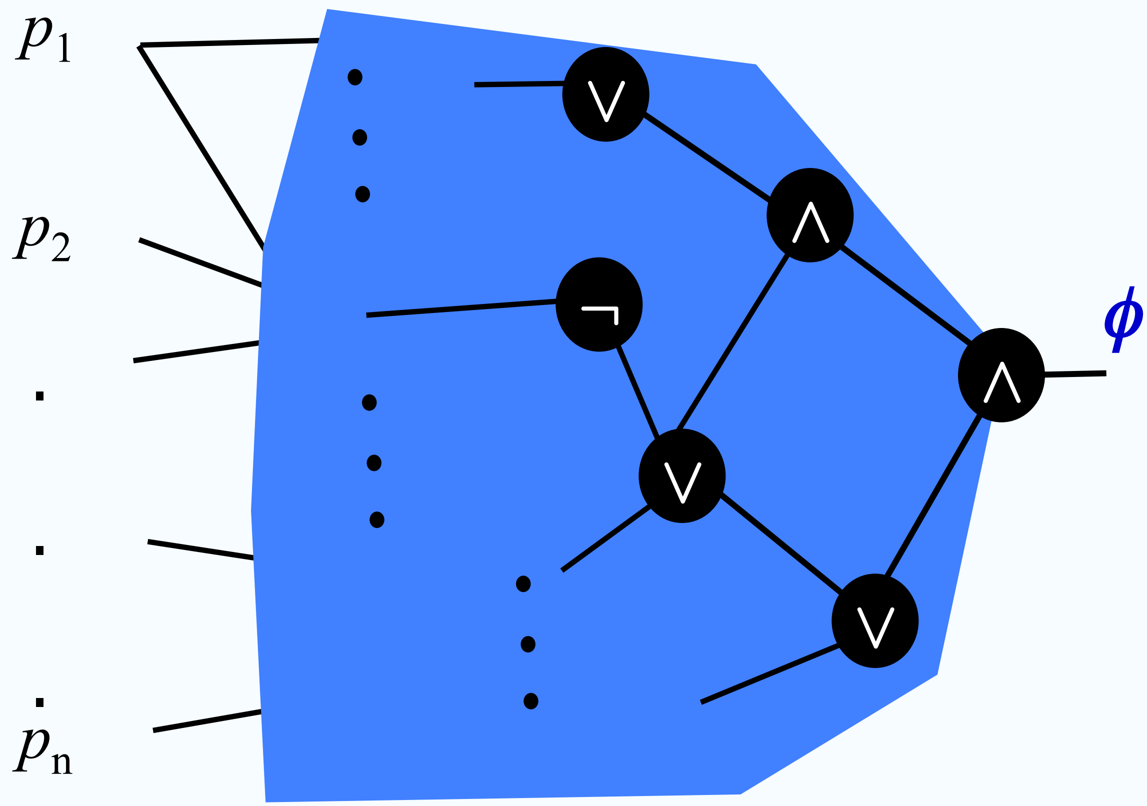
SMT Solvers

An overview from the perspective of Symbolic Excution

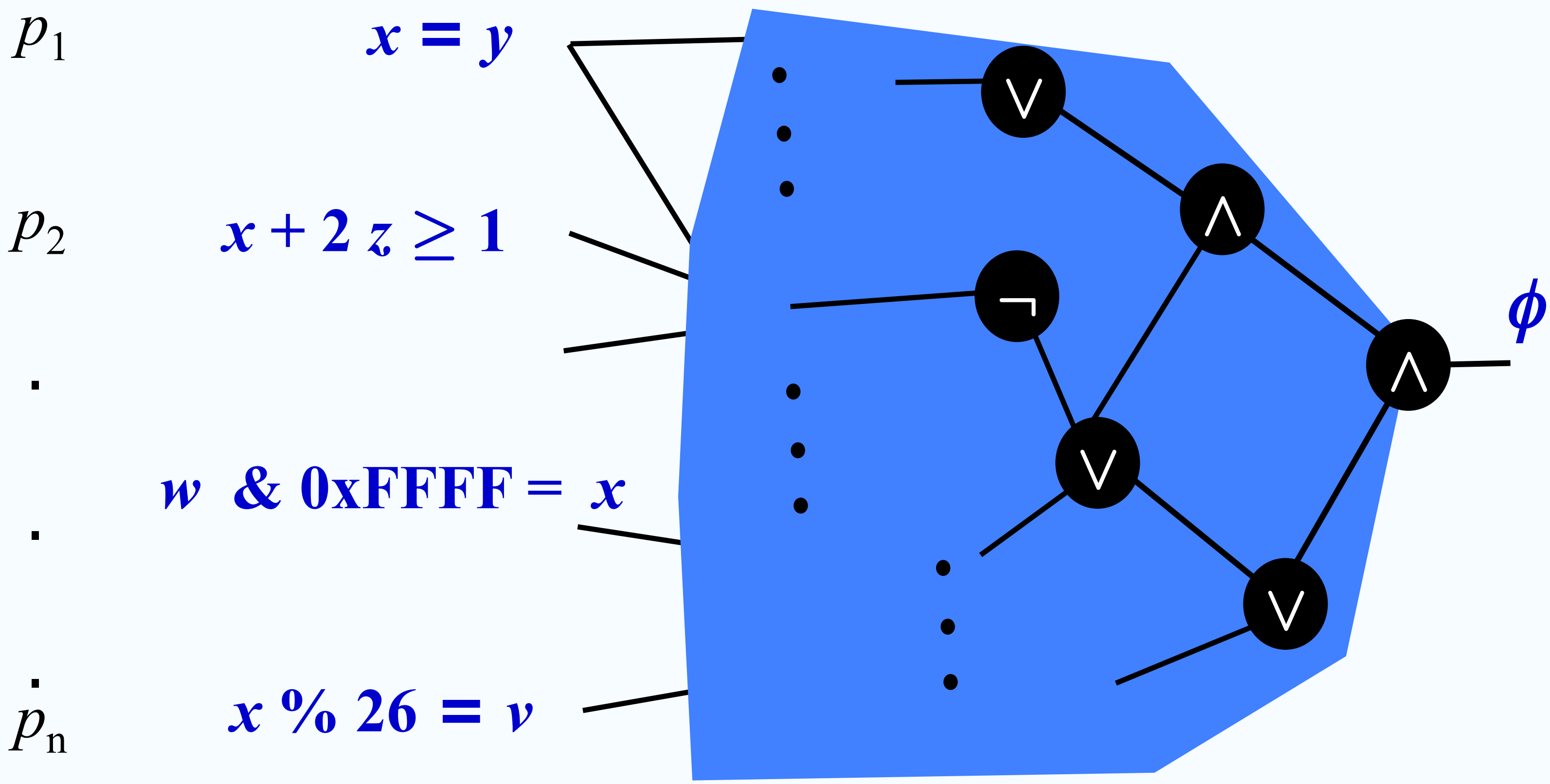
Based in part on Barrett & Seshia's ICCAD'09 Tutorial

Boolean Satisfiability (SAT)



Is there an assignment to the $p_1, p_2, \ldots, p_n$ variables such that ϕ evaluates to 1?

