UPPSALA UNIVERSITET

Flatten and Conquer

A Framework for Efficient Analysis of String Constraints

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Yu-Fang Chen[2], **Bui Phi Diep**[1], Lukáš Holík[3], Ahmed Rezine[4], Philipp Rümmer[1]

[1] Uppsala University, Sweden

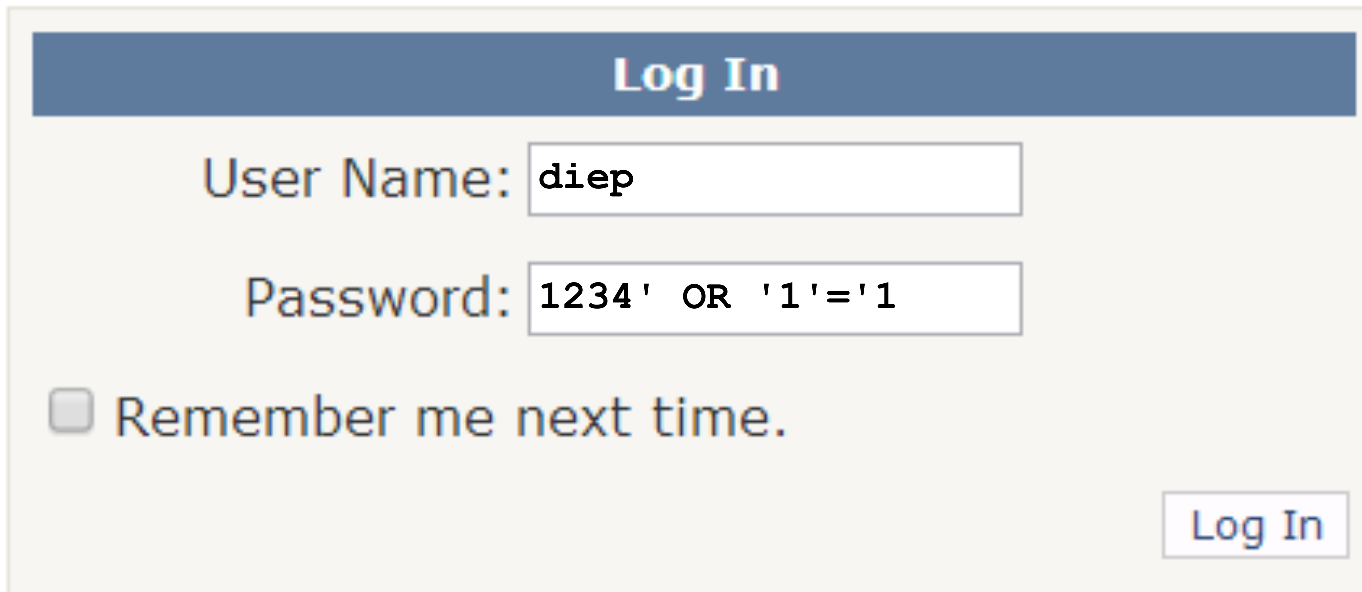
[2] Academia Sinica, Taiwan

[3] Brno University of Technology, Czech Republic

[4] Linköping University, Sweden

SQL Injections

- Consider a webpage that has two input fields: username and password



- The code behind the webpage is the following:

always TRUE

```
void Login_Authenticate(object sender, AuthenticateEventArgs e){
    SqlConnection con = new SqlConnection(@"Data Source=.\sqlexpress;Initial Catalog=... =True");
    string stmt = "select * from Table where name = '" + Name + "' and passwd = '" + Pas...
    adpt = new SqlDataAdapter(qry,con);
    dt = new DataTable();
    adpt.Fill(dt);
    if (dt.Rows.Count >= 1){
```

```
select * from Table where name = 'diep' and passwd = '1234' OR '1'='1'
```

Detect SQL Injection via String Constraints

Step 1: Identify variables

```
stmt = "select * from Table where name = `" + Name + "' and passwd = `" + Passwd + "'";
```

```
SQL_QUERY : "select * from Table where " EXPR
EXPR : COMP | EXPR or EXPR
COMP : TERM (=|>|<) TERM
TERM : [a-z]+[0-9]*| [0-9]+ | 'TERM'
```

context free grammar

Step 2: Find forbidden patterns

SQL injection is to create a SQL query that contains " or '1' = '1'"

Step 3: Transform to string constraints

Step 4: Solve the string constraints

Detect SQL Injection via String Constraints

Step 1: Identify variables

```
stmt = "select * from Table where name = '" + Name + "' and passwd = '" + Passwd + "'";
```

```
SQL_QUERY : "select * from Table where " EXPR
EXPR : COMP | EXPR or EXPR
COMP : TERM (=|>|<) TERM
TERM : [a-z]+[0-9]*| [0-9]+ | 'TERM'
```

Step 2: Find forbidden patterns

SQL injection is to create a SQL query that contains " or '1' = '1' "

Step 3: Transform to string constraints

```
stmt = "select * from Table where name = '" . Name .
                                 "' and passwd = '" . Passwd . "'"

SQL_injection ∈ L(SQL_QUERY);
SQL_injection = A . " or '1' = '1'" . B
stmt = SQL_injection
```

concatenation

mem

equality

Step 4: Solve the string constraints

String Solver

✓ **Applications**

- Detect vulnerabilities in web applications

    SQL Injection

    Code Injection

- Used in Program Testing, Program Verification, Model Checking

✓ Requirements

- Arithmetic constraints `length(A) > 5`
- String equations `stmt = A . " or '1 = 1'" . B`
- Context free grammar membership `stmt ∈ L(SQL_QUERY);`

...

String Solver

✓ **Applications**

- Detect vulnerabilities in web applications

    SQL Injection

    Code Injection

- Used in Program Testing, Program Verification, Model Checking

✓ **Requirements**

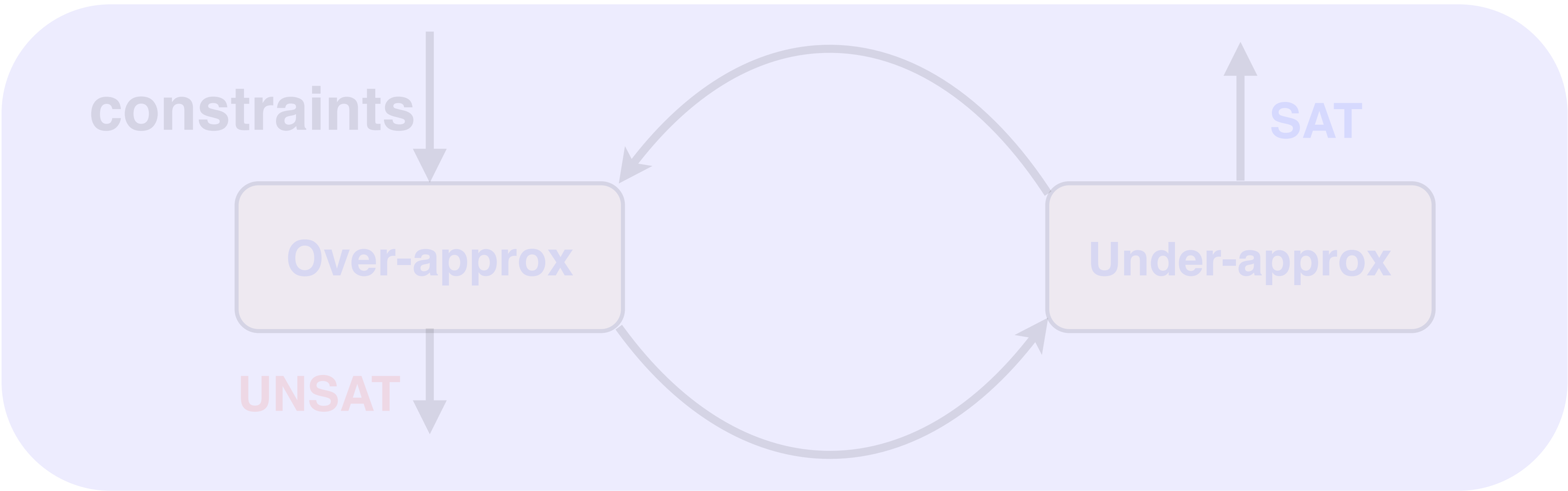
- Arithmetic constraints `length(A) > 5`
- String equations `stmt = A . " or '1 = 1'" . B`
- Context free grammar membership `stmt ∈ L(SQL_QUERY);`

...

Contributions

1. New framework for solving string constraints:

- Handle rich class of constraints: CFG membership, transducer, etc.
- Based on Counter-Example Guided Abstract Refinement.



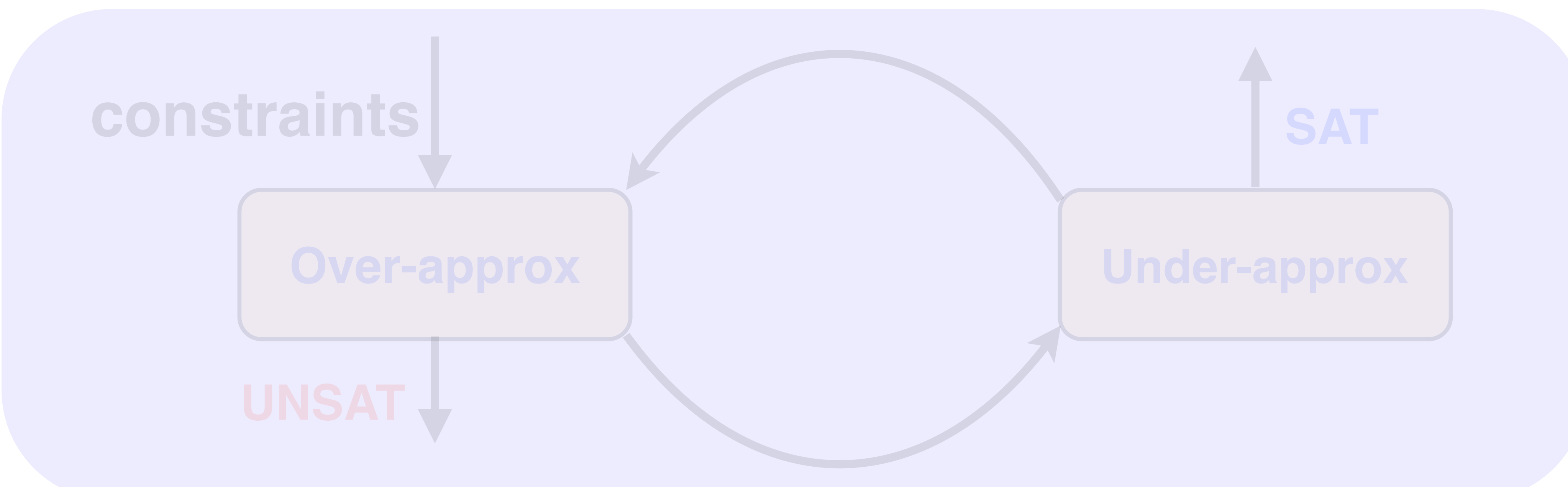
2. Open-source tool: outperforming all existing tools.

Contributions

no other tools can do

1. New framework for solving string constraints:

- Handle rich class of constraints: CFG membership, transducer, etc.
- Based on Counter-Example Guided Abstract Refinement.



