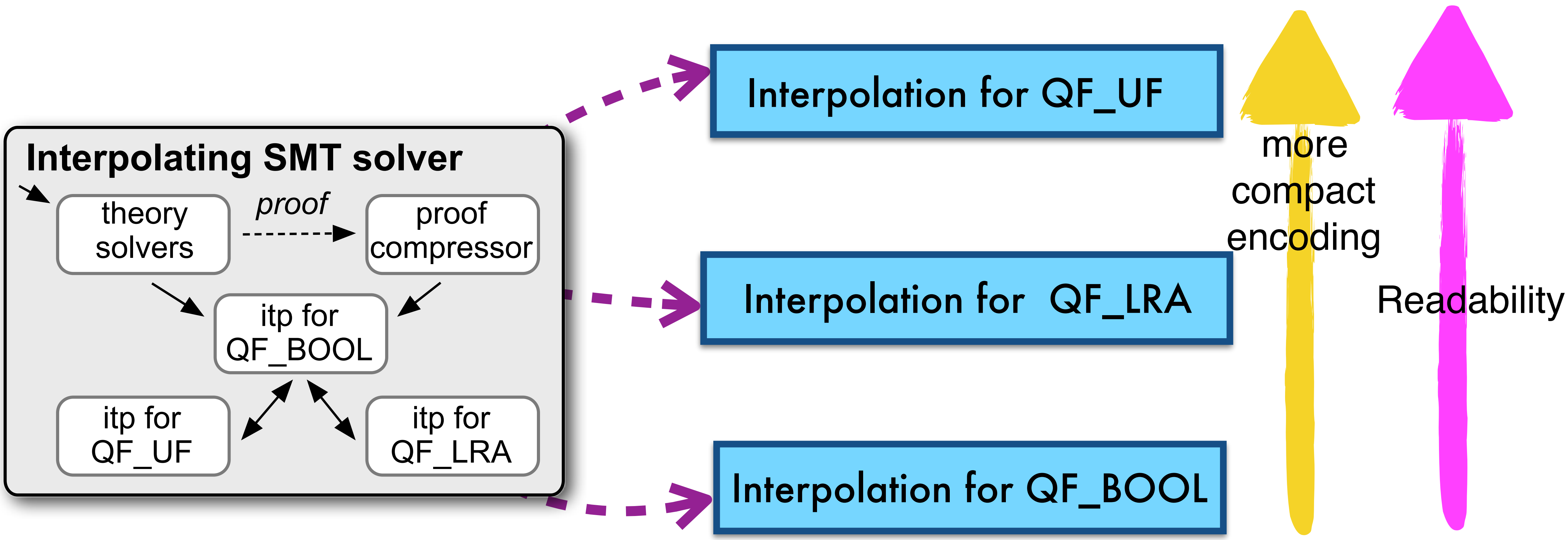
Interpolation for various theories

Each theory has its own interpolation procedure



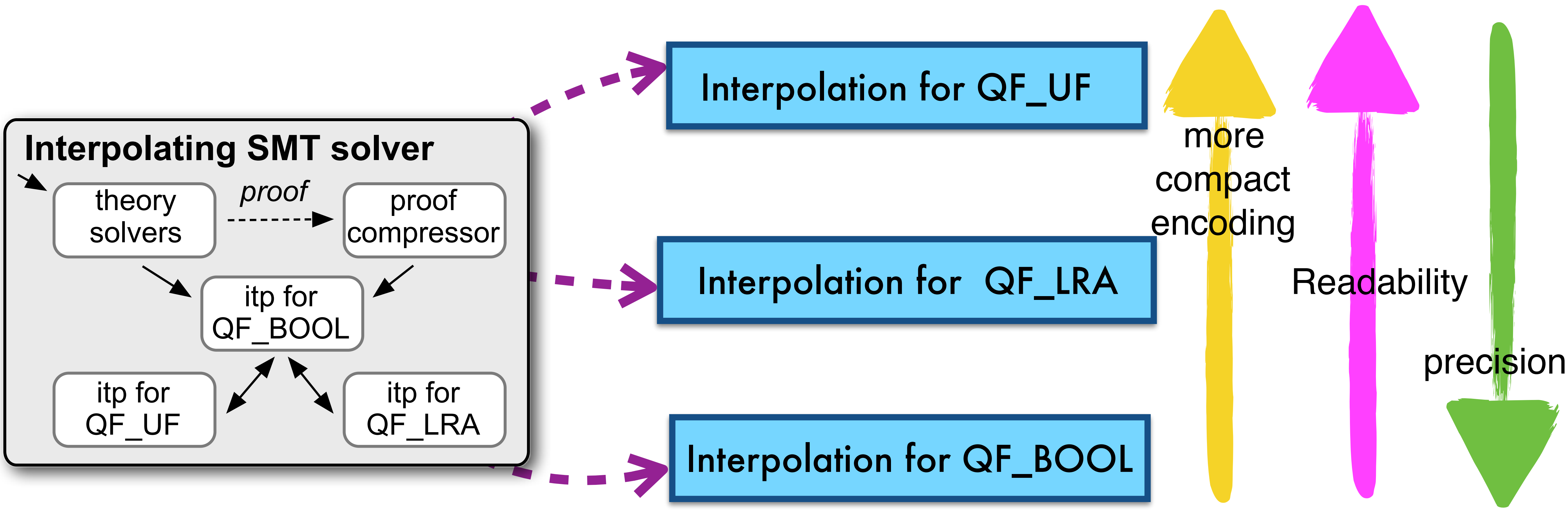
Interpolation for various theories

Each theory has its own interpolation procedure



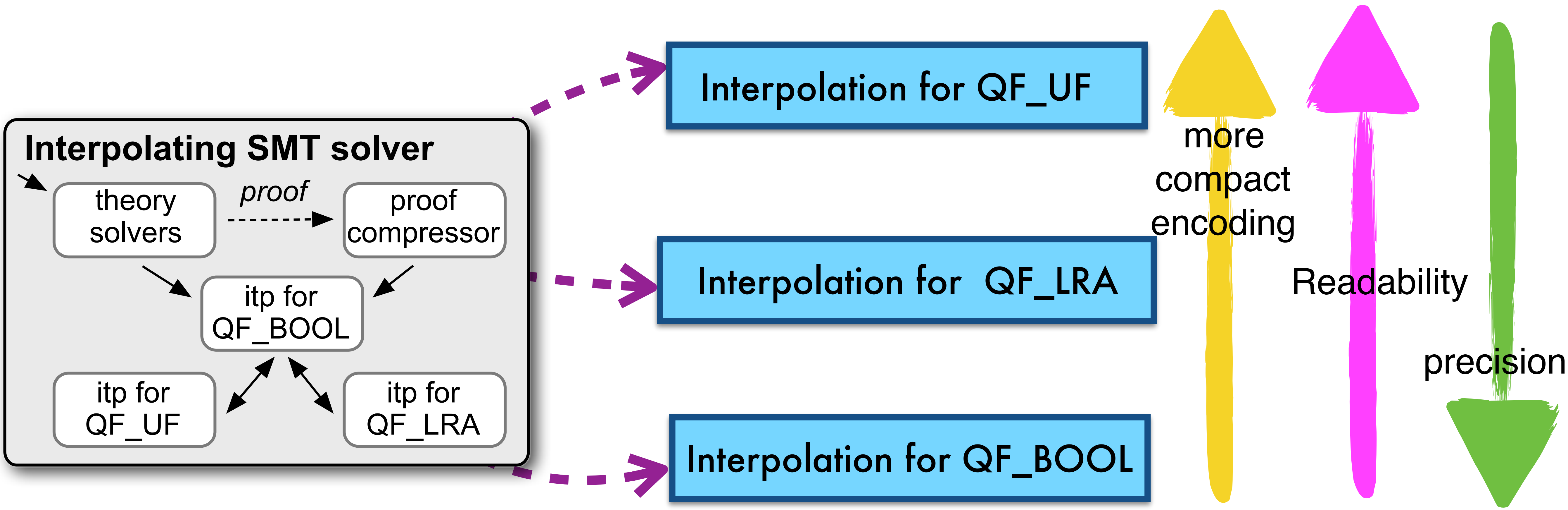
Interpolation for various theories

Each theory has its own interpolation procedure



Interpolation for various theories

Each theory has its own interpolation procedure



Generated interpolants are controllable w.r.t Size and Strength

UF and LRA interpolation system

- Flexibility in generating interpolants → control expressiveness of summaries
- Labeling functions can be partially ordered with respect to strength
- Proof reduction: process the resolution proof to obtain smaller interplants

Different algorithms of Interpolation

QF_UF Alg.

- itps [Alt et al. ’17] [Fuchs et al. ’09]
- itpw [Alt PhD Thesis]
- itpr [Alt et al. '17]

QF_LRA Alg.

- itps [McMillan '05]
- Custom [Alt PhD Thesis '17]

QF_Bool Alg.