Satisfiability Modulo Theories



Is there an assignment to the *x,y,z,w* variables s.t. ϕ evaluates to 1?

Satisfiability Modulo Theories

- Given a formula in first-order logic, with associated background theories, is the formula satisfiable?
  - Yes: return a satisfying solution
  - No [generate a proof of unsatisfiability]

Applications of SMT

- Hardware verification at higher levels of abstraction (RTL and above)
- Verification of analog/mixed-signal circuits
- Verification of hybrid systems
- Software model checking
- Software testing
- Security: Finding vulnerabilities, verifying electronic voting machines, …
- Program synthesis
- …

References

Satisfiability Modulo Theories

Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli.

Chapter 8 in the Handbook of Satisfiability, Armin Biere, Hans van Maaren, and Toby Walsh, editors, IOS Press, 2009.

SMTLIB: A repository for SMT formulas (common format) and tools

SMTCOMP: An annual competition of SMT solvers

Roadmap

- Background and Notation
- Survey of Theories
- Theory Solvers
- A Parameterized Solver Framework

First-Order Logic

- A formal notation for mathematics, with expressions involving
  - Propositional symbols
  - Predicates
  - Functions and constant symbols
  - Quantifiers
- In contrast, propositional (Boolean) logic only involves propositional symbols and operators

First-Order Logic: Syntax

- As with propositional logic, expressions in first-order logic are made up of sequences of symbols.
- Symbols are divided into *logical symbols* and *non-logical symbols* or *parameters*.
- Example:

$$(x = y) \wedge (y = z) \wedge (f(z) \geq f(x)+1)$$

First-Order Logic: Syntax

- Logical Symbols
  - Propositional connectives: $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$
  - Variables: v1, v2, . . .
  - Quantifiers: $\forall$, $\exists$
- Non-logical symbols/Parameters
  - Equality: =
  - Functions: +, -, %, bit-wise &, f(), concat, …
  - Predicates: $\leq$, is_substring, …
  - Constant symbols: 0, 1.0, null, …

Quantifier-free Subset

- We will largely restrict ourselves to formulas without quantifiers (∀, ∃)
- This is called the quantifier-free subset/ fragment of first-order logic with the relevant theory

Logical Theory

- Defines a set of parameters (non-logical symbols) and their meanings
- This definition is called a *signature*.
- Example of a signature:

  Theory of linear arithmetic over integers

  Signature is $(0,1,+,-,\leq)$ interpreted over $\mathbb{Z}$

Roadmap

- Background and Notation
- Survey of Theories
- Theory Solvers
- A Parameterized Solver Framework

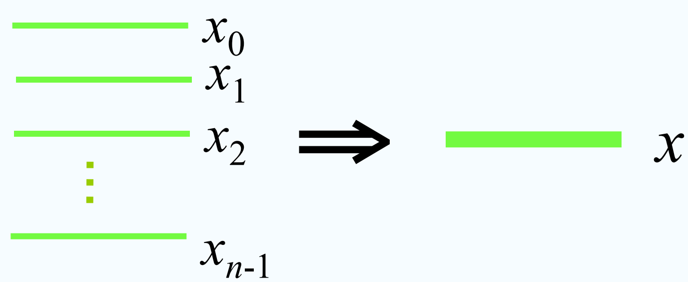
Some Useful Theories

- Equality (with uninterpreted functions)
- Linear arithmetic (over $\mathbb{Q}$ or $\mathbb{Z}$)
- Difference logic (over $\mathbb{Q}$ or $\mathbb{Z}$)
- Finite-precision bit-vectors
  - integer or floating-point
- Arrays / memories
- Misc.: Non-linear arithmetic, strings, inductive datatypes (e.g. lists), sets, …

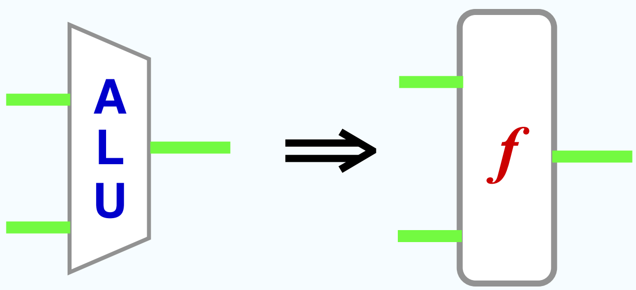
Theory of Equality and Uninterpreted Functions (EUF)

- Also called the "free theory"
  - Because function symbols can take any meaning
  - Only property required is *congruence*: that these symbols map identical arguments to identical values i.e., $x = y \Rightarrow f(x) = f(y)$
- SMTLIB name: QF_UF

Data and Function Abstraction with EUF



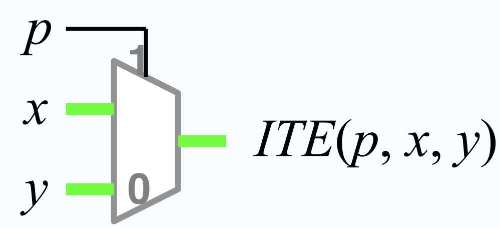
Bit-vectors to Abstract Domain (e.g. $\mathbb{Z}$)



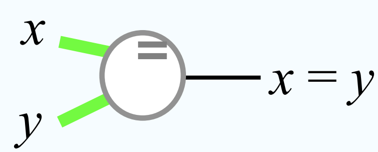
Functional units to Uninterpreted Functions