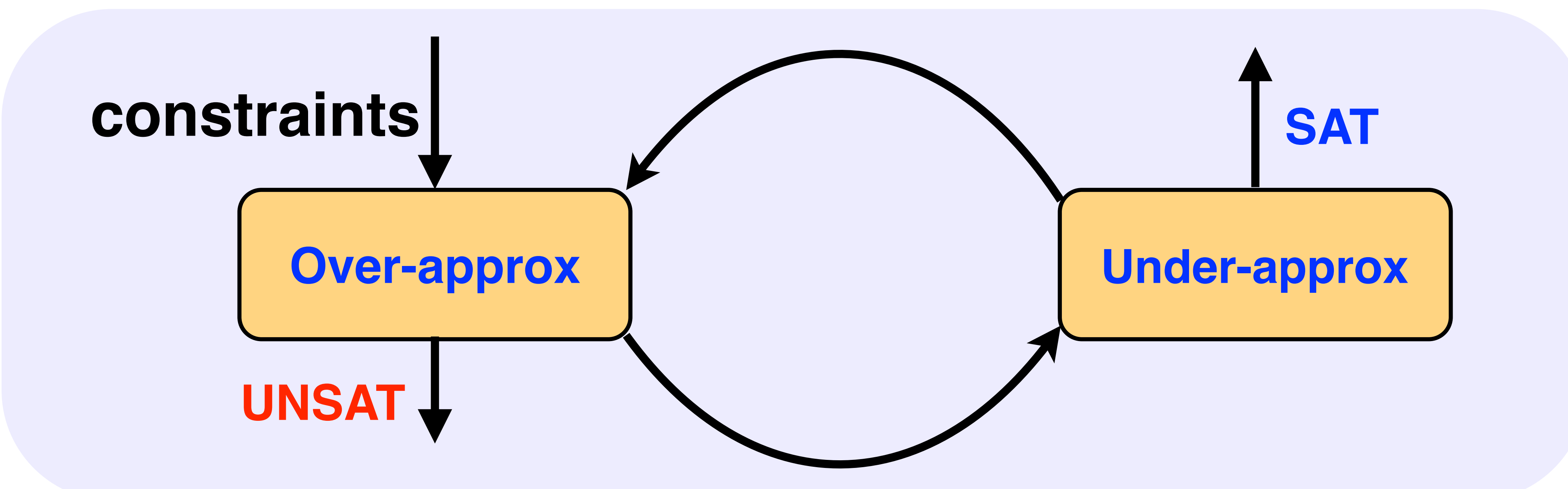
2. Open-source tool: outperforming all existing tools.

no other tools can do

Contributions

1. New framework for solving string constraints:

- Handle rich class of constraints: CFG membership, transducer, etc.
- Based on Counter-Example Guided Abstract Refinement.



2. Open-source tool: outperforming all existing tools.

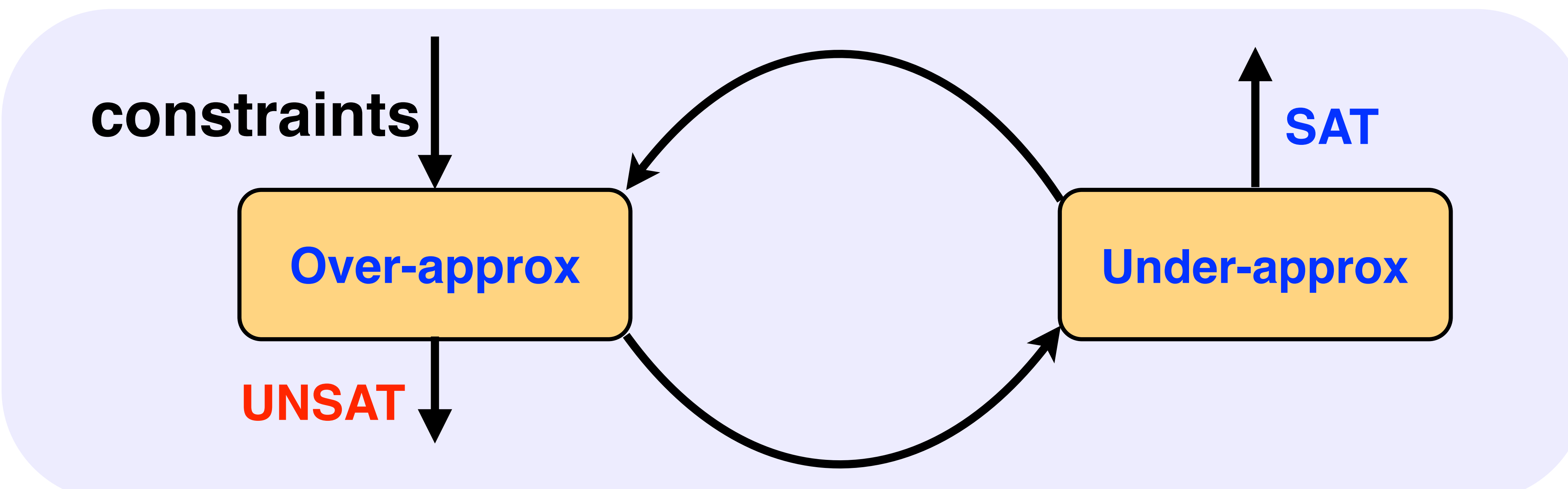
no other tools can do

Contributions

1. New framework for solving string constraints:

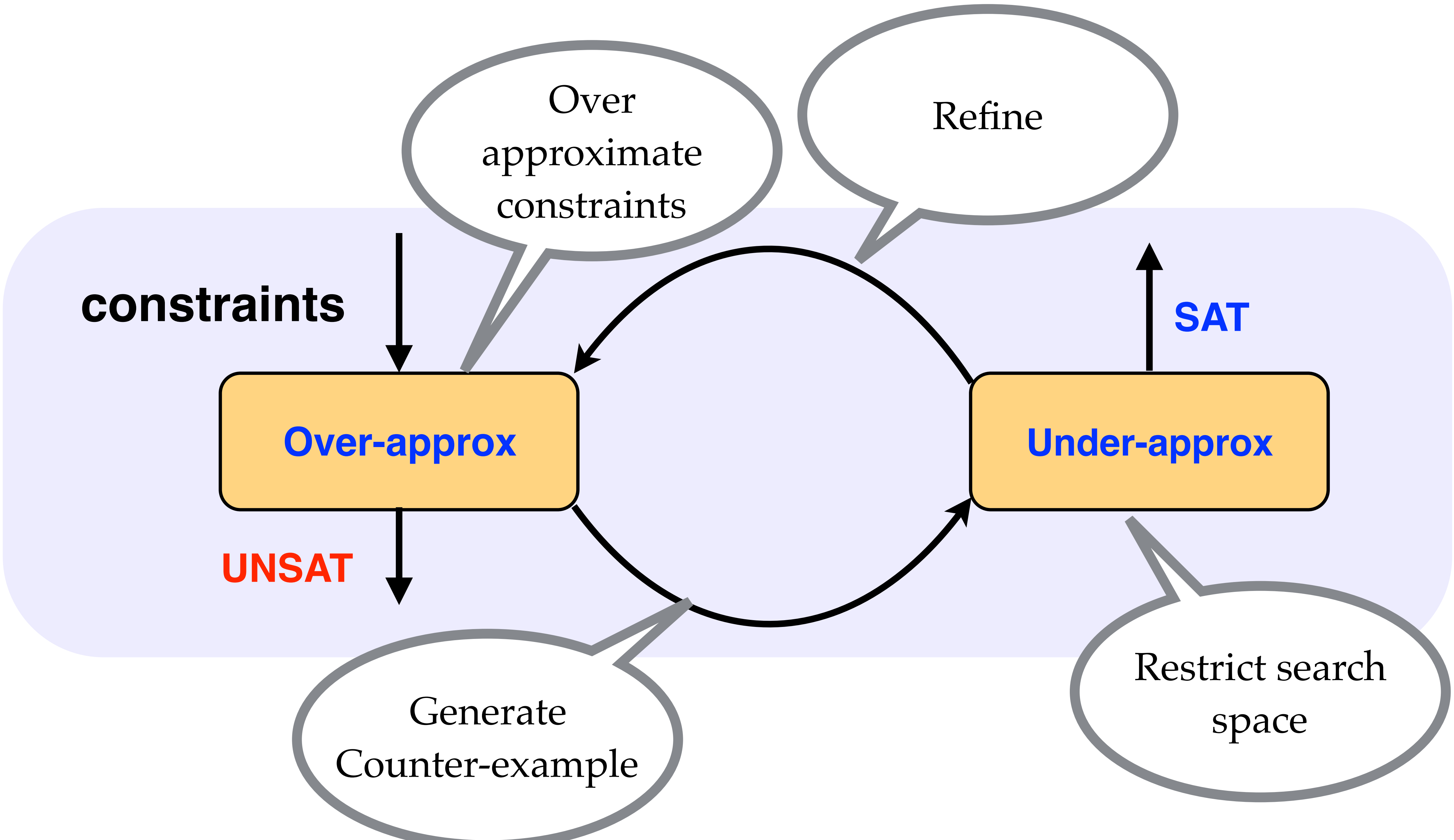
- Handle rich class of constraints: CFG membership, transducer, etc.

- Based on Counter-Example Guided Abstract Refinement.



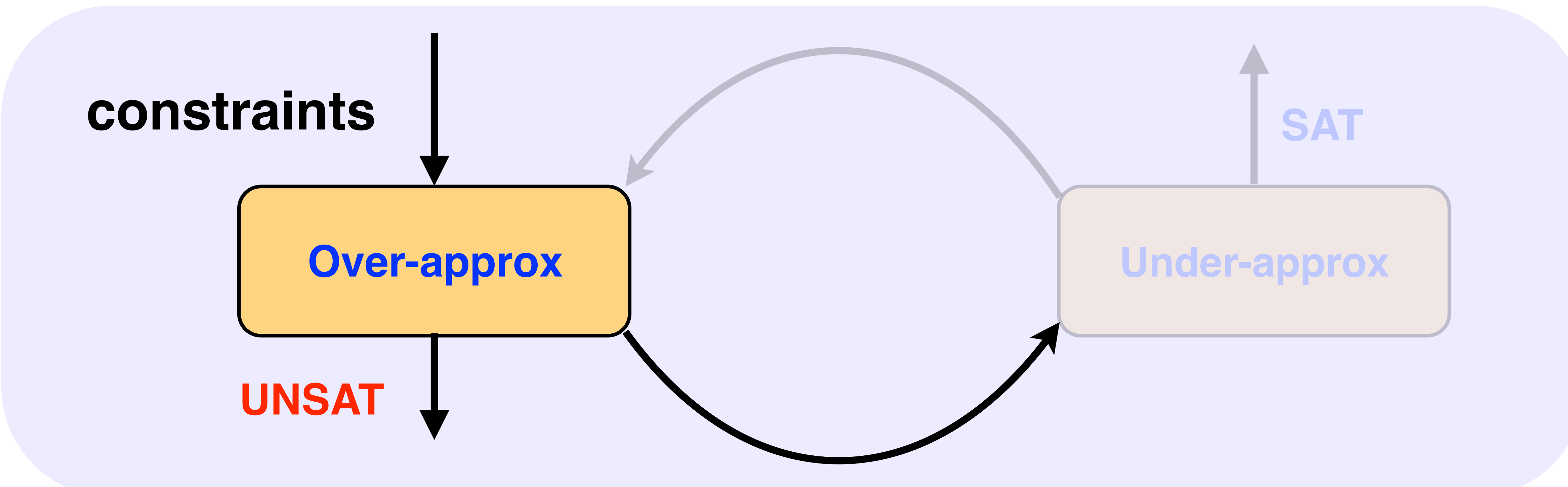
2. Open-source tool: outperforming all existing tools.