-Ms [McMillan ,05]
- P [Pudlák ‘97]
- Mw [D’Silva et al. '10]
- PS [Alt et al. '15]
- PSw [Alt et al. '15]
- PSs [Alt et al. '15]
- Dmin [D’Silva et al. '10]

User-Provided Summaries

User-Provided Summaries

```c
#include <math.h>

double nondet();

double nonlin(double x)
{
    double x_sin = sin(x);
    double x_cos = cos(x);
    return x_sin*x_sin + x_cos*x_cos;
}

void main()
{
    double y = nondet();
    double z = nonlin(y);
    assert(z == 1);
}
```

User-Provided Summaries

Summary of function:

```
(nonlin_return = 1)
```

```c
#include <math.h>

double nondet();

double nonlin(double x)
{
    double x_sin = sin(x);
    double x_cos = cos(x);
    return x_sin*x_sin + x_cos*x_cos;
}

void main()
{
    double y = nondet();
    double z = nonlin(y);
    assert(z == 1);
}
```

USI

User-Provided Summaries

Summary of function:

(nonlin_return = 1)

```c
#include <math.h>

double nondet();

double nonlin(double x)
{
    double x_sin = sin(x);
    double x_cos = cos(x);
    return x_sin*x_sin + x_cos*x_cos;