$$a = x \;\wedge\; b = y \;\Rightarrow\; f(a,b) = f(x,y)$$

Common Operations

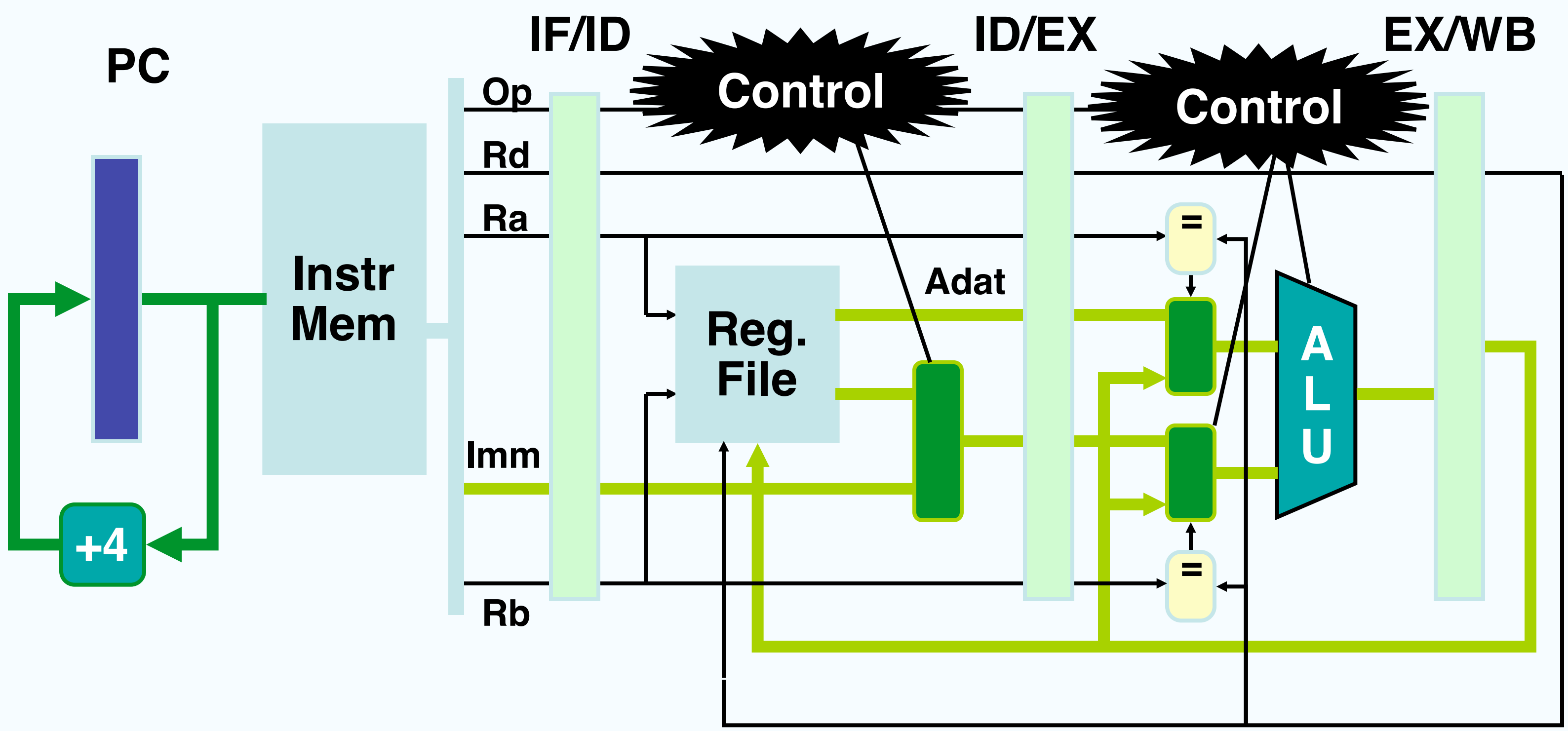


If-then-else



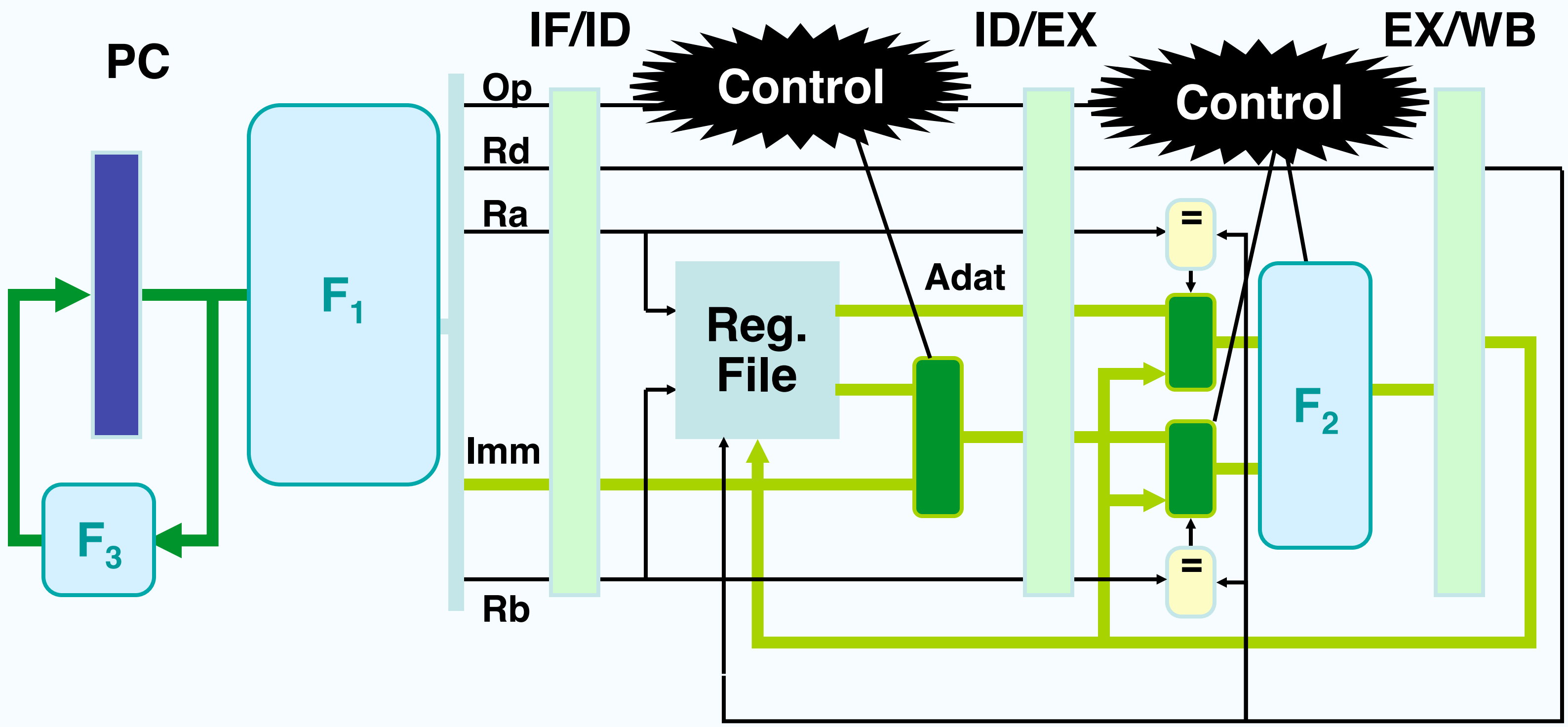
Test for equality

Hardware Abstraction with EUF



- For any Block that Transforms or Evaluates Data:
  - Replace with generic, unspecified function
  - Also view instruction memory as function

Hardware Abstraction with EUF



- For any Block that Transforms or Evaluates Data:
  - Replace with generic, unspecified function
  - Also view instruction memory as function