Overview



Using CEGAR for string constraint solving

Overview



Using CEGAR for string constraint solving

Running Example

Grammar

$S : a\ S\ b \mid S\ b \mid \varepsilon$

$X, Y \in L(S)$

$X = \text{"a"} \,.\, Y$

$X = Z$

Membership

Equality

Step 1: Over approximate CFG constraints to regular constraints

S : a S b | S b | ε
X, Y ∈ L(S)

$\Rightarrow$

X, Y ∈ L($a^* b^*$)

Step 2: Rename each occurrence of variables in equalities

X = “a” . Y
X = Z

$\Rightarrow$

X_1 = “a” . Y
X_2 = Z

S : a S b | S b | ε
X, Y ∈ L(S)
X = “a” . Y
X = Z

$\Rightarrow$

X_1, X_2, Y ∈ L($a^* b^*$)
X_1 = “a” . Y
X_2 = Z

Step 1: Over approximate CFG constraints to regular constraints

S : a S b | S b | ε
$X, Y \in L(S)$

⇒

$X, Y \in L(a^* b^*)$

Step 2: Rename each occurrence of variables in equalities

$X = $ "a" $. Y$
$X = Z$

⇒

$X_1 = $ "a" $. Y$
$X_2 = Z$

S : a S b | S b | ε
$X, Y \in L(S)$
$X = $ "a" $. Y$
$X = Z$

$X_1, X_2, Y \in L(a^* b^*)$
$X_1 = $ "a" $. Y$
$X_2 = Z$

Step 1: Over approximate CFG constraints to regular constraints

S : a S b | S b | ε
X, Y ∈ L(S)

⇒

X, Y ∈ L(a* b*)

Step 2: Rename each occurrence of variables in equalities

X = "a" . Y
X = Z

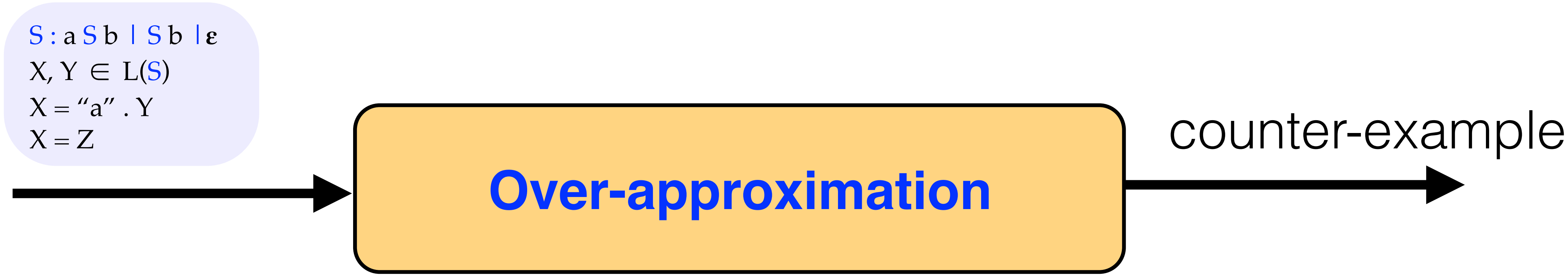
⇒

X_1 = "a" . Y
X_2 = Z

S : a S b | S b | ε
X, Y ∈ L(S)
X = "a" . Y
X = Z

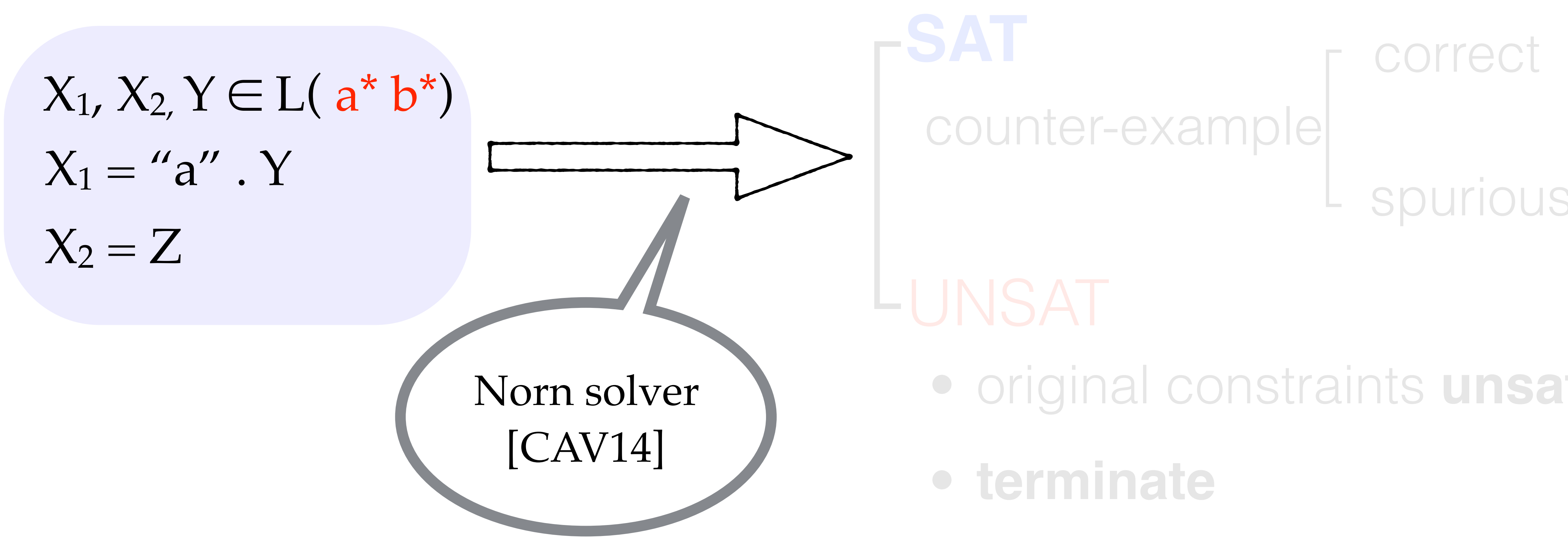
⇒

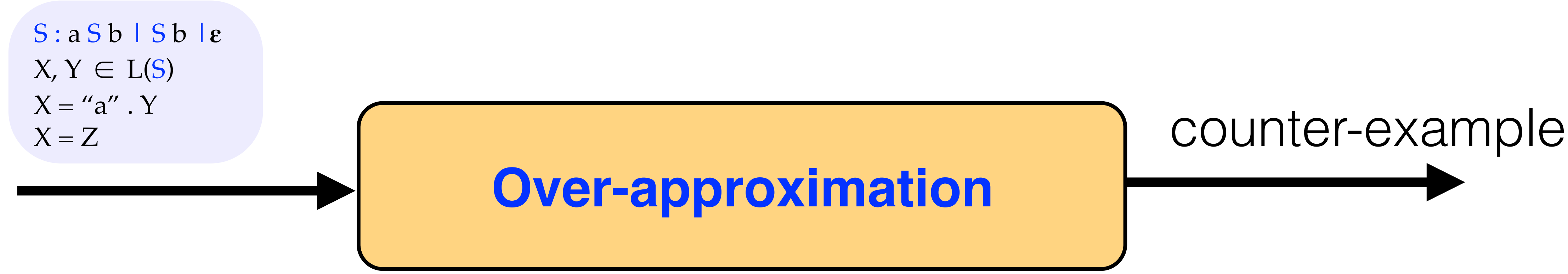
X_1, X_2, Y ∈ L(a* b*)
X_1 = "a" . Y
X_2 = Z



Step 3: Solve the approximate constraints

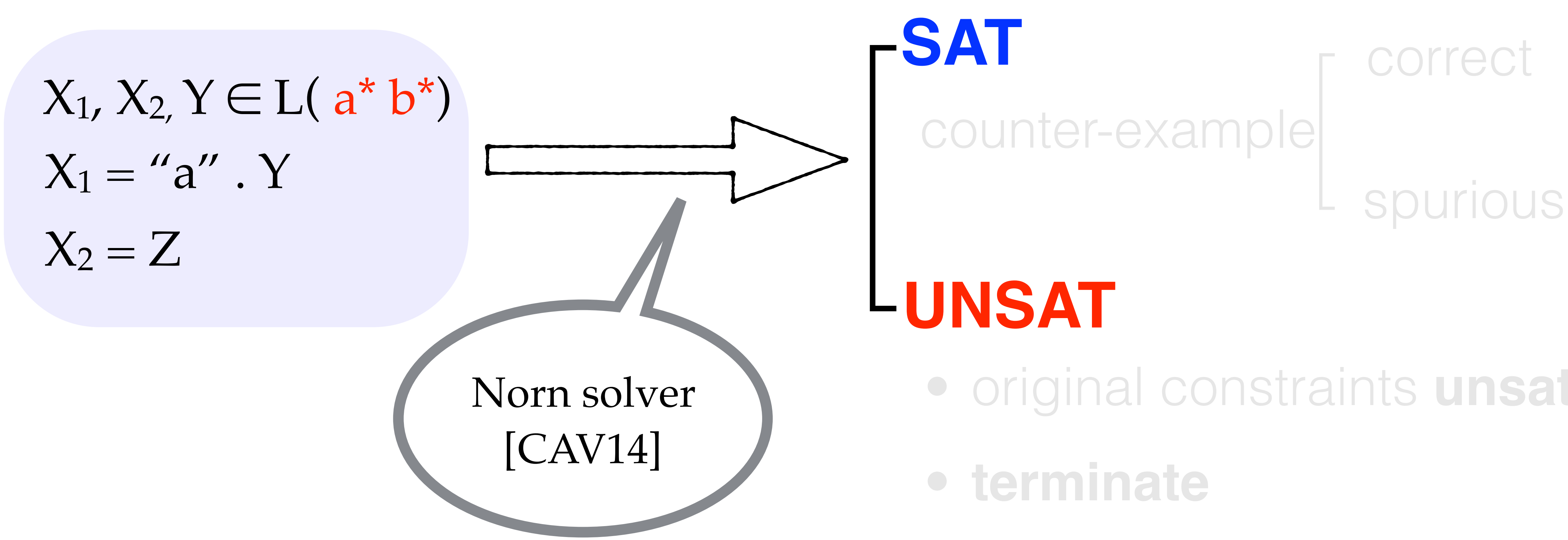
The new constraints fall into decidable fragments, can be handled efficiently.