}

void main()
{
    double y = nondet();
    double z = nonlin(y);
    assert(z == 1);
}
```

```
(define-fun |c::nonlin#0| (
(|c::nonlin::x!0| Real)
(|hifrog::?fun_start| Bool)
(|hifrog::?fun_end| Bool)
(|c::nonlin::?retval| Real) ) Bool
(let ((?def0 true))

?def0
))
```

User-Provided Summaries

Summary of function:

```
(nonlin_return = 1)
```

```
(define-fun |c::nonlin#0| (
  (|c::nonlin::x!0| Real)
  (|hifrog::?fun_start| Bool)
  (|hifrog::?fun_end| Bool)
  (|c::nonlin::?retval| Real) )  Bool
  (= 1 |c::nonlin::?retval|)
)
```

```
#include <math.h>

double nondet();

double nonlin(double x)
{
    double x_sin = sin(x);
    double x_cos = cos(x);
    return x_sin*x_sin + x_cos*x_cos;
}

void main()
{
    double y = nondet();
    double z = nonlin(y);
    assert(z == 1);
}
```

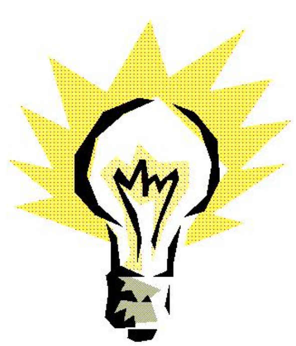
```
(define-fun |c::nonlin#0| (
(|c::nonlin::x!0| Real)
(|hifrog::?fun_start| Bool)
(|hifrog::?fun_end| Bool)
(|c::nonlin::?retval| Real) ) Bool
(let ((?def0 true))

?def0
))
```