Example QF_UF (EUF) Formula

$$(x = y) \wedge (y = z) \wedge (f(x) \neq f(z))$$

Transitivity:

$$(x = y) \wedge (y = z) \Rightarrow (x = z)$$

Congruence:

$$(x = z) \Rightarrow (f(x) = f(z))$$

Equivalence Checking of Program Fragments

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return x*x;
}

int fun2(int y) {
     return y*y;
}
```

SMT formula ϕ
Satisfiable iff programs non-equivalent

$$(z = y \land y1 = x \land x1 = z \land ret1 = x1*x1)$$
$$\land$$
$$(ret2 = y*y)$$
$$\land$$
$$(ret1 \neq ret2)$$

What if we use SAT to check equivalence?

Equivalence Checking of Program Fragments

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return x*x;
}

int fun2(int y) {
     return y*y;
}
```

SMT formula ϕ
Satisfiable iff programs non-equivalent

$$(z = y \wedge y1 = x \wedge x1 = z \wedge ret1 = x1*x1)$$
$$\wedge$$
$$(ret2 = y*y)$$
$$\wedge$$
$$(ret1 \neq ret2)$$

Using SAT to check equivalence (w/ Minisat)
- 32 bits for y: Did not finish in over 5 hours
- 16 bits for y: 37 sec.
- 8 bits for y: 0.5 sec.

Equivalence Checking of Program Fragments

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return x*x;
}

int fun2(int y) {
     return y*y;
}
```

SMT formula ϕ'

$$(z = y \wedge y1 = x \wedge x1 = z \wedge ret1 = sq(x1))$$
$$\wedge$$
$$(ret2 = sq(y))$$
$$\wedge$$
$$(ret1 \neq ret2)$$

Using EUF solver: 0.01 sec

Linear Arithmetic (QF_LRA, QF_LIA)

- Boolean combination of linear constraints of the form

  $$(a_1\, x_1 + a_2\, x_2 + \ldots + a_n\, x_n \sim b)$$

- x_i's could be in $\mathbb{Q}$ or $\mathbb{Z}$, $\sim\ \in\ \{\geq, >, \leq, <, =\}$

- Many applications, including:
  - Verification of analog circuits
  - Software verification, e.g., of array bounds

Difference Logic (QF_IDL, QF_RDL)

- Boolean combination of linear constraints of the form

  $x_i - x_j \sim c_{ij}$ or $x_i \sim c_i$

  $\sim \in \{\geq, >, \leq, <, =\}$, x_i's in $\mathbb{Q}$ or $\mathbb{Z}$

- Applications:
  - Software verification (most linear constraints are of this form)
  - Processor datapath verification
  - Job shop scheduling / real-time systems
  - Timing verification for circuits

Arrays/Memories

- SMT solvers can also be very effective in modeling data structures in software and hardware
  - Arrays in programs
  - Memories in hardware designs: e.g. instruction and data memories, CAMs, etc.

Theory of Arrays (QF_AX)
Select and Store

- Two interpreted functions: select and store
  - select(A,i)   Read from A at index i
  - store(A,i,d)   Write d to A at index i
- Two main axioms:
  - select(store(A,i,d), i) = d
  - select(store(A,i,d), j) = select(A,j) for $i \neq j$
- One other axiom:
  - $(\forall$ i. select(A,i) = select(B,i)) $\Rightarrow$ A = B

Equivalence Checking of Program Fragments

```
int fun1(int y) {
    int x[2];
    x[0] = y;
    y = x[1];
    x[1] = x[0];

