From SAT To SMT: Part 2

Vijay Ganesh
MIT

Talk Outline

Topics covered in Lecture 1

- ☑ **Motivation for SAT/SMT solvers in software engineering**
  - Software engineering (SE) problems reduced to logic problems
  - Automation, engineering, usability of SE tools through solvers
- ☑ **High-level description of the SAT/SMT problem & logics**
  - Rich logics close to program semantics
  - Demonstrably easy to solve in many practical cases
- ☑ **Modern SAT solver architecture & techniques**
  - DPLL search, shortcomings
  - Modern CDCL SAT solver: propagate (BCP), decide (VSIDS), conflict analysis, clause learn, backJump,
  - Termination, correctness
  - Big lesson: learning from mistakes

Topics covered in Lecture 2

- **Modern SMT solver architecture & techniques**
  - Rich logics closer to program semantics
  - DPLL(T), Combinations of solvers, Over/under approximations
- **My own contributions: STP & HAMPI**
  - Abstraction-refinement for solving
  - Bounded logics
- **SAT/SMT-based applications**
- **Future of SAT/SMT solvers**

Modern SMT Solvers
Are SAT Solvers Enough?

What is SMT

- Satisfiability Modulo Theories. Just a fancy name for a mathematical theory

Motivations: why we need SMT?

- A satisfiability solver for rich logics/natural theories
- Easier to encode program semantics in these theories
- Easier to exploit rich logic structure, greater opportunity for optimizations