User-Provided Summaries

Summary of function:

(nonlin_return = 1)

```
(define-fun |c::nonlin#0| (
  (|c::nonlin::x!0| Real)
  (|hifrog::?fun_start| Bool)
  (|hifrog::?fun_end| Bool)
  (|c::nonlin::?retval| Real) )  Bool
  (= 1 |c::nonlin::?retval|)
)
```



➢ We can inject any summary we want for functions !

```c
#include <math.h>

double nondet();

double nonlin(double x)
{
    double x_sin = sin(x);
    double x_cos = cos(x);
    return x_sin*x_sin + x_cos*x_cos;
}

void main()
{
    double y = nondet();
    double z = nonlin(y);
    assert(z == 1);
}
```

```
(define-fun |c::nonlin#0| (
(|c::nonlin::x!0| Real)
(|hifrog::?fun_start| Bool)
(|hifrog::?fun_end| Bool)
(|c::nonlin::?retval| Real) ) Bool
(let ((?def0 true))

?def0
))
```

a pre-compiled Linux-binary available at the Virtual Machine at http://verify.inf.usi.ch/hifrog/binary

Demo

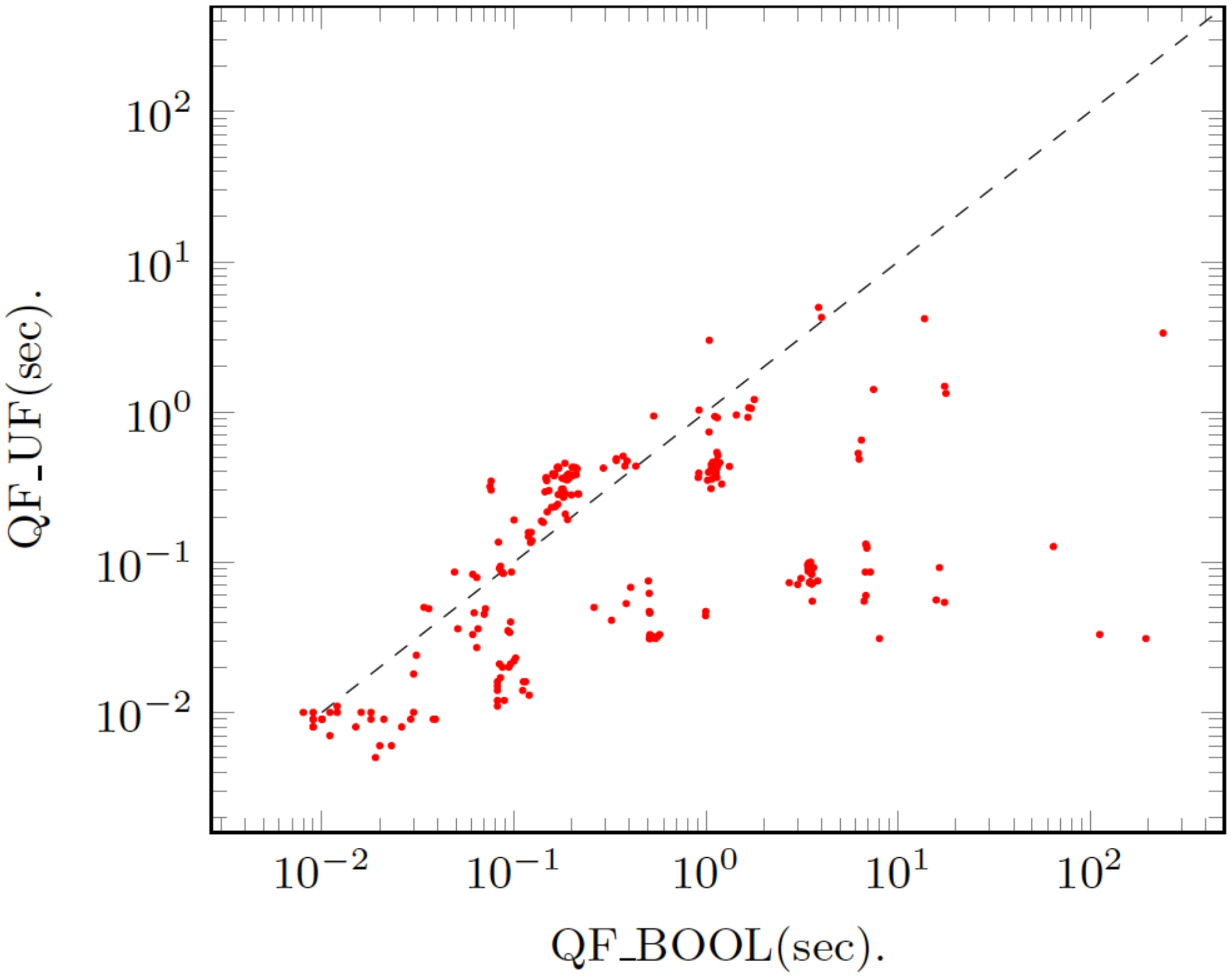
a pre-compiled Linux-binary available at the Virtual Machine at http://verify.inf.usi.ch/hifrog/binary