   return x[1]*x[1];
}

 int fun2(int y) {
      return y*y;
 }
```

SMT formula ϕ''

$[\quad x1 = store(x,0,y) \wedge y1 = select(x1,1)$
$\wedge\; x2 = store(x1,1,select(x1,0))$
$\wedge\; ret1 = sq(select(x2,1)) \quad]$
$\wedge$
$(\; ret2 = sq(y)\;)$
$\wedge$
$(\; ret1 \neq ret2\;)$

Roadmap

- Background and Notation
- Survey of Theories
- Theory Solvers
- A Parameterized Solver Framework

Difference Logic

In *difference logic* [NO05], we are interested in the satisfiability of a conjunction of arithmetic atoms.

Each atom is of the form $x - y \bowtie c$, where x and y are variables, c is a numeric constant, and $\bowtie \in \{=, <, \leq, >, \geq\}$.

The variables can range over either the *integers* (QF_IDL) or the *reals* (QF_RDL).

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \quad \Longrightarrow \quad x - y \leq c \wedge x - y \geq c$

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \quad \Longrightarrow \quad x - y \leq c \wedge x - y \geq c$
- $x - y \geq c \quad \Longrightarrow \quad y - x \leq -c$

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \quad \Longrightarrow \quad x - y \leq c \land x - y \geq c$
- $x - y \geq c \quad \Longrightarrow \quad y - x \leq -c$
- $x - y > c \quad \Longrightarrow \quad y - x < -c$

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \quad \Longrightarrow \quad x - y \leq c \wedge x - y \geq c$
- $x - y \geq c \quad \Longrightarrow \quad y - x \leq -c$
- $x - y > c \quad \Longrightarrow \quad y - x < -c$
- $x - y < c \quad \Longrightarrow \quad x - y \leq c - 1$ (integers)

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \quad \Longrightarrow \quad x - y \leq c \wedge x - y \geq c$
- $x - y \geq c \quad \Longrightarrow \quad y - x \leq -c$
- $x - y > c \quad \Longrightarrow \quad y - x < -c$
- $x - y < c \quad \Longrightarrow \quad x - y \leq c - 1$ (integers)
- $x - y < c \quad \Longrightarrow \quad x - y \leq c - \delta$ (reals)

Difference Logic

Now we have a conjunction of literals, all of the form $x - y \leq c$.

From these literals, we form a weighted directed graph with a vertex for each variable.

For each literal $x - y \leq c$, there is an edge $x \xrightarrow{c} y$.

The set of literals is satisfiable iff there is no cycle for which the sum of the weights on the edges is negative.

There are a number of efficient algorithms for detecting negative cycles in graphs [CG96].

Example: QF_IDL

$$x - y = 5 \ \wedge \ z - y \geq 2 \ \wedge \ z - x > 2 \ \wedge \ w - x = 2 \ \wedge \ z - w < 0$$

Example: QF_IDL

$$x - y = 5 \ \wedge \ z - y \geq 2 \ \wedge \ z - x > 2 \ \wedge \ w - x = 2 \ \wedge \ z - w < 0$$

$$x - y = 5$$
$$z - y \geq 2$$
$$z - x > 2 \quad \Rightarrow$$
$$w - x = 2 \qquad w - x \leq 2 \wedge x - w \leq -2$$
$$z - w < 0$$

Example: QF_IDL

$$x - y = 5 \ \wedge \ z - y \geq 2 \ \wedge \ z - x > 2 \ \wedge \ w - x = 2 \ \wedge \ z - w < 0$$

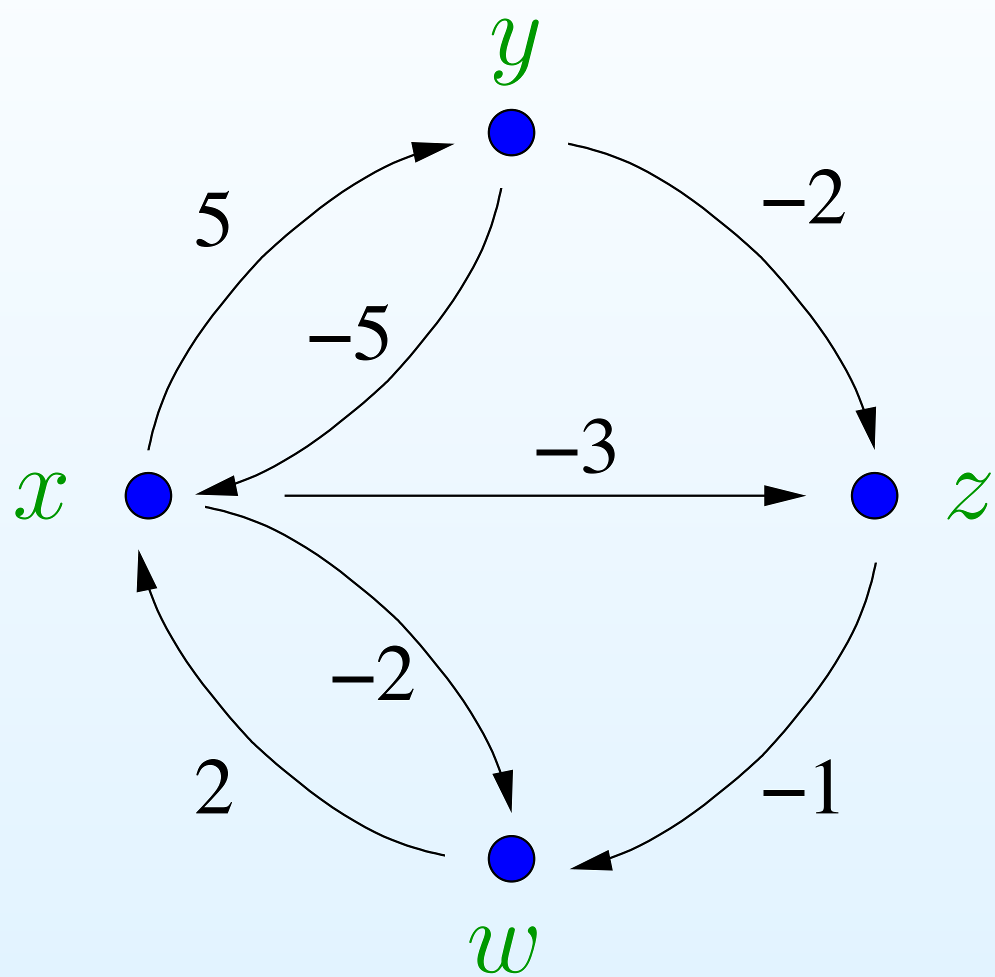
$$\begin{aligned} & x - y = 5 \\ & z - y \geq 2 \\ & z - x > 2 \quad \Rightarrow \\ & w - x = 2 \qquad\quad w - x \leq 2 \wedge x - w \leq -2 \\ & z - w < 0 \end{aligned}$$

Example: QF_IDL

$$x - y = 5 \;\wedge\; z - y \geq 2 \;\wedge\; z - x > 2 \;\wedge\; w - x = 2 \;\wedge\; z - w < 0$$