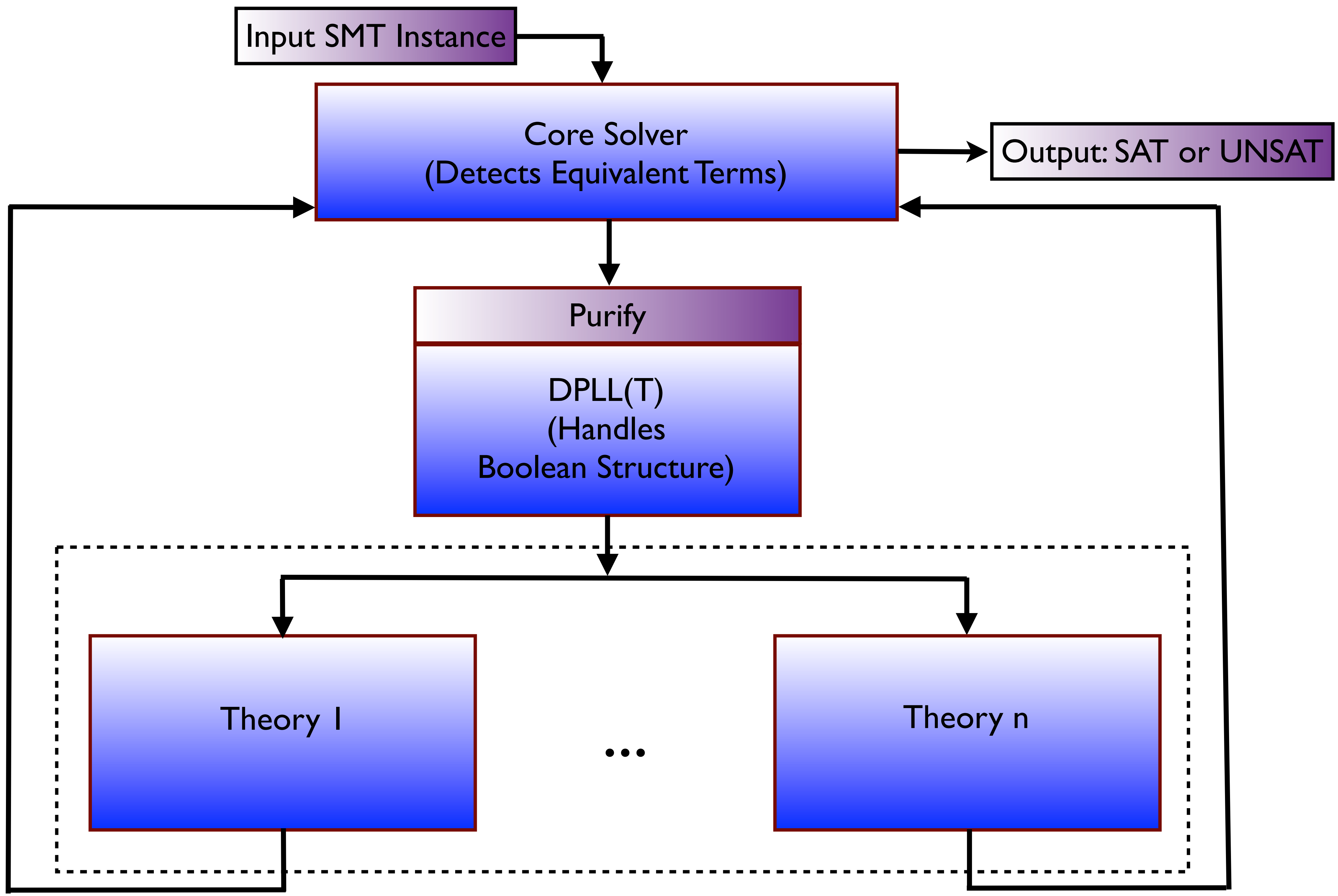
SMT Logics

- Bit-vectors, arrays, functions, linear integer/real arithmetic, strings, non-linear arithmetic
- Datatypes, quantifiers, non-linear arithmetic, floating point
- Extensible, programmable

SAT & SMT is an explosive combo: incredible impact

Standard-issue SMT Solver Architecture

Combination of theories & DPLL(T)



Standard-issue SMT Solver Architecture

Combination of theories: Nelson-Oppen

Problem Statement

- Combine theory solvers to obtain a solver for a union theory

Motivation

- Software engineering constraints over many natural theories
- Natural theories well understood
- Modularity

How

- Setup communication between individual theory solvers
- Communication over shared signature
- Soundness, completeness and termination

Standard-issue SMT Solver Architecture

Combination of theories: Nelson-Oppen

Example Constraint over Linear Reals (R) and Uninterpreted Functions (UF)

$$f(f(x) - f(y)) = a$$
$$f(0) = a+2$$
$$x = y$$

IDEA: $\Phi_{comb} \Leftrightarrow (\Phi_{T1} \wedge EQ) \wedge (\Phi_{T2} \wedge EQ)$

- First Step: purify each literal so that it belongs to a single theory
- Second Step: check satisfiability and exchange entailed equalities over shared vars (EQ)
- The solvers have to agree on equalities/disequalities between shared vars

UF	**R**
$f(e_1) = a$	$e_2 - e_3 = e_1$
$f(x) = e_2$	$e_4 = 0$
$f(y) = e_3$	$e_5 = a + 2$
$f(e_4) = e_5$	
$x = y$	

Standard-issue SMT Solver Architecture

Combination of theories: Nelson-Oppen

Example Constraint over Linear Reals (R) and Uninterpreted Functions (UF)

$$f(f(x) - f(y)) = a$$
$$f(0) = a+2$$
$$x = y$$

IDEA: $\Phi_{comb} \Leftrightarrow (\Phi_{T1} \wedge EQ) \wedge (\Phi_{T2} \wedge EQ)$

- First Step: purify each literal so that it belongs to a single theory
- Second Step: check satisfiability and exchange entailed equalities over shared vars (EQ)
- The solvers have to agree on equalities/disequalities between shared vars

UF	**R**
$f(e_1) = a$	$e_2 - e_3 = e_1$
$f(x) = e_2$	$e_4 = 0$
$f(y) = e_3$	$e_5 = a + 2$
$f(e_4) = e_5$	$e_2 = e_3$
$x = y$	

Standard-issue SMT Solver Architecture

Combination of theories: Nelson-Oppen

Example Constraint over Linear Reals (R) and Uninterpreted Functions (UF)

$$f(f(x) - f(y)) = a$$
$$f(0) = a+2$$
$$x = y$$

IDEA: $\Phi_{comb} \Leftrightarrow (\Phi_{T1} \wedge EQ) \wedge (\Phi_{T2} \wedge EQ)$

- First Step: purify each literal so that it belongs to a single theory
- Second Step: check satisfiability and exchange entailed equalities over shared vars (EQ)
- The solvers have to agree on equalities/disequalities between shared vars

UF	**R**
$f(e_1) = a$	$e_2 - e_3 = e_1$
$f(x) = e_2$	$e_4 = 0$
$f(y) = e_3$	$e_5 = a + 2$
$f(e_4) = e_5$	$e_2 = e_3$
$x = y$	
$e_1 = e_4$	

Standard-issue SMT Solver Architecture

Combination of theories: Nelson-Oppen

Example Constraint over Linear Reals (R) and Uninterpreted Functions (UF)

$$f(f(x) - f(y)) = a$$
$$f(0) = a+2$$
$$x = y$$

IDEA: $\Phi_{comb} \Leftrightarrow (\Phi_{T1} \wedge EQ) \wedge (\Phi_{T2} \wedge EQ)$

- First Step: purify each literal so that it belongs to a single theory
- Second Step: check satisfiability and exchange entailed equalities over shared vars (EQ)
- The solvers have to agree on equalities/disequalities between shared vars
- UF says SAT, R says UNSAT. Combination returns UNSAT.