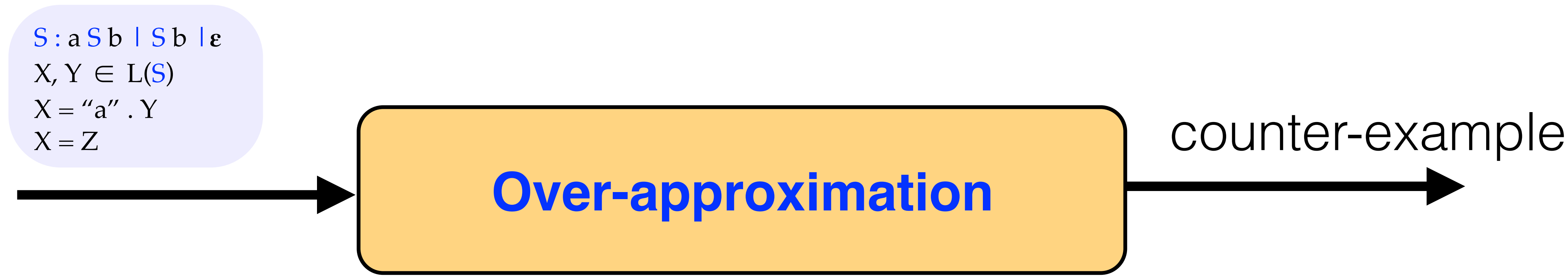




Step 3: Solve the approximate constraints

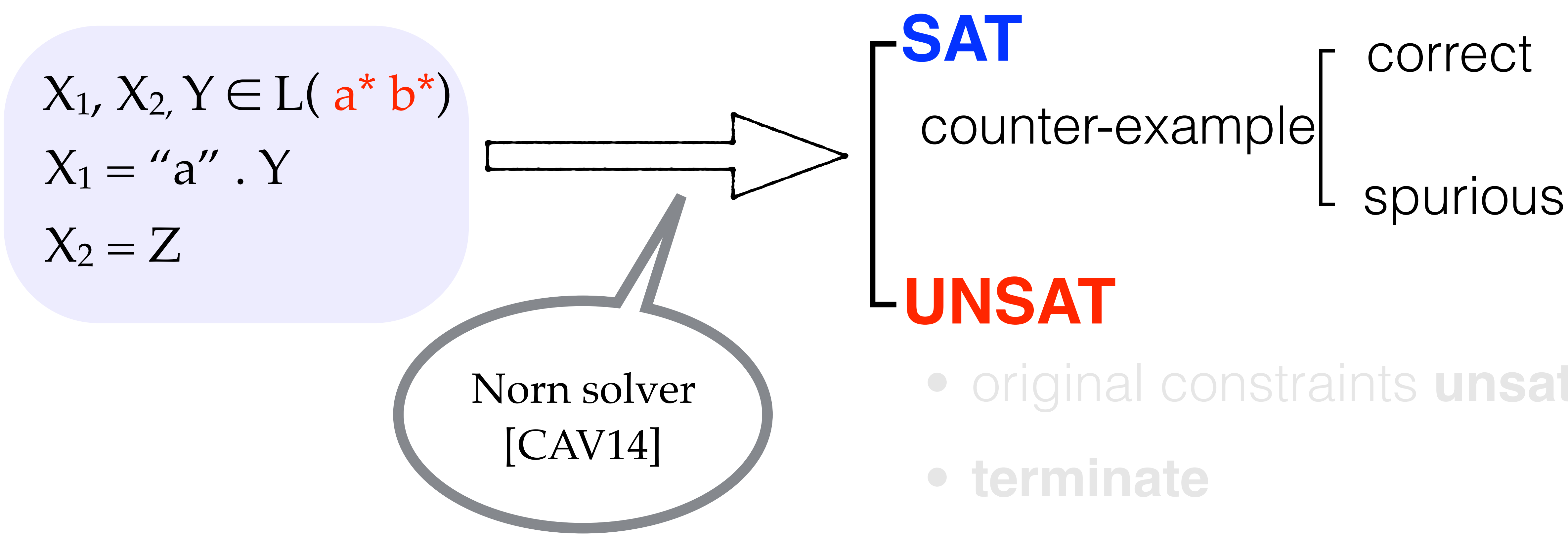
The new constraints fall into decidable fragments, can be handled efficiently.

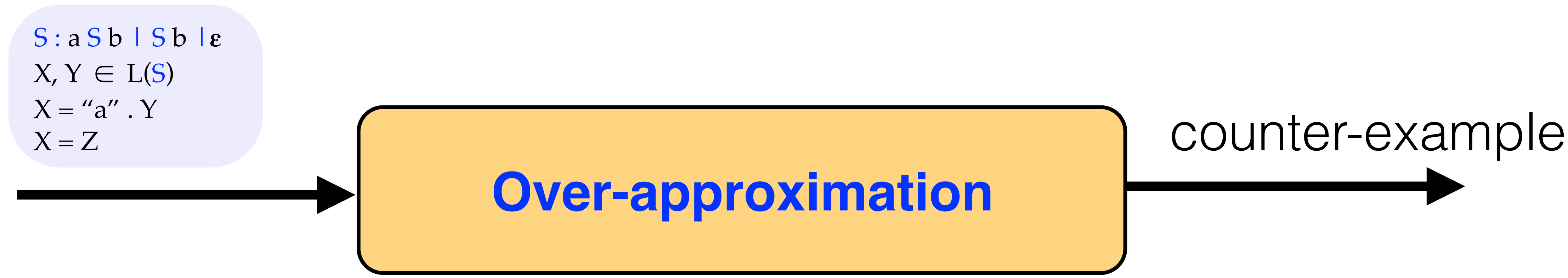




Step 3: Solve the approximate constraints

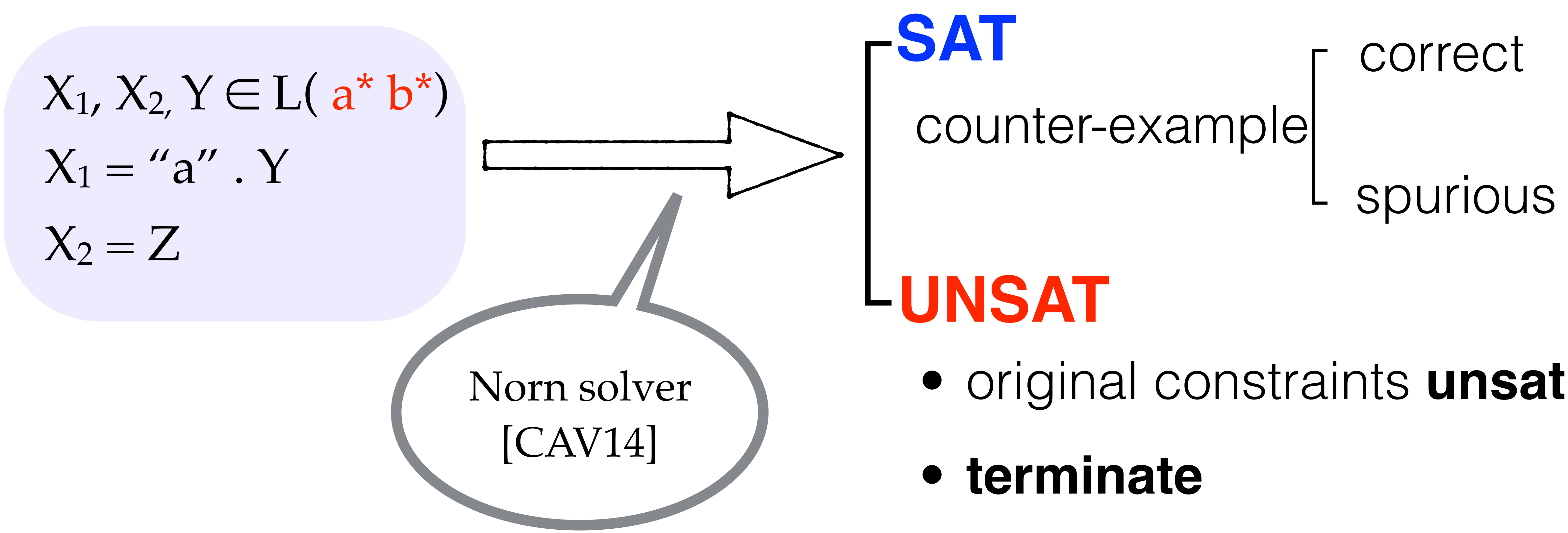
The new constraints fall into decidable fragments, can be handled efficiently.





Step 3: Solve the approximate constraints

The new constraints fall into decidable fragments, can be handled efficiently.



S : a S b | S b | ε

X, Y ∈ L(S)

X = "a" . Y

X = Z

Over-approximation → counter-example

Step 3: Solve the approximate constraints

The new constraints fall into decidable fragments, can be handled efficiently.

$X_1, X_2, Y \in L(a^* b^*)$

$X_1 = \text{"a"} . Y$

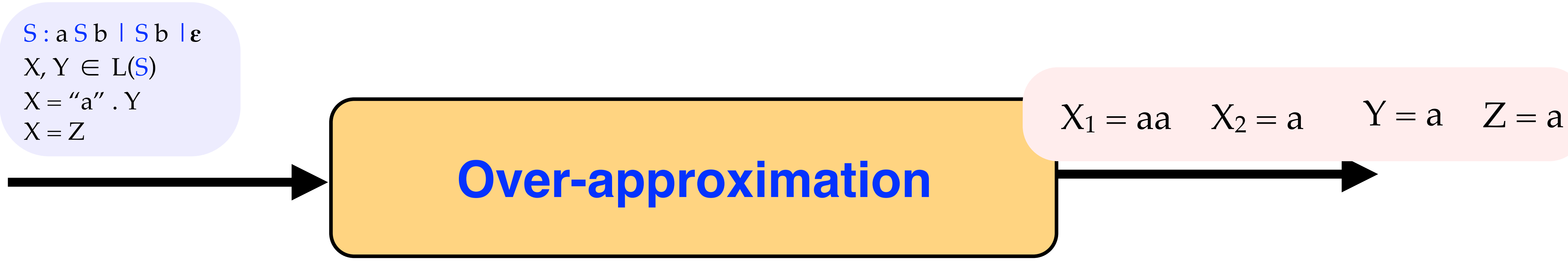
$X_2 = Z$

Norn solver [CAV14]

SAT

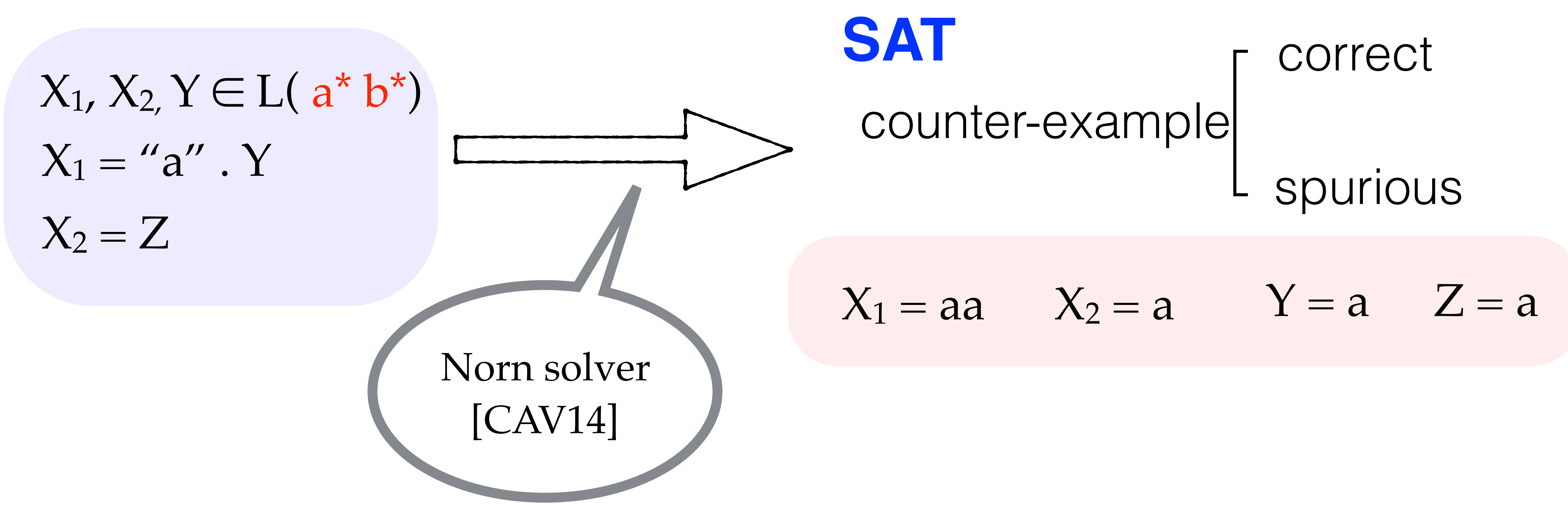
counter-example: correct / spurious

$X_1 = aa$ $X_2 = a$ $Y = a$ $Z = a$

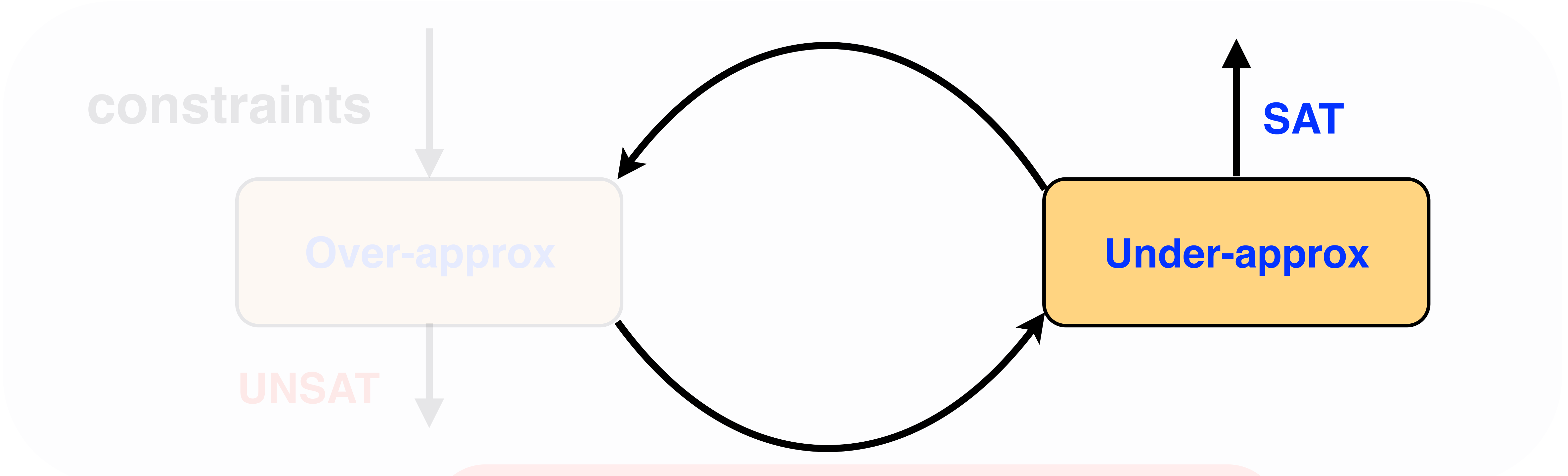


Step 3: Solve the approximate constraints

The new constraints fall into decidable fragments, can be handled efficiently.