$$\begin{array}{lcl} x - y = 5 & & x - y \leq 5 \wedge y - x \leq -5 \\ z - y \geq 2 & & y - z \leq -2 \\ z - x > 2 & \Rightarrow & x - z \leq -3 \\ w - x = 2 & & w - x \leq 2 \wedge x - w \leq -2 \\ z - w < 0 & & z - w \leq -1 \end{array}$$

Example: QF_IDL



Roadmap

Theory Solvers

- Examples of Theory Solvers
- Combining Theory Solvers
- Extending Theory Solvers for SMT

Combining Theory Solvers

Theory solvers become much more useful if they can be used together.

$$mux\_sel = 0 \rightarrow mux\_out = select(regfile, addr)$$
$$mux\_sel = 1 \rightarrow mux\_out = ALU(alu0, alu1)$$

For such formulas, we are interested in satisfiability with respect to a *combination* of theories.

Fortunately, there exist methods for combining theory solvers. The standard technique for this is the Nelson-Oppen method [NO79, TH96].

The Nelson-Oppen Method

The Nelson-Oppen method is applicable when:

1. The theories have *no shared symbols* (other than equality).
2. The theories are *stably-infinite*.

   A theory T is *stably-infinite* if every T-satisfiable quantifier-free formula is satisfiable in an infinite model.

3. The formulas to be tested for satisfiability are *quantifier-free*

Many theories fit these criteria, and extensions exist in some cases when they do not.

The Nelson-Oppen Method

Suppose that T_1 and T_2 are theories and that *Sat*$_1$ is a theory solver for T_1-satisfiability and *Sat*$_2$ for T_1-satisfiability.

We wish to determine if ϕ is $T_1 \cup T_2$-satisfiable.

1. Convert ϕ to its *separate form* $\phi_1 \wedge \phi_2$.
2. Let S be the set of variables shared between ϕ_1 and ϕ_2.
3. For each *arrangement* Δ of S:
   (a) Run *Sat*$_1$ on $\phi_1 \cup \Delta$.
   (b) Run *Sat*$_2$ on $\phi_2 \cup \Delta$.

The Nelson-Oppen Method

If there exists an arrangement such that both Sat_1 and Sat_2 succeed, then ϕ is $T_1 \cup T_2$-satisfiable.

If no such arrangement exists, then ϕ is $T_1 \cup T_2$-unsatisfiable.

Example

Consider the following QF_UFLIA formula:

$\phi = 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2)$.

Example

Consider the following QF_UFLIA formula:

$$\phi = 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2).$$

We first convert ϕ to a separate form:

$$\phi_{UF} = f(x) \neq f(y) \ \wedge \ f(x) \neq f(z)$$
$$\phi_{LIA} = 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ y = 1 \ \wedge \ z = 2$$

The shared variables are $\{x, y, z\}$. There are 5 possible arrangements based on equivalence classes of x, y, and z.

Example

$$\phi_{UF} = f(x) \neq f(y) \;\; \wedge \;\; f(x) \neq f(z)$$
$$\phi_{LIA} = 1 \leq x \;\; \wedge \;\; x \leq 2 \;\; \wedge \;\; y = 1 \;\; \wedge \;\; z = 2$$

1. $\{x = y, x = z, y = z\}$
2. $\{x = y, x \neq z, y \neq z\}$
3. $\{x \neq y, x = z, y \neq z\}$
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

Example

$$\phi_{UF} = f(x) \neq f(y) \;\; \wedge \;\; f(x) \neq f(z)$$
$$\phi_{LIA} = 1 \leq x \;\; \wedge \;\; x \leq 2 \;\; \wedge \;\; y = 1 \;\; \wedge \;\; z = 2$$

1. $\{x = y, x = z, y = z\}$: inconsistent with ϕ_{UF}.
2. $\{x = y, x \neq z, y \neq z\}$
3. $\{x \neq y, x = z, y \neq z\}$
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

Example

$$\phi_{UF} = f(x) \neq f(y) \ \wedge \ f(x) \neq f(z)$$
$$\phi_{LIA} = 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ y = 1 \ \wedge \ z = 2$$

1. $\{x = y, x = z, y = z\}$: inconsistent with ϕ_{UF}.