UF	**R**
$f(e_1) = a$	$e_2 - e_3 = e_1$
$f(x) = e_2$	$e_4 = 0$
$f(y) = e_3$	$e_5 = a + 2$
$f(e_4) = e_5$	$e_2 = e_3$
$x = y$	$e_5 = a$
$e_1 = e_4$	

Standard-issue SMT Solver Architecture

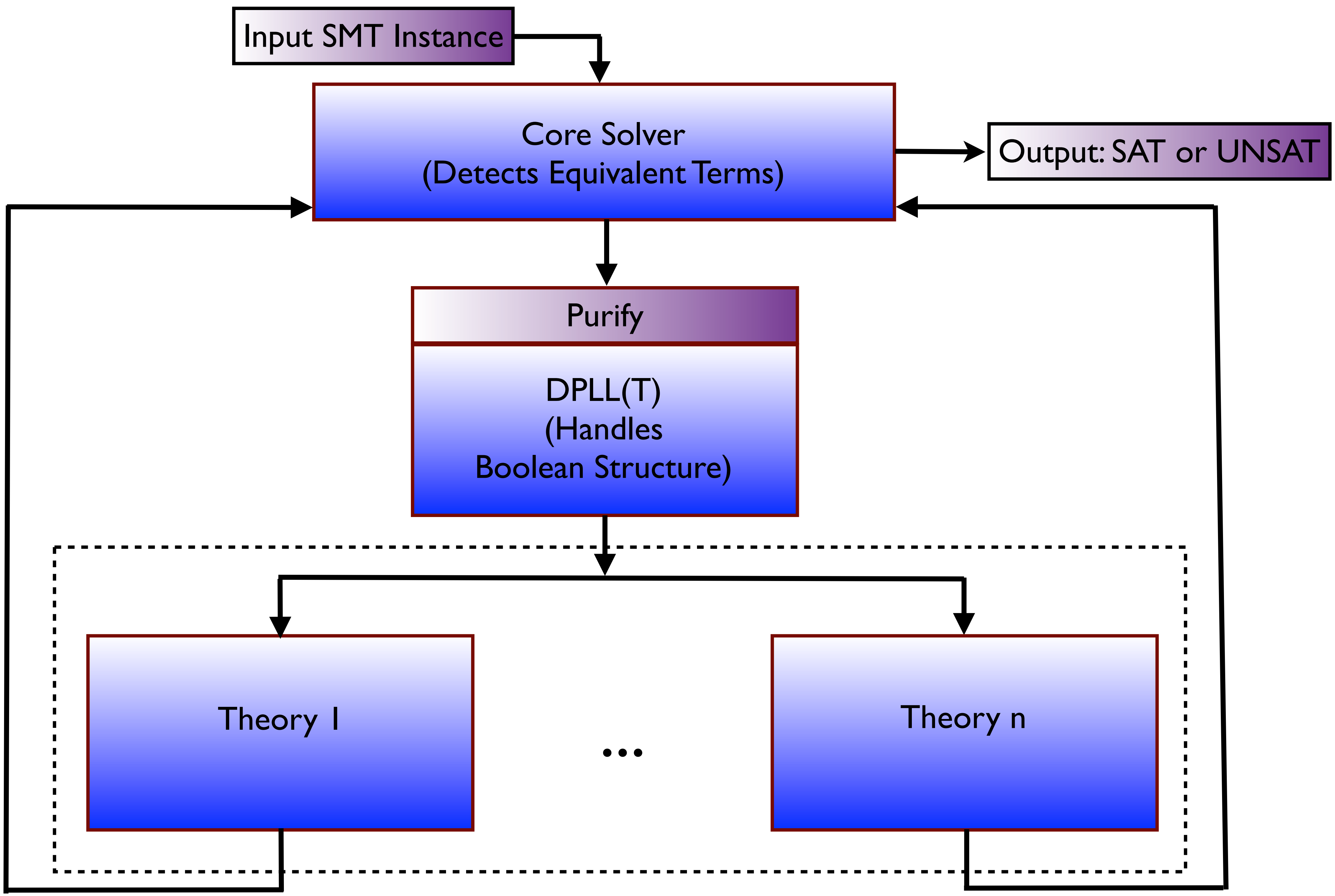
Combination of theories: Nelson-Oppen

IDEA: $\Phi_{comb} \Leftrightarrow (\Phi_{T1} \wedge EQ) \wedge (\Phi_{T2} \wedge EQ)$