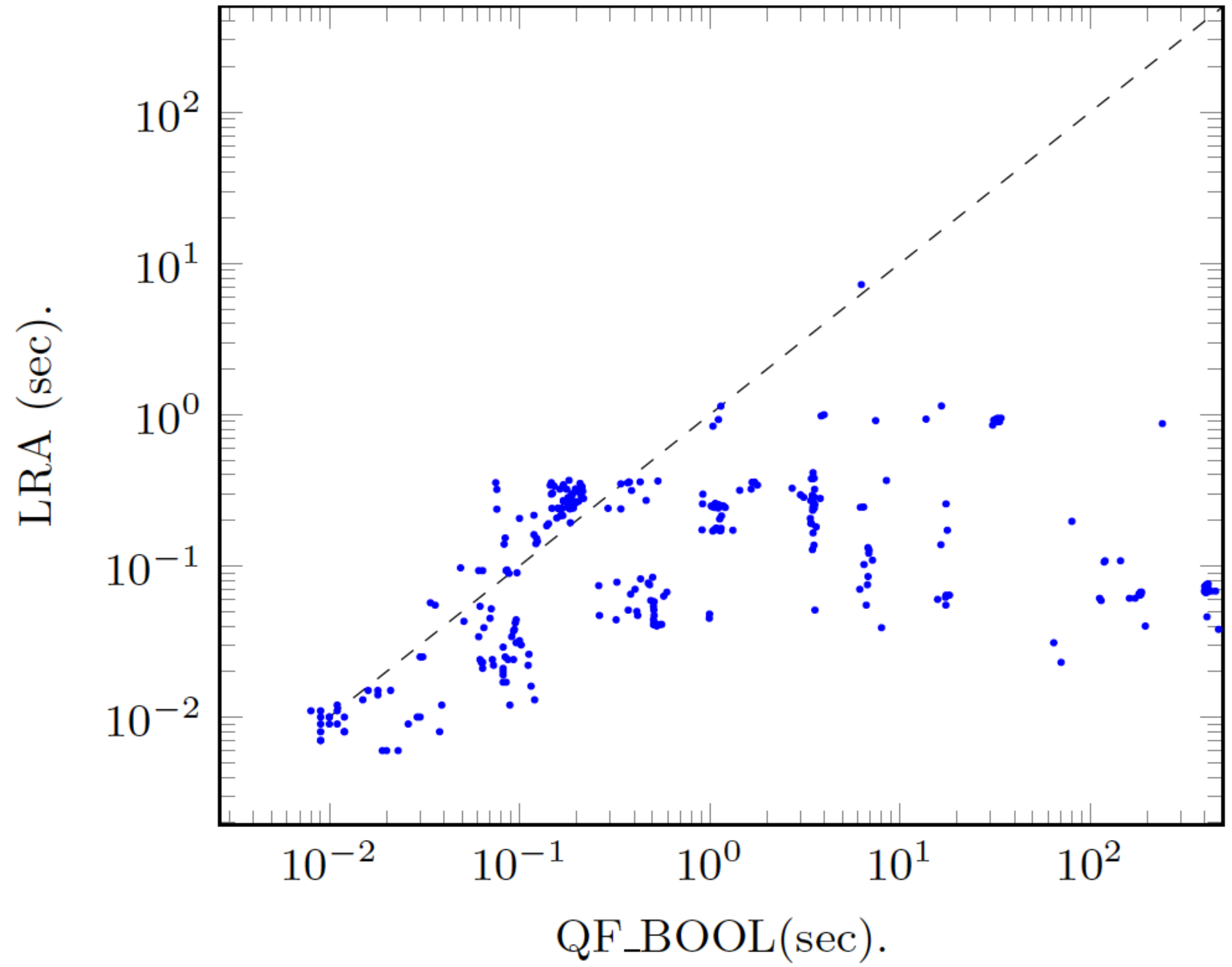
HiFrog evaluation

C Benchmarks	#assertion	QF_UF	QF_LRA	QF_Bool
token.c	54	34	34	34
s3.c	131	18	21	26
mem.c	149	96	96	96
disk.c	79	6	6	23
ddv.c	152	47	47	142
café.c	115	15	20	30
tcas_asrt.c	162	16	29	29
p2p.c	244	8	20	94
floppy1.c	18	15	16	18
floppy2.c	21	15	16	21
floppy4.c	22	11	13	22
floppy3.c	19	13	14	19
diskperf1.c	14	9	10	14
diskperf2.c	4	2	2	4
kbfilter1.c	10	10	10	10
kbfilter2.c	13	13	13	13
kbfilter3.c	14	11	11	14
Percentage of success		**50.65%**	**58%**	**100%**

USI

Experimental Results





Running time by QF_BOOL against QF_UF and QF_LRA.

Recent Related Work

- **FunFrog:** old generation of HiFrog [Sery, Fedyukovich, Sharygina: ATVA'12]
- **eVolCheck**: Incremental upgrade checker for C [Fedyukovich et. al 2013]
- **CBMC** [Kroening et. al 2004]
  - A BMC for C with incremental capabilities of a SAT solver (limited)
- **ESBMC** [Cordeiro 2016]
  - SMT-based tool based on CProver infrastructure, no incrementality
- **Viper** [Muller et al. 2016]
  - A deductive verification tool based on modular verification
- **Dafny** [Leino et al. 2015]
  - A deductive verification tool cashing the intermediate verification results

USI

Future and On-going Work

- Automatic theory Refinement
- Support for other SMT-theories: LIA, Bit-Vector,…
- Parallel verification of several assertions
- Extend to loop summaries (invariants)

Conclusion

- HiFrog → function-summarization-based BMC
- Supports SMT as the modelling and summarization language
  - QF_UF, QF_LRA, QF_LIA and propositional logic

Other features of HiFrog

- User-Provided Summaries
- Removal of redundant assertions
- Counter-example guided summary and theory refinement
- Generating multitude of different interpolants and giving more control to the model checker over them w.r.t Size and Strength

Conclusion

Questions?

- HiFrog → function-summarization-based BMC
- Supports SMT as the modelling and summarization language
  - QF_UF, QF_LRA, QF_LIA and propositional logic