Overview



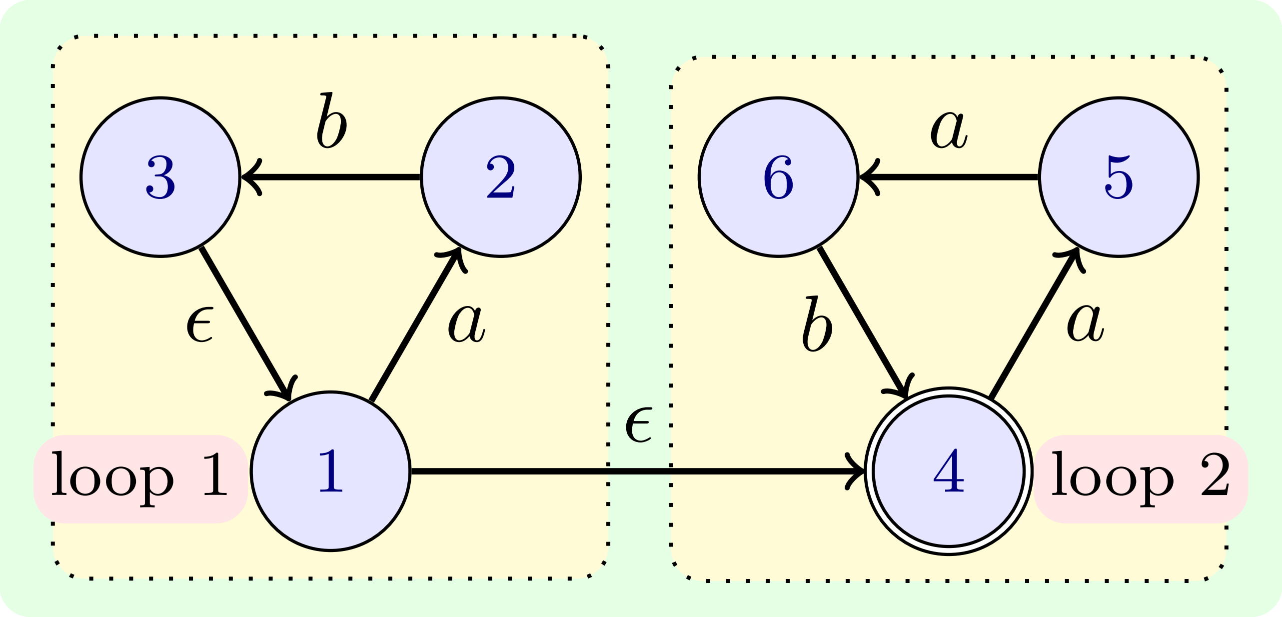
$X_1 = aa \quad X_2 = a \quad Y = a \quad Z = a$

Flat Automata

basic concepts

Definition:

- finite state automata
- consist of a sequence of simple loops



(ab)* (aab)*

Idea: search for solutions accepted by **flat automata**

Step 1: Generate the minimal flat automaton that accepts the counter-example

Step 2: Intersect the constraints with the generated flat automaton

$S : a\,S\,b \mid S\,b \mid \varepsilon$
$X, Y \in L(S)$
$X = \text{“a”} . Y$
$X = Z$

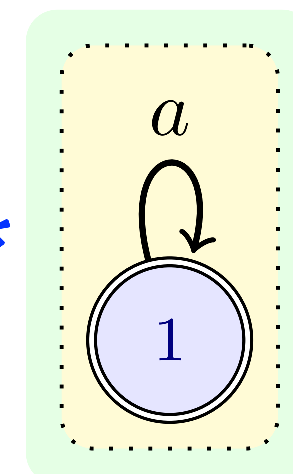
$\cap\ a^*$ ⇒

$X, Y \in L(\varepsilon)$
$X = \text{“a”} . Y$
$X = Z$
$X, Y, Z \in L(a^*)$

Idea: search for solutions accepted by **flat automata**

Step 1: Generate the minimal flat automaton that accepts the counter-example

$X_1 = aa$ $X_2 = a$ $Y = a$ $Z = a$ ⇒ a^*



Step 2: Intersect the constraints with the generated flat automaton

S : a S b | S b | ε
X, Y ∈ L(S)
X = "a" . Y
X = Z

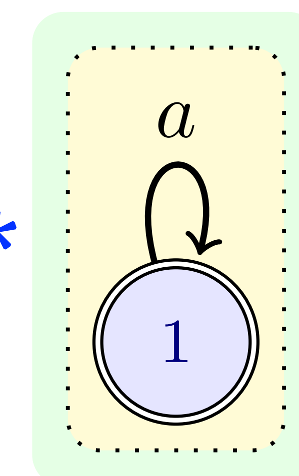
∩ a^* ⇒

X, Y ∈ L(ε)
X = "a" . Y
X = Z
Z ∈ L(a^*)

Idea: search for solutions accepted by **flat automata**

Step 1: Generate the minimal flat automaton that accepts the counter-example

$X_1 = aa \quad X_2 = a \quad Y = a \quad Z = a \;\Rightarrow\; a^*$



Step 2: Intersect the constraints with the generated flat automaton a^*

S : a S b | S b | ε

X, Y ∈ L(S)

X = “a” . Y

X = Z

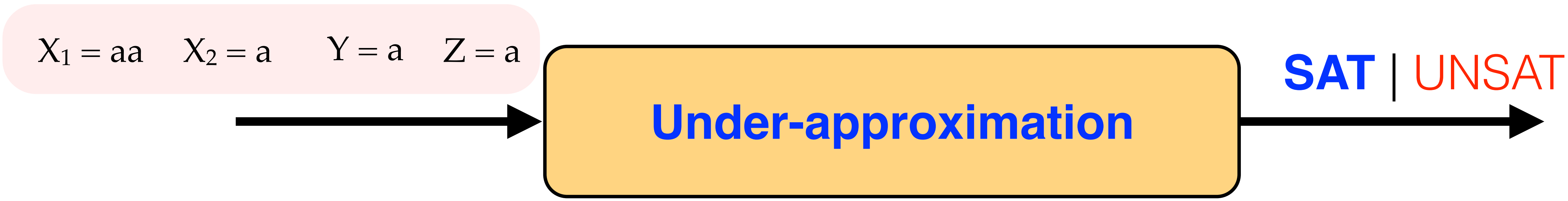
$\cap \; a^* \;\Rightarrow$

X, Y ∈ L(ε)

X = “a” . Y

X = Z

Z ∈ L(a*)