- Does NOT always work, i.e., does not always give a complete solver
- Example: Cannot combine T_1 with only finite models, and T_2 with infinite models
- Impose conditions on T_1 and T_2
  - Stably Infinite: If a T-formula has a model it has an infinite model
  - Examples: Functions, Arithmetic
  - Extensions proved to be artificial or difficult
  - Deep model-theoretic implications (Ghilardi 2006, G. 2007)

Standard-issue SMT Solver Architecture

Combination of theories & DPLL(T)



Standard-issue SMT Solver Architecture
DPLL(T)

Problem Statement

- Efficiently handle the Boolean structure of the input formula

Basic Idea

- Use a SAT solver for the Boolean structure & check assignment consistency against a T-solver
- T-solver only supports conjunction of T-literals

Improvements

- Check partial assignments against T-solver
- Do theory propagation (similar to SAT solvers)
- Conflict analysis guided by T-solver & generate conflict clauses (similar to SAT solvers)
- BackJump (similar to SAT solvers)

Standard-issue SMT Solver Architecture
DPLL(T)

Uninterpreted Functions formula

(1) $(g(a) = c) \wedge$
($\neg 2 \vee 3$) $(f(g(a)) \neq f(c) \vee (g(a) = d)) \wedge$
($\neg 4$) $(c \neq d)$

Theory and Unit Propagation Steps by DPLL(T)

(Unit Propagate) (1)
(Unit Propagate) ($\neg 4$)
(Theory Propagate) (2)
(Theory Propagate) (3)
UNSAT

History of SMT Solvers

Category	Research Project	Researcher/Institution/Time Period
Theorem Proving (very early roots of decision procedures)	NuPRL Boyer-Moore Theorem Prover ACL2 PVS Proof Checker	Robert Constable / Cornell / 1970's-present Boyer & Moore / UT Austin / 1970's-present Moore, Kauffmann et al. / UT Austin / 1980's - present Natarajan Shankar / SRI International / 1990's-present
SAT Solvers	DPLL GRASP (Clause learning and backjumping) Chaff & zChaff MiniSAT	Davis, Putnam, Logemann & Loveland / 1962 Marques-Silva & Sakallah / U. Michigan / 1996-2000 Zhang, Malik et al. / Princeton / 1997-2002 Een & Sorensson / 2005 - present
Combinations	Simplify Shostak ICS SVC, CVC, CVC-Lite, CVC3 ... Non-disjoint theories	Nelson & Oppen / DEC and Compaq / late 1980s Shostak / SRI International / late 1980's Ruess & Shankar / SRI International / late 1990's Barrett & Dill / Stanford U. / late 1990's Tinelli, Ghilardi,..., / 2000 - 2008
DPLL(T)	Barcelogic and Tinelli group	Oliveras, Nieuwenhuis & Tinelli / UPC and Iowa / 2006
Under/Over Approximations	UCLID STP	Seshia & Bryant / CMU / 2004 - present Ganesh & Dill / Stanford / 2005 - present
Widely-used SMT Solvers	Z3 CVC4 OpenSMT Yices MathSAT STP UCLID	DeMoura & Bjorner / Microsoft / 2006 - present Barrett & Tinelli / NYU and Iowa / early 2000's - present Bruttomesso / USI Lugano / 2008 - present Deuterre / SRI International / 2005 - present Cimatti et al. / Trento / 2005 - present Ganesh / Stanford & MIT / 2005 - present Seshia / CMU & Berkeley / 2004 - present

Talk Outline

Topics covered in Lecture 1

☑ **Motivation for SAT/SMT solvers in software engineering**
- Software engineering (SE) problems reduced to logic problems
- Automation, engineering, usability of SE tools through solvers

☑ **High-level description of the SAT/SMT problem & logics**
- Rich logics close to program semantics
- Demonstrably easy to solve in many practical cases

☑ **Modern SAT solver architecture & techniques**
- DPLL search, shortcomings
- Modern CDCL SAT solver: propagate (BCP), decide (VSIDS), conflict analysis, clause learn, backJump,
- Termination, correctness
- Big lesson: learning from mistakes