Other features of HiFrog

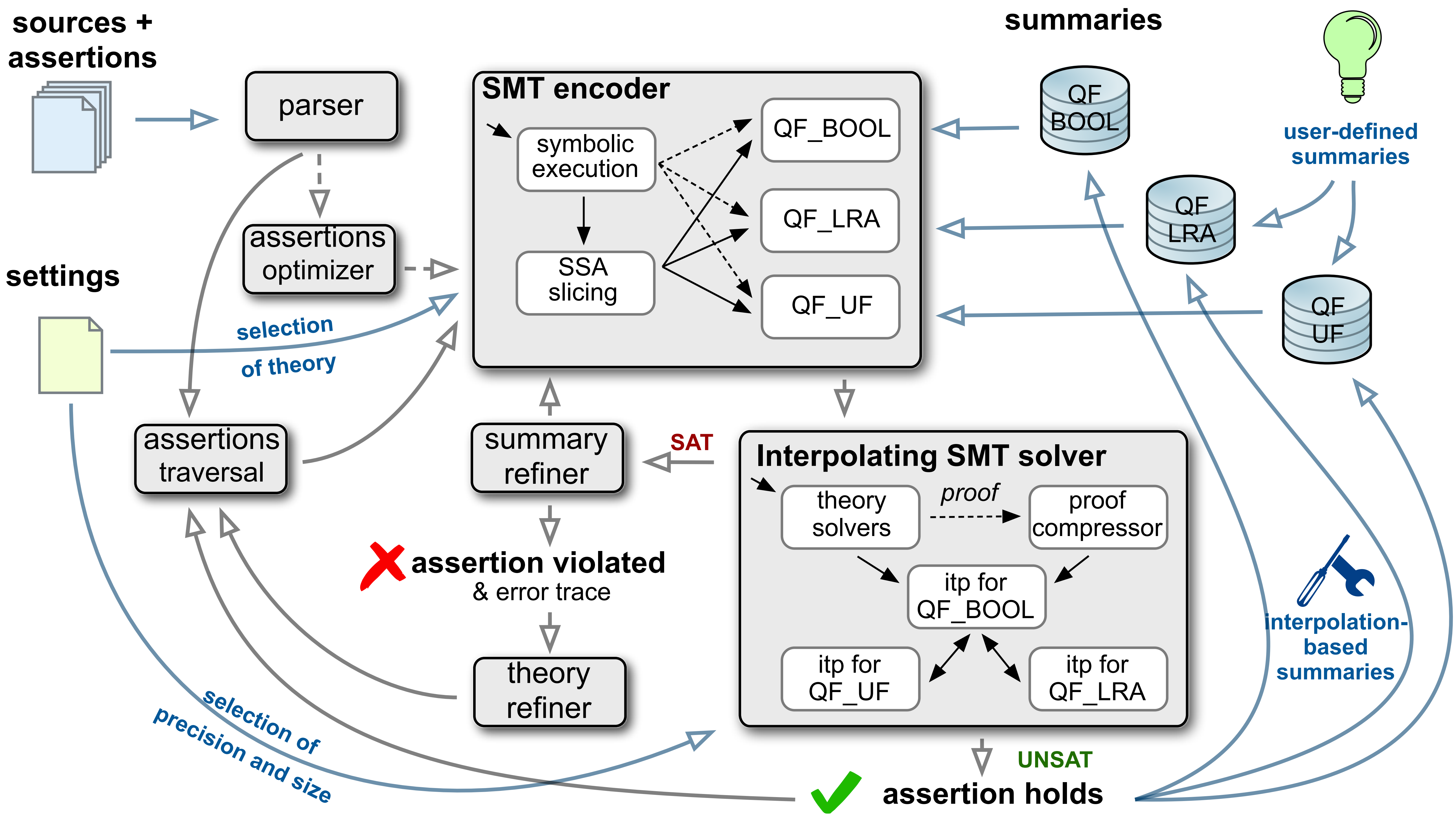
- User-Provided Summaries
- Removal of redundant assertions
- Counter-example guided summary and theory refinement
- Generating multitude of different interpolants and giving more control to the model checker over them w.r.t Size and Strength

Thank you!