Step 3: Convert to quantifier-free Presburger formulas

$X, Y \in L(\varepsilon)$

$X = \text{“a”} . Y$

$X = Z$

$Z \in L(a^*)$

⇒

$|X| = 0$

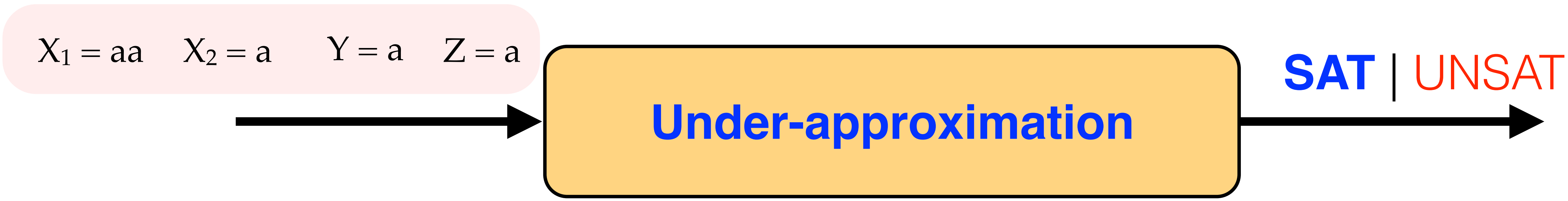
$|Y| = 0$

$|X| = 1 + |Y|$

$|X| = |Z|$

$|X|, |Y|, |Z| \geq 0$

Step 4: Feed the formulas to a SMT solver



Step 3: Convert to quantifier-free Presburger formulas

$X, Y \in L(\varepsilon)$

$X = \text{“a”} \, . \, Y$

$X = Z$

$Z \in L(a^*)$

⇒

$|X| = 0$

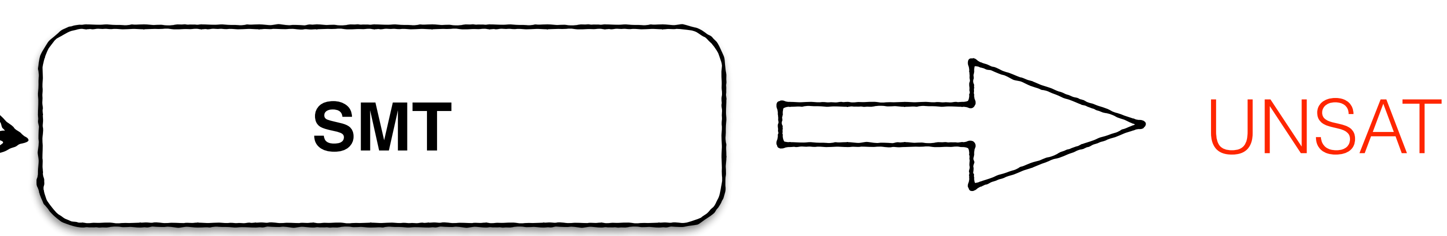
$|Y| = 0$

$|X| = 1 + |Y|$

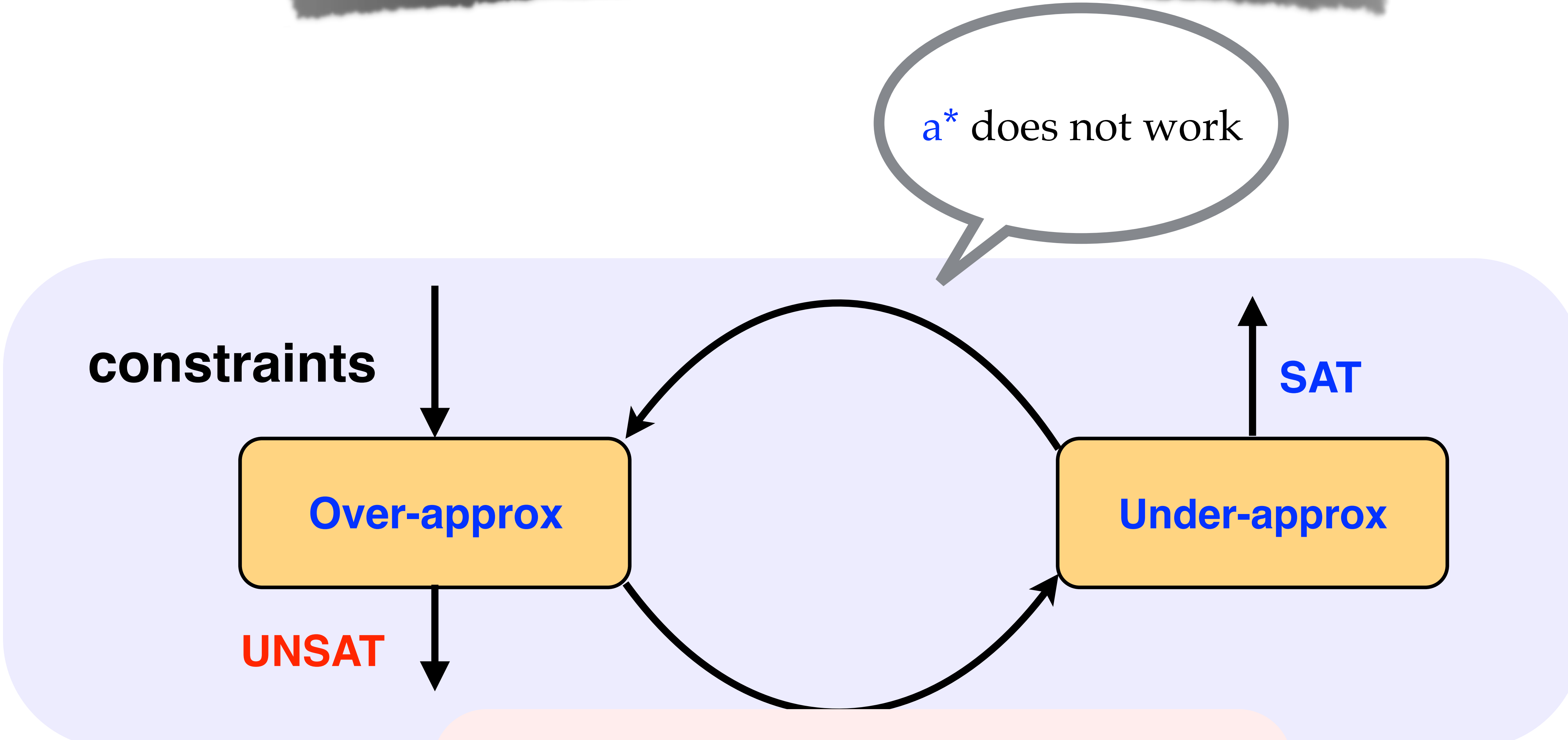
$|X| = |Z|$

$|X|, |Y|, |Z| \geq 0$

Step 4: Feed the formulas to a SMT solver



Overview



$X_1 = aa$ $X_2 = a$ $Y = a$ $Z = a$

Step 1: Over approximate CFG constraints to regular constraints

Step 2: Rename each occurrence of variables in equalities

S : a S b | S b | ε

X, Y ∈ L(S)

X = "a" . Y

X = Z

⇒

$X_1, X_2, Y \in L(a^* b^*)$

X_1 = "a" . Y

X_2 = Z

Step 3: Refine the over-approximation

Step 1: Over approximate CFG constraints to regular constraints

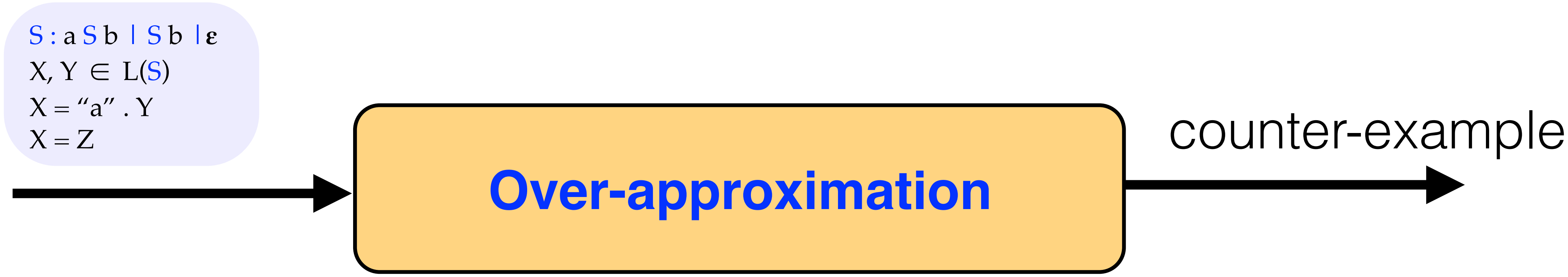
Step 2: Rename each occurrence of variables in equalities

S : a S b | S b | ε X, Y ∈ L(S) X = "a" . Y X = Z	⇒	X_1, X_2, Y ∈ L(a* b*) X_1 = "a" . Y X_2 = Z

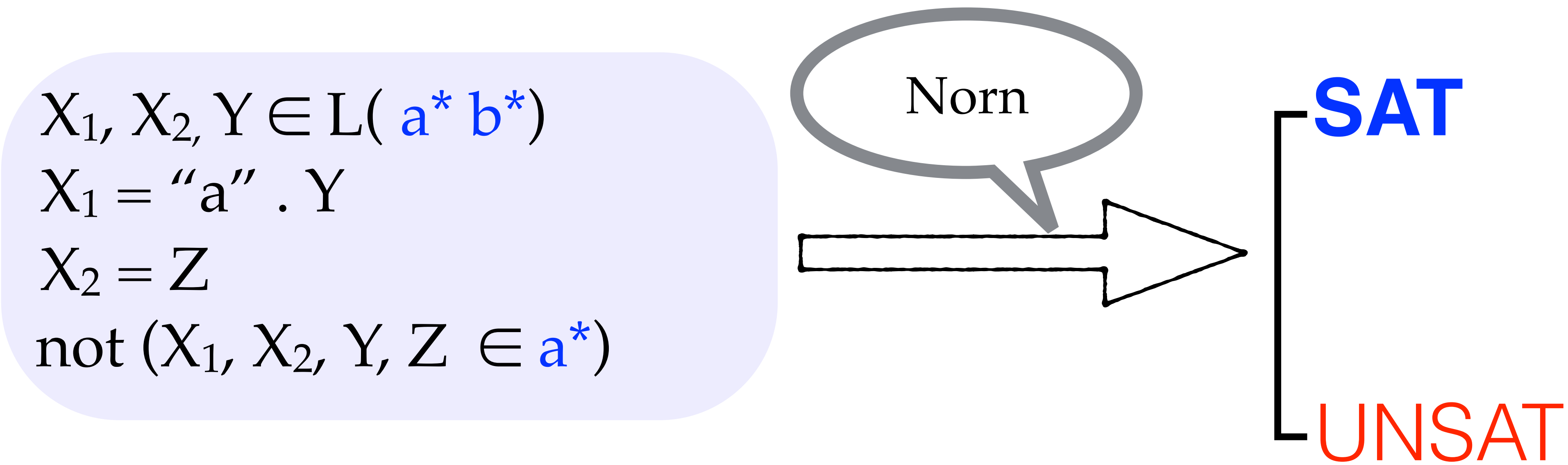
Step 3: Refine the over-approximation

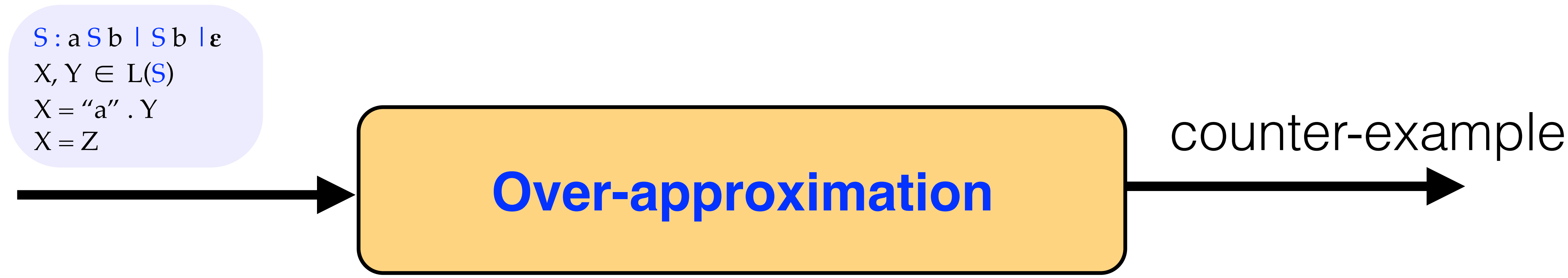
NEW

not (X, Y, Z ∈ L(a*)) ⇒ **not (X_1, X_2, Y, Z ∈ L(a*))**



Step 4: Solve the approximate constraints





Step 4: Solve the approximate constraints

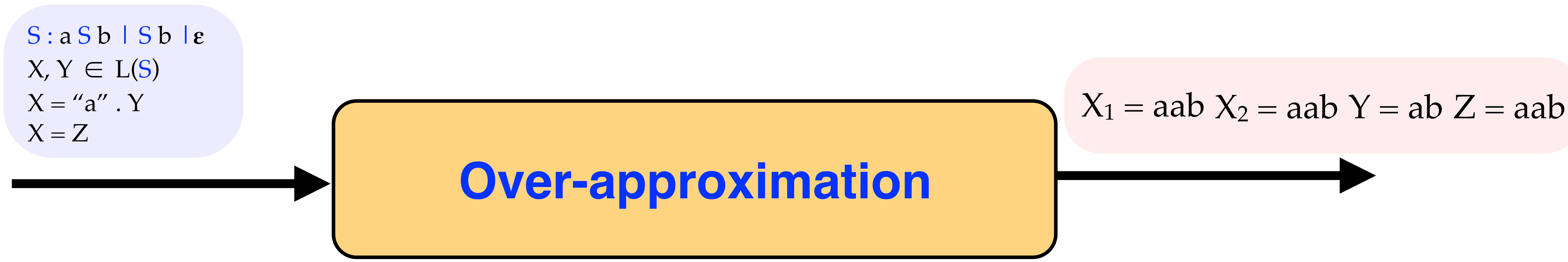
$X_1, X_2, Y \in L(a^* b^*)$
$X_1 = \text{"a"} . Y$
$X_2 = Z$
not $(X_1, X_2, Y, Z \in a^*)$