Topics covered in Lecture 2

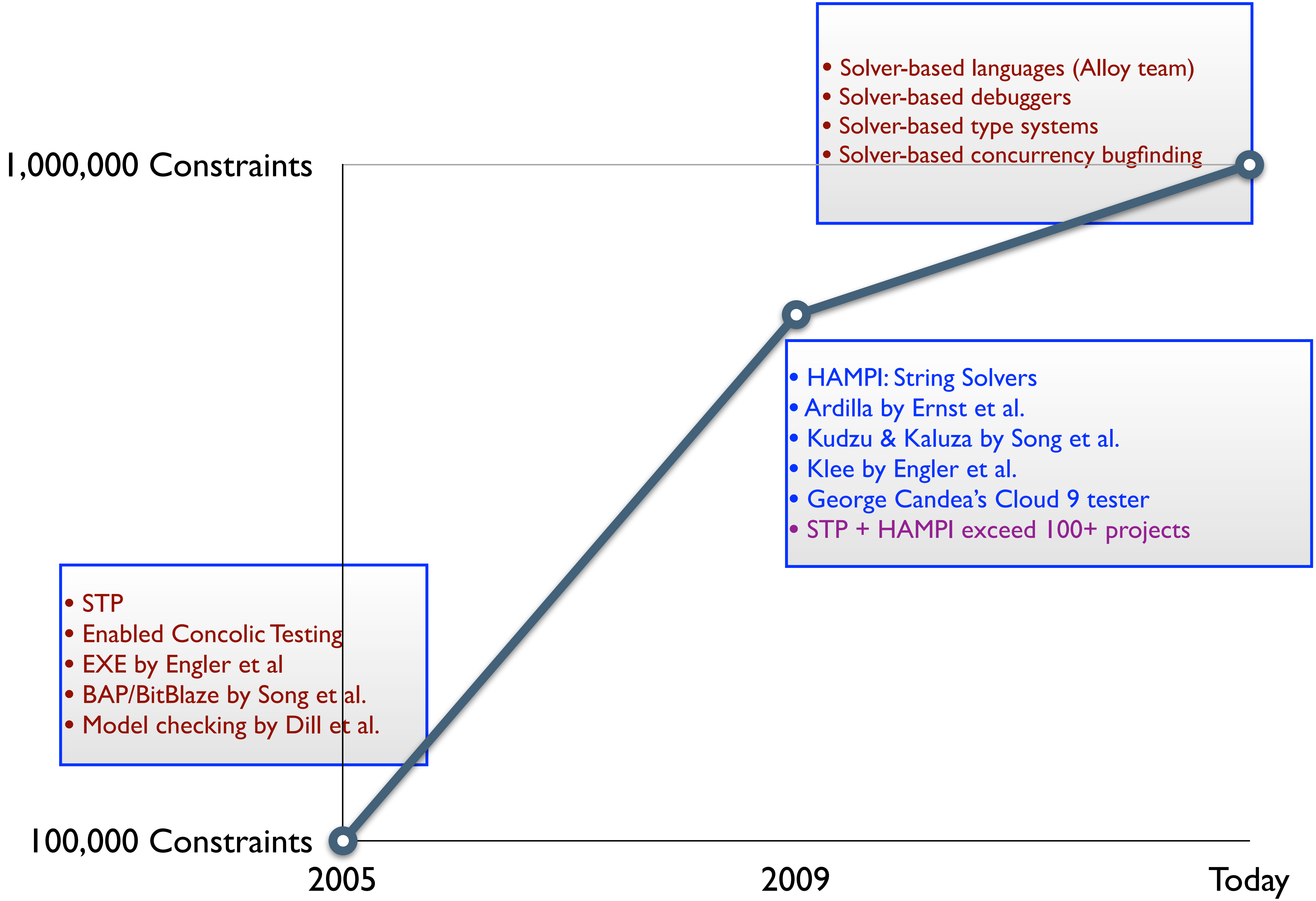
☑ **Modern SMT solver architecture & techniques**
- Rich logics closer to program semantics
- DPLL(T), Combinations of solvers, Over/under approximations

- **My own contributions: STP & HAMPI**
  - STP: Abstraction-refinement for solving
  - Applications to dynamic symbolic testing (aka concolic testing)
  - HAMPI: Bounded logics
- **SAT/SMT-based applications**
- **Future of SAT/SMT solvers**

STP Bit-vector & Array Solver

- Bit-vector or machine arithmetic
- Arrays for memory
- C/C++/Java expressions
- NP-complete

The History of STP



Programs Reasoning & STP
Why Bit-vectors and Arrays