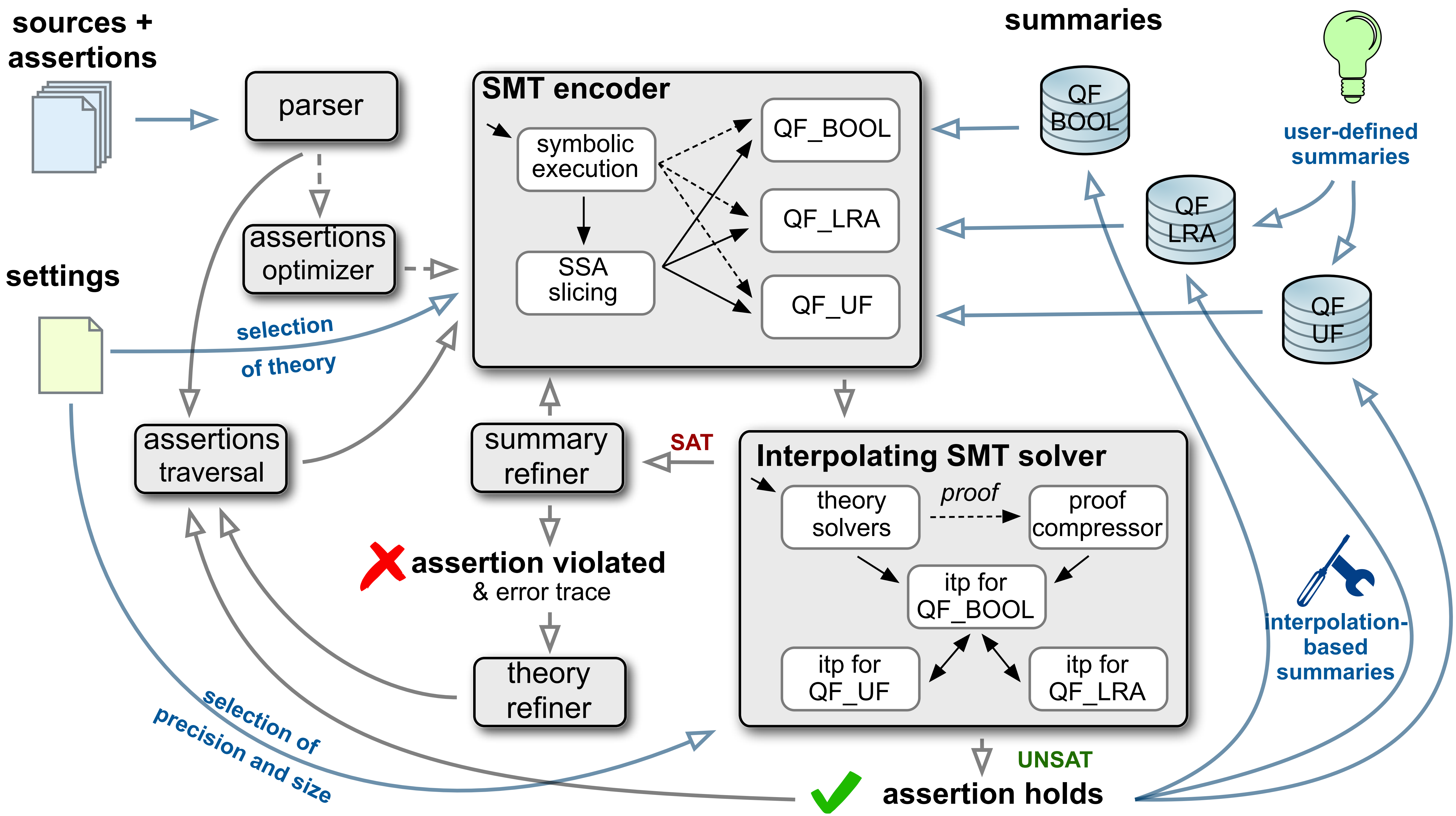
P.S. We are seeking motivated PhD students

www.verify.inf.usi.ch

Contact: natasha.sharygina@usi.ch



a pre-compiled Linux-binary available at the Virtual Machine at http://verify.inf.usi.ch/hifrog/binary



Questions?

a pre-compiled Linux-binary available at the Virtual Machine at http://verify.inf.usi.ch/hifrog/binary