Norn

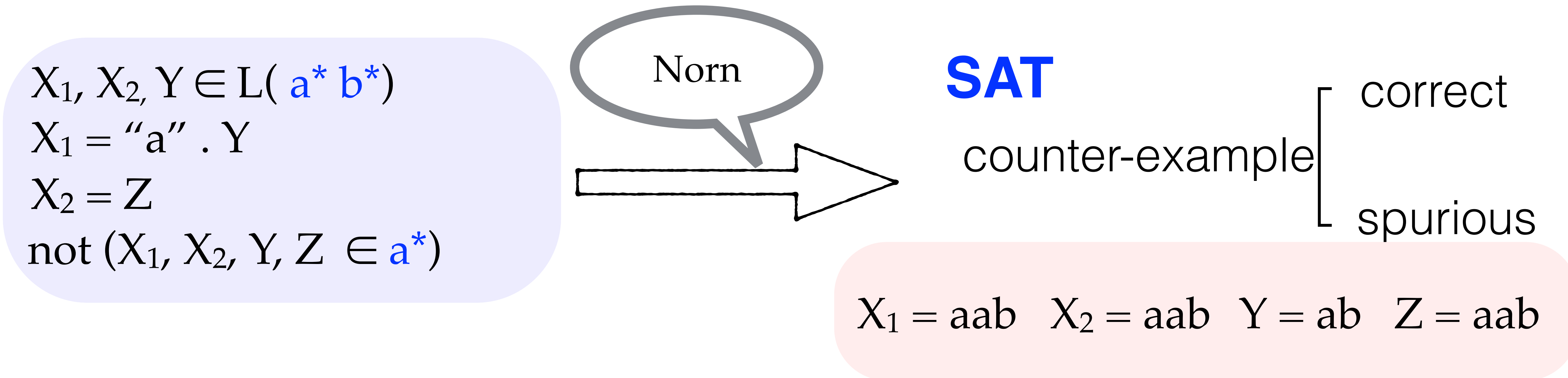
SAT

counter-example
- correct
- spurious

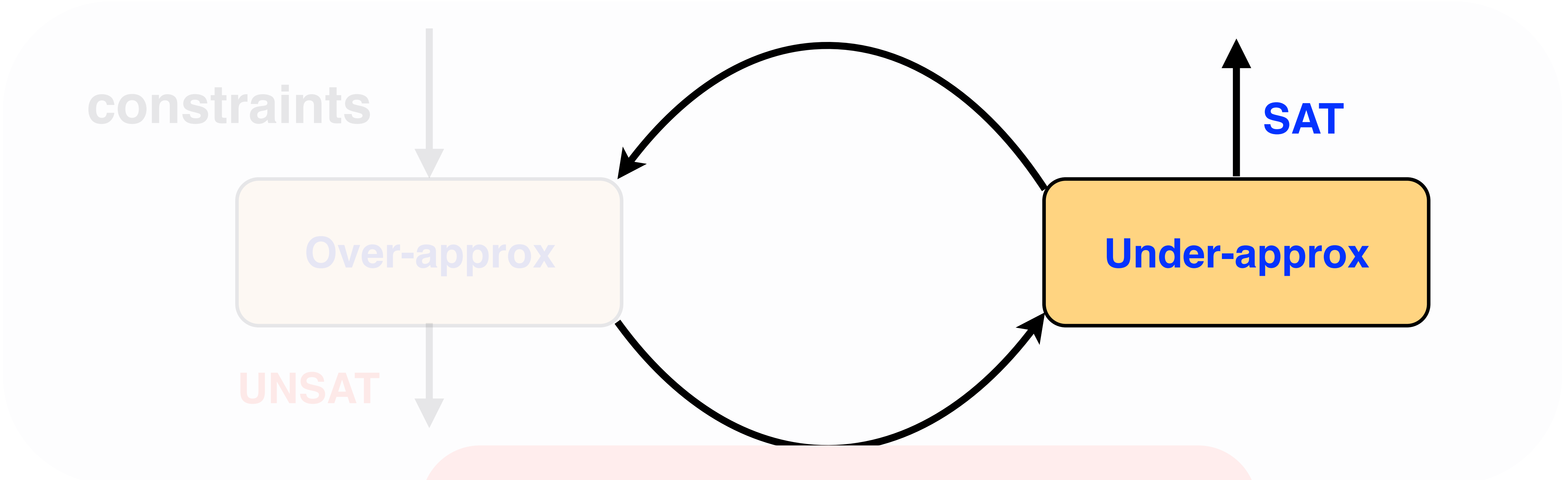
$X_1 = aab \quad X_2 = aab \quad Y = ab \quad Z = aab$



Step 4: Solve the approximate constraints



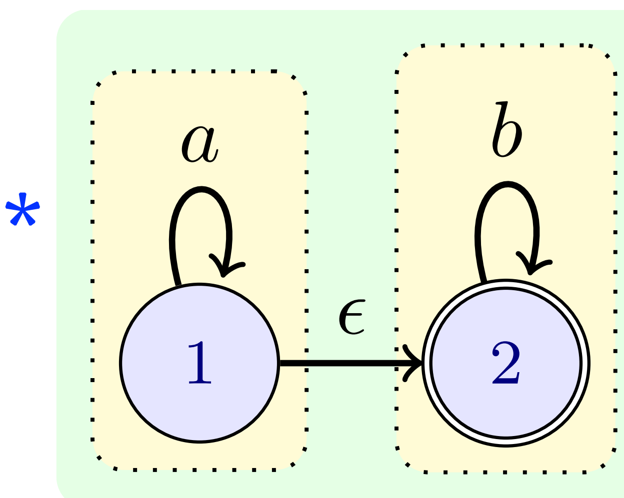
Overview



$X_1 = aab \quad X_2 = aab \quad Y = ab \quad Z = aab$

Step 1: Generate the minimal flat automaton that accepts the counter-example

$X_1 = aab \quad X_2 = aab \quad Y = ab \quad Z = aab$ ⇒ a^*b^*



Step 2: Intersect the constraints with the generated flat automaton

S : a S b | S b | ε
X, Y ∈ L(S)
X = "a" . Y
X = Z

∩ a^*b^* ⇒

X, Y ∈ L(ab^+ | b^*)
X = "a" . Y
X = Z
Z ∈ L(a^*b^*)

Step 3: Convert to quantifier-free Presburger formulas

$X, Y \in L(ab^+ \mid b^*)$

$X = \text{"a"} \, . \, Y$

$X = Z$

$Z \in L(a^*b^*)$

$$X, Y \in L(ab^+ \mid b^*) \longleftrightarrow \left[\begin{array}{l} X \in L(ab^+) \text{ and } Y \in L(ab^+) \\ X \in L(ab^+) \text{ and } Y \in L(b^*) \\ X \in L(b^*) \text{ and } Y \in L(ab^+) \\ X \in L(b^*) \text{ and } Y \in L(b^*) \end{array}\right.$$

Step 3: Convert to quantifier-free Presburger formulas

$X, Y \in L(ab^+ \mid b^*)$

$X = \text{"a"} . Y$

$X = Z$

$Z \in L(a^*b^*)$

$$X, Y \in L(ab^+ \mid b^*) \leftrightarrow \left[\begin{array}{l} X \in L(ab^+) \text{ and } Y \in L(ab^+) \\ X \in L(ab^+) \text{ and } Y \in L(b^*) \\ X \in L(b^*) \text{ and } Y \in L(ab^+) \\ X \in L(b^*) \text{ and } Y \in L(b^*) \end{array} \right.$$

Step 3: Convert to quantifier-free Presburger formulas

X, Y ∈ L(ab⁺ | b*)
X = "a" . Y
X = Z
Z ∈ L(a*b*)

→

X ∈ L(ab⁺) and Y ∈ L(b*)
X = "a" . Y
X = Z
Z ∈ L(a*b*)

X, Y ∈ L(ab⁺ | b*) ⟷

X ∈ L(ab⁺) and Y ∈ L(ab⁺)
X ∈ L(ab⁺) and Y ∈ L(b*)
X ∈ L(b*) and Y ∈ L(ab⁺)
X ∈ L(b*) and Y ∈ L(b*)