2. $\{x = y, x \neq z, y \neq z\}$: inconsistent with ϕ_{UF}.
3. $\{x \neq y, x = z, y \neq z\}$
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

Example

$$\phi_{UF} = f(x) \neq f(y) \;\; \wedge \;\; f(x) \neq f(z)$$
$$\phi_{LIA} = 1 \leq x \;\; \wedge \;\; x \leq 2 \;\; \wedge \;\; y = 1 \;\; \wedge \;\; z = 2$$

1. $\{x = y, x = z, y = z\}$: inconsistent with ϕ_{UF}.
2. $\{x = y, x \neq z, y \neq z\}$: inconsistent with ϕ_{UF}.
3. $\{x \neq y, x = z, y \neq z\}$: inconsistent with ϕ_{UF}.
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

Example

$$\phi_{UF} = f(x) \neq f(y) \;\; \wedge \;\; f(x) \neq f(z)$$
$$\phi_{LIA} = 1 \leq x \;\; \wedge \;\; x \leq 2 \;\; \wedge \;\; y = 1 \;\; \wedge \;\; z = 2$$

1. $\{x = y, x = z, y = z\}$: inconsistent with ϕ_{UF}.
2. $\{x = y, x \neq z, y \neq z\}$: inconsistent with ϕ_{UF}.
3. $\{x \neq y, x = z, y \neq z\}$: inconsistent with ϕ_{UF}.
4. $\{x \neq y, x \neq z, y = z\}$: inconsistent with ϕ_{LIA}.
5. $\{x \neq y, x \neq z, y \neq z\}$

Example

$$\phi_{UF} = f(x) \neq f(y) \;\; \wedge \;\; f(x) \neq f(z)$$
$$\phi_{LIA} = 1 \leq x \;\; \wedge \;\; x \leq 2 \;\; \wedge \;\; y = 1 \;\; \wedge \;\; z = 2$$

1. $\{x = y, x = z, y = z\}$: inconsistent with ϕ_{UF}.
2. $\{x = y, x \neq z, y \neq z\}$: inconsistent with ϕ_{UF}.
3. $\{x \neq y, x = z, y \neq z\}$: inconsistent with ϕ_{UF}.
4. $\{x \neq y, x \neq z, y = z\}$: inconsistent with ϕ_{LIA}.
5. $\{x \neq y, x \neq z, y \neq z\}$: inconsistent with ϕ_{LIA}.

Example

$$\phi_{UF} = f(x) \neq f(y) \ \wedge \ f(x) \neq f(z)$$
$$\phi_{LIA} = 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ y = 1 \ \wedge \ z = 2$$

1. $\{x = y, x = z, y = z\}$: inconsistent with ϕ_{UF}.
2. $\{x = y, x \neq z, y \neq z\}$: inconsistent with ϕ_{UF}.
3. $\{x \neq y, x = z, y \neq z\}$: inconsistent with ϕ_{UF}.
4. $\{x \neq y, x \neq z, y = z\}$: inconsistent with ϕ_{LIA}.
5. $\{x \neq y, x \neq z, y \neq z\}$: inconsistent with ϕ_{LIA}.

Therefore, ϕ is *unsatisfiable*.

Roadmap

Theory Solvers

- Examples of Theory Solvers
- Combining Theory Solvers
- Extending Theory Solvers for SMT

Desirable Characteristics of Theory Solvers

Theory solvers must be able to determine whether a conjunction of literals is satisfiable.

However, in order to integrate a theory solver into a modern SMT solver, it is helpful if the theory solvers can do more.

Desirable Characteristics of Theory Solvers

Some desirable characterstics of theory solvers include:

- *Incrementality* - easy to add new literals or backtrack to a previous state
- *Layered/Lazy* - able to detect simple inconsistencies quickly, able to detect difficult inconsistencies eventually
- *Equality Propagating* - If theory solvers can detect when two terms are equivalent, this greatly simplifies the search for a satisfying arrangement

Desirable Characteristics of Theory Solvers

Some desirable characterstics of theory solvers include:

- *Model Generating* - When reporting satisfiable, the theory solver also provides a concrete value for each variable or function symbol
- *Proof Generating* - When reporting unsatisfiable, the theory solver also provides a checkable proof
- *Interpolant Generating* - If $\phi \wedge \neg\psi$ is unsatisfiable, find a formula α containing only symbols appearing in both ϕ and ψ such that:
    - $\phi \wedge \neg\alpha$ is unsatisfiable
    - $\alpha \wedge \neg\psi$ is unsatisfiable

Lazy SMT

Theory solvers check the satisfiability of conjunctions of literals.

What about more general Boolean structure?

What is needed is a combination of *Boolean reasoning* and *theory reasoning*.

The *eager* approach to SMT does this by encoding theory reasoning as a Boolean satisfiability problem.

Here, I will focus on the *lazy* approach in which both a Boolean engine and a theory solver work together to solve the problem [dMRS02, BDS02a].

The architecture of Lazy SMT

1. Separate ϕ into ϕ_{T_i} and ϕ_s
2. Abst ula ϕ_a
3. Chec
4. If UN
5. If SA solver
6. If all
7. Try another SAT assignment for $[\phi_s]_a$ and go to 5, if there are none, then done (ϕ UNSAT)

Caveat: This is a very high level sketch that abstracts many details. SMT papers will not explain it in this way.

The architecture of Lazy SMT

1. Separate ϕ into ϕ_{T_i} and ϕ_s
2. Abstract the result to a propositional formula ϕ_a
3. Check ϕ_a for SAT
4. If UNSAT, then done (ϕ UNSAT)
5. If SAT, then check $\phi_{T_i} \wedge [\phi_s]_a$ with theory solver
6. If all ϕ_{T_i} are SAT, then done (ϕ SAT)
7. Try another SAT assignment for $[\phi_s]_a$ and go to 5, if there are none, then done (ϕ UNSAT)

Separating a formula

Recall the formula

$$1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We separate it as follows

$$\phi_{UF} = f(x) \neq f(y) \wedge f(x) \neq f(z)$$

$$\phi_{LIA} = 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$$

$$\phi_s = x = y \wedge x = z \wedge y = z$$

So the original formula is

$$\phi_{UF} \wedge \phi_{LIA} \wedge \phi_S$$

Abstracting a formula

Take each unique conjunct and represent it as a propositional variable

So

$$\phi_{UF} = f(x) \neq f(y) \wedge f(x) \neq f(z)$$

becomes

$$\phi_{UF} = a \wedge b$$

where, for example,

$$a = f(x) \neq f(y)$$

Arrangements

When the abstracted formula is SAT we have an assignment to the propositional variables

The abstracted version of

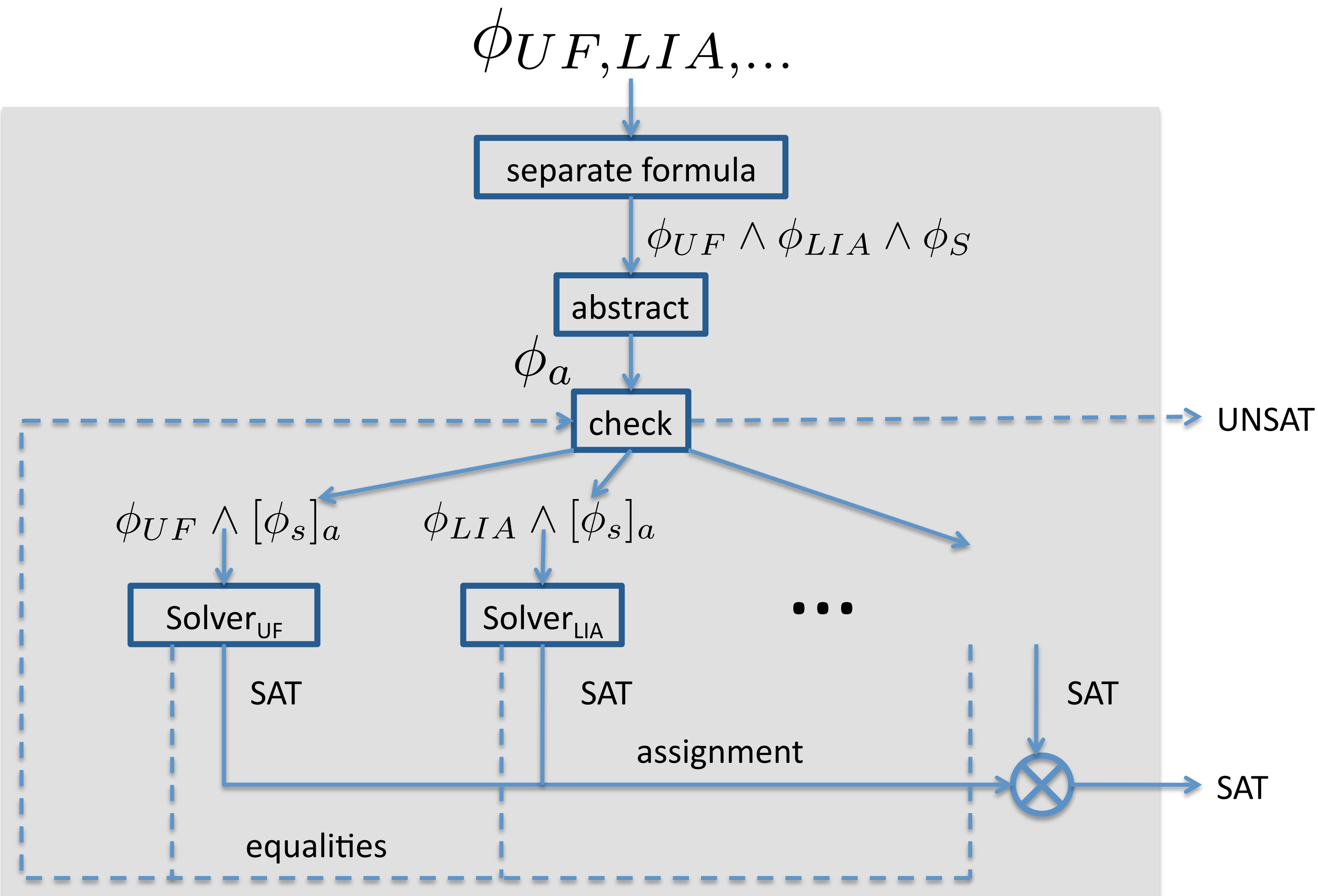
$$\phi_s = x = y \wedge x = z \wedge y = z$$

becomes

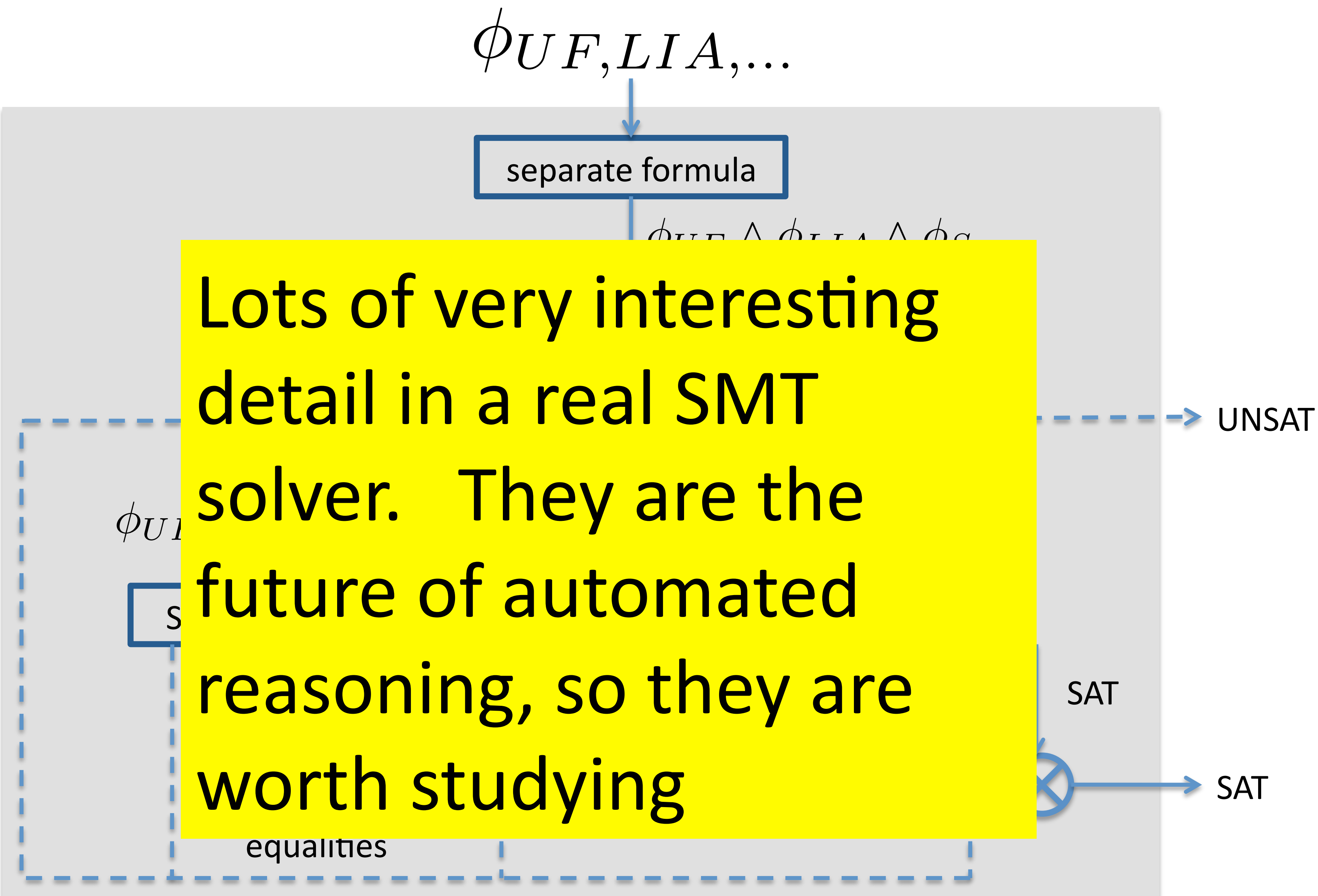
$$[\phi_s]_a = g \wedge h \wedge i$$

and in a SAT assignment we may have $g, \neg h, \neg i$

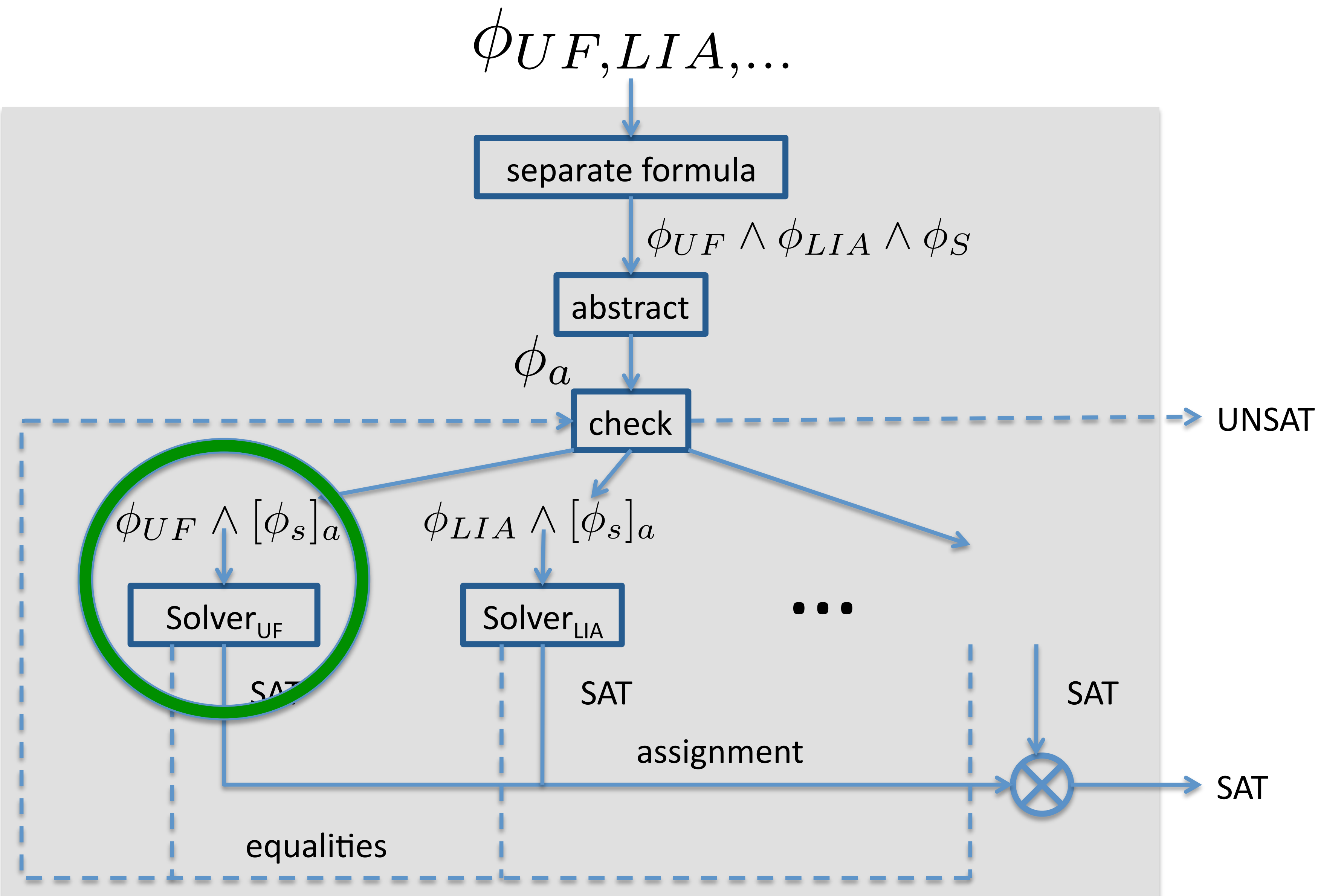
$$x = y \wedge x \neq z \wedge y \neq z$$



$\phi_{UF,LIA,...}$
separate formula
$\phi_{UF} \wedge \phi_{LIA} \wedge \phi_S$
abstract
ϕ_a
check
UNSAT
$\phi_{UF} \wedge [\phi_s]_a$
$\phi_{LIA} \wedge [\phi_s]_a$
$Solver_{UF}$
$Solver_{LIA}$
...
SAT
SAT
SAT
assignment
SAT
equalities



Lots of very interesting detail in a real SMT solver. They are the future of automated reasoning, so they are worth studying



$\phi_{UF,LIA,\ldots}$
separate formula
$\phi_{UF} \wedge \phi_{LIA} \wedge \phi_S$
abstract
ϕ_a
check
UNSAT
$\phi_{UF} \wedge [\phi_s]_a$
$\phi_{LIA} \wedge [\phi_s]_a$
Solver$_{UF}$
Solver$_{LIA}$
SAT
SAT
SAT
assignment
SAT
equalities

For symbolic execution …

- What do the path conditions look like?
- What different theories are involved?
- What is the most restrictive theory possible?
- Do path conditions vary with the program?
- Can we determine, for a program what theories are needed?
- Can we extend an SMT solver with a solver for those theories?