Step 3: Convert to quantifier-free Presburger formulas

$X \in L(ab^+)$ and $Y \in L(b^*)$

$X = “a” . Y$

$X = Z$

$Z \in L(a^*b^*)$

⇒

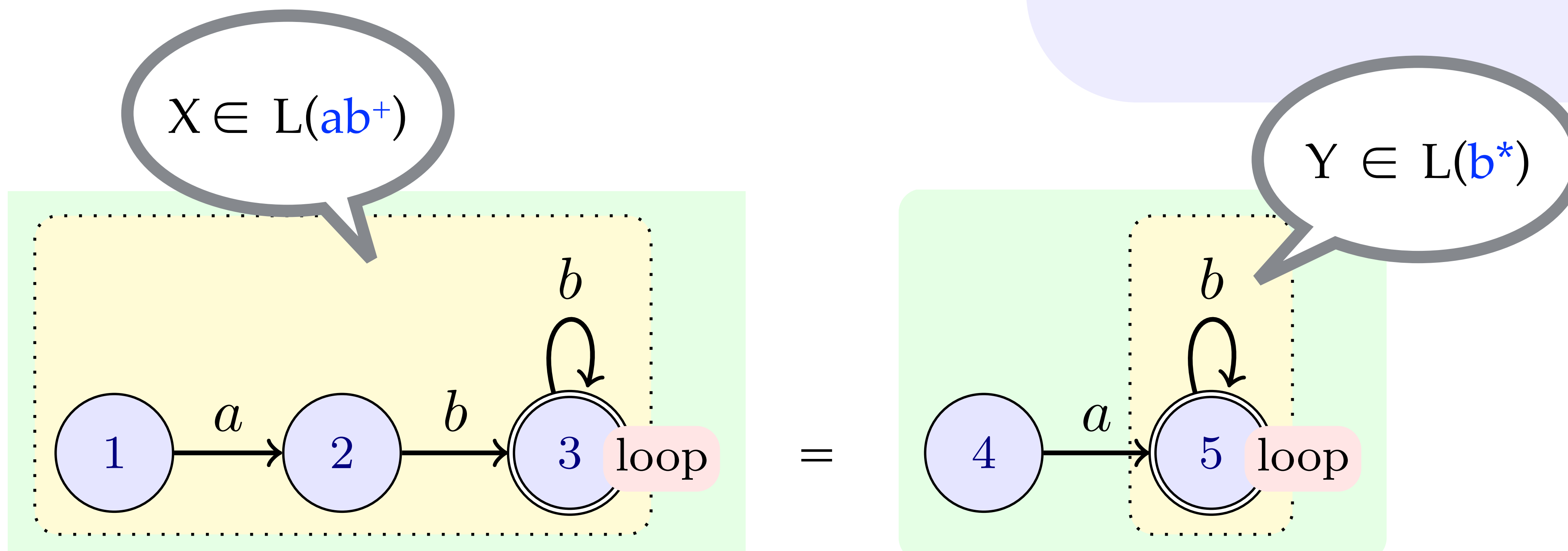
$|X| = 1 + |Y|$

$|X| = |Z|$

$|X|, |Y|, |Z| \geq 0$

$\#Y(“a”) = 0$, $\#X(“a”) = 1$

$\#Y(“b”) = \#X(“b”)$



$\#X(“a”)$: number of occurrences of “a” in X

Step 3: Convert to quantifier-free Presburger formulas

$X \in L(ab^+)$ and $Y \in L(b^*)$

$X = “a” . Y$

$X = Z$

$Z \in L(a^*b^*)$

$|X| = 1 + |Y|$

$|X| = |Z|$

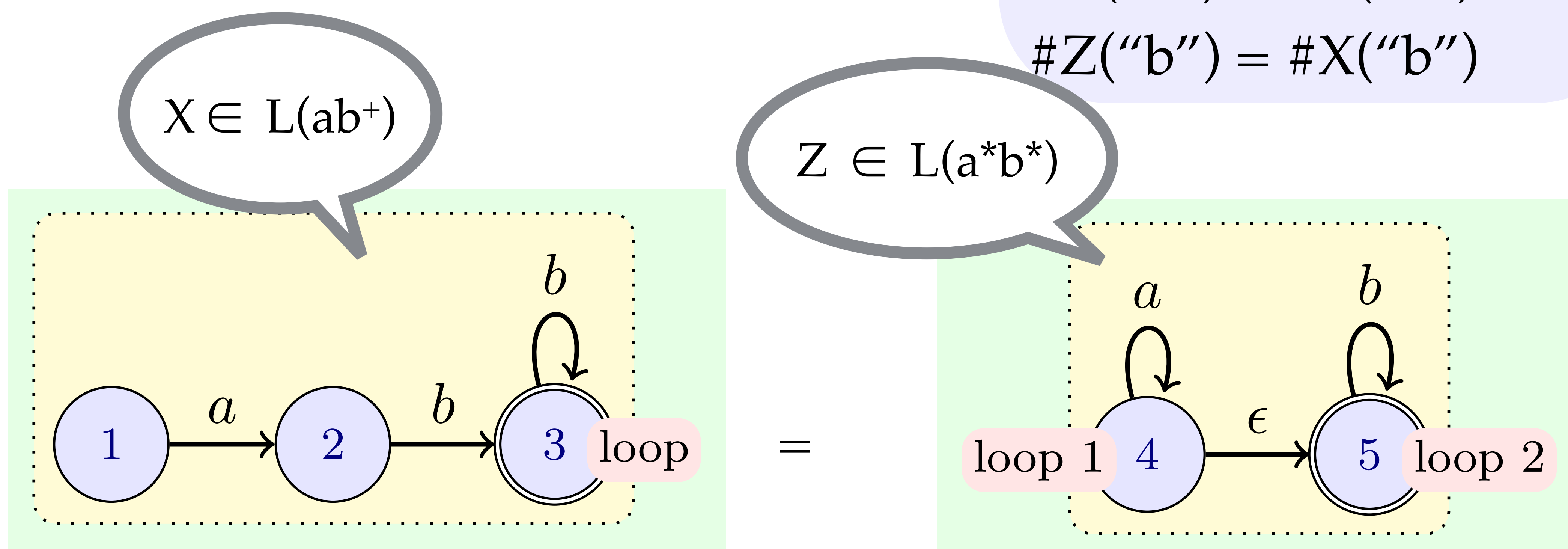
$|X|, |Y|, |Z| \geq 0$

$\#Y(“a”) = 0$, $\#X(“a”) = 1$

$\#Y(“b”) = \#X(“b”)$

$\#Z(“a”) = \#X(“a”)$

$\#Z(“b”) = \#X(“b”)$



$\#X(“a”)$: number of occurrences of “a” in X

Step 3: Convert to quantifier-free Presburger formulas

$X \in L(ab^+)$ and $Y \in L(b^*)$

$X = \text{"a"} . Y$

$X = Z$

$Z \in L(a^*b^*)$

$\Rightarrow$

$|X| = 1 + |Y|$

$|X| = |Z|$

$|X|, |Y|, |Z| \geq 0$

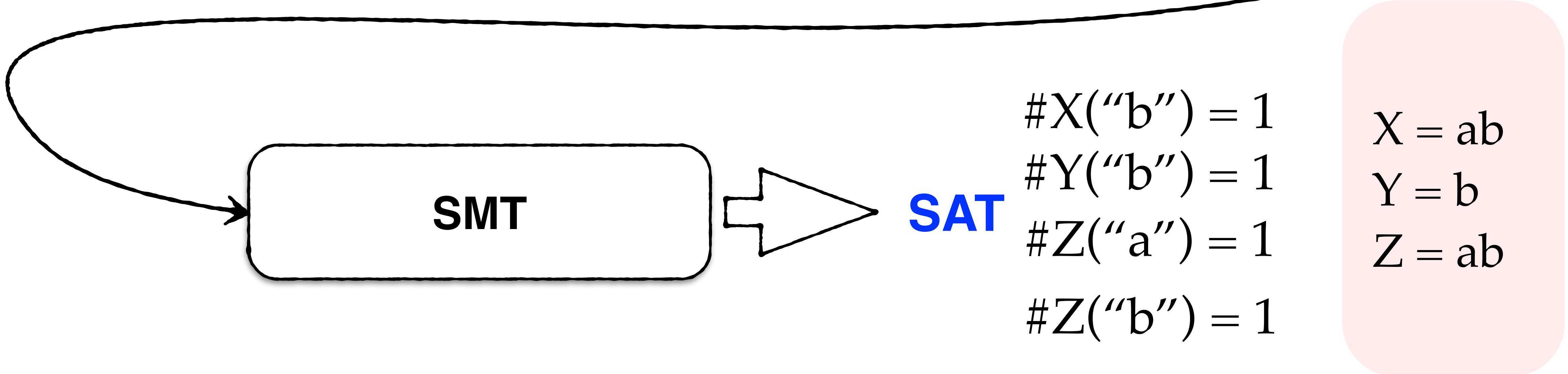
$\#Y(\text{"a"}) = 0$, $\#X(\text{"a"}) = 1$

$\#Y(\text{"b"}) = \#X(\text{"b"})$

$\#Z(\text{"a"}) = \#X(\text{"a"})$

$\#Z(\text{"b"}) = \#X(\text{"b"})$

Step 4: Feed the formulas to a SMT solver



Experiment Results

✓ Open-source tool: TRAU

✓ Use Z3 as a backend tool

✓ Run on the standard Kaluza & SQL injection benchmarks

- Kaluza: ~50,000 tests

  Javascript symbolic execution engine

- SQL injection: 10 tests

  detect SQL injections with CFG constraints

Experiment Results

Kaluza benchmark result

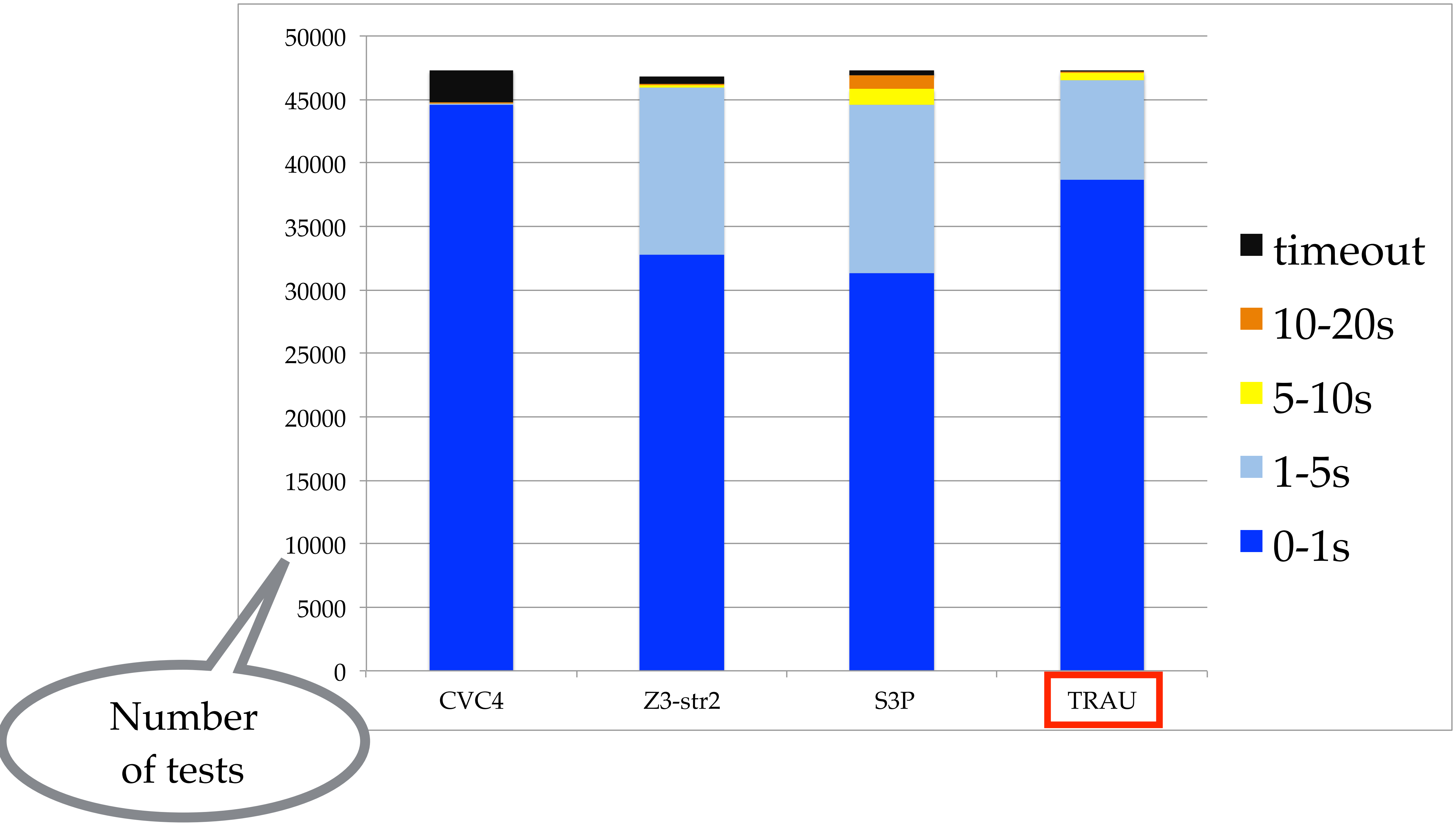
	CVC4	Z3-str2	S3P	TRAU
sat	33191	34459	34829	35202
unsat	11625	11747	12033	12019
timeout	2468	553	422	63

timeout 20s

No. finished tests

Experiment Results

Kaluza benchmark result



Experiment Results

SQL Injection result

length bound for vars

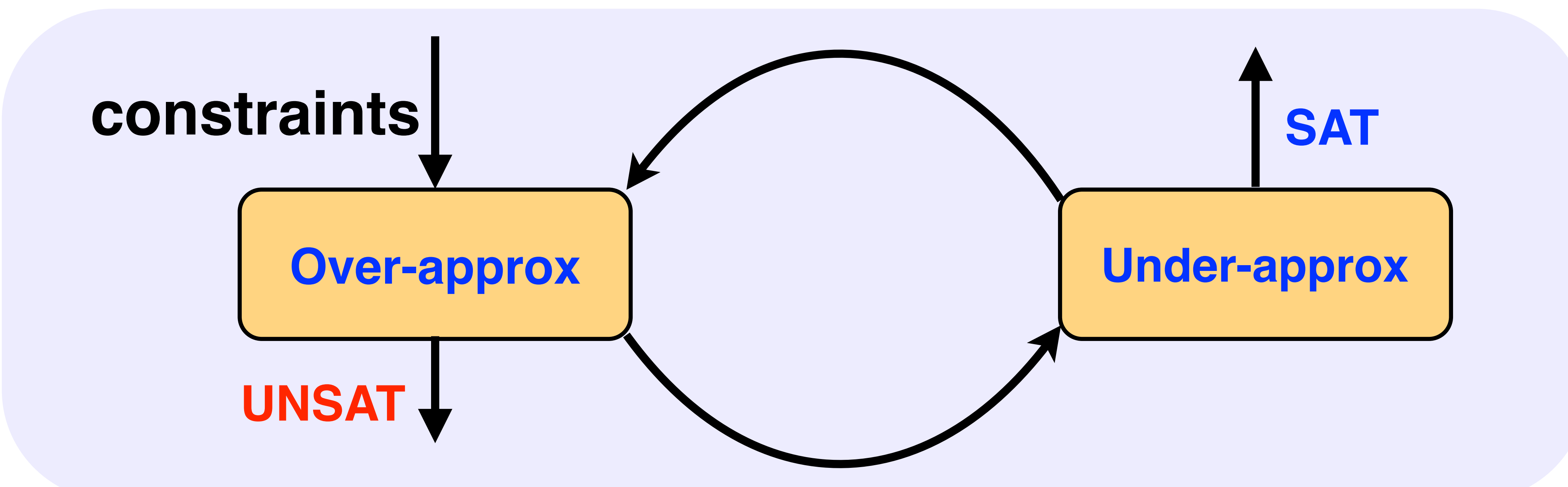
Input	Var	Length	TRAU Bounded Length Result	TRAU Bounded Length Time(s)	TRAU Unbouned Length Result	TRAU Unbouned Length Times(s)	HAMPI Bounded Length Result	HAMPI Bounded Length Times(s)
cfg01	6	20	sat	1.14	sat	1.24	sat	0.52
cfg02	6	20	unsat	1.02	unsat	1.11	unsat	0.20
cfg03	8	50	sat	1.01	sat	1.45	sat	9.34
cfg04	8	50	unsat	1.56	unsat	1.54	unsat	9.33
cfg05	10	70	sat	1.55	sat	2.00	-	timeout
cfg06	10	70	unsat	2.01	unsat	1.12	-	timeout
cfg07	14	50	sat	2.13	sat	3.36	-	timeout
cfg08	14	50	unsat	1.56	unsat	2.58	unsat	8.85
cfg09	20	70	sat	1.78	sat	2.27	-	timeout
cfg10	20	70	unsat	2.46	unsat	1.89	-	timeout

20s

Summary

1. New framework for solving string constraints:

   - Handle rich class of constraints: CFG membership, transducer, etc.
   - Based on Counter-Example Guided Abstract Refinement.



2. Open-source tool: outperforming all existing tools.

Thank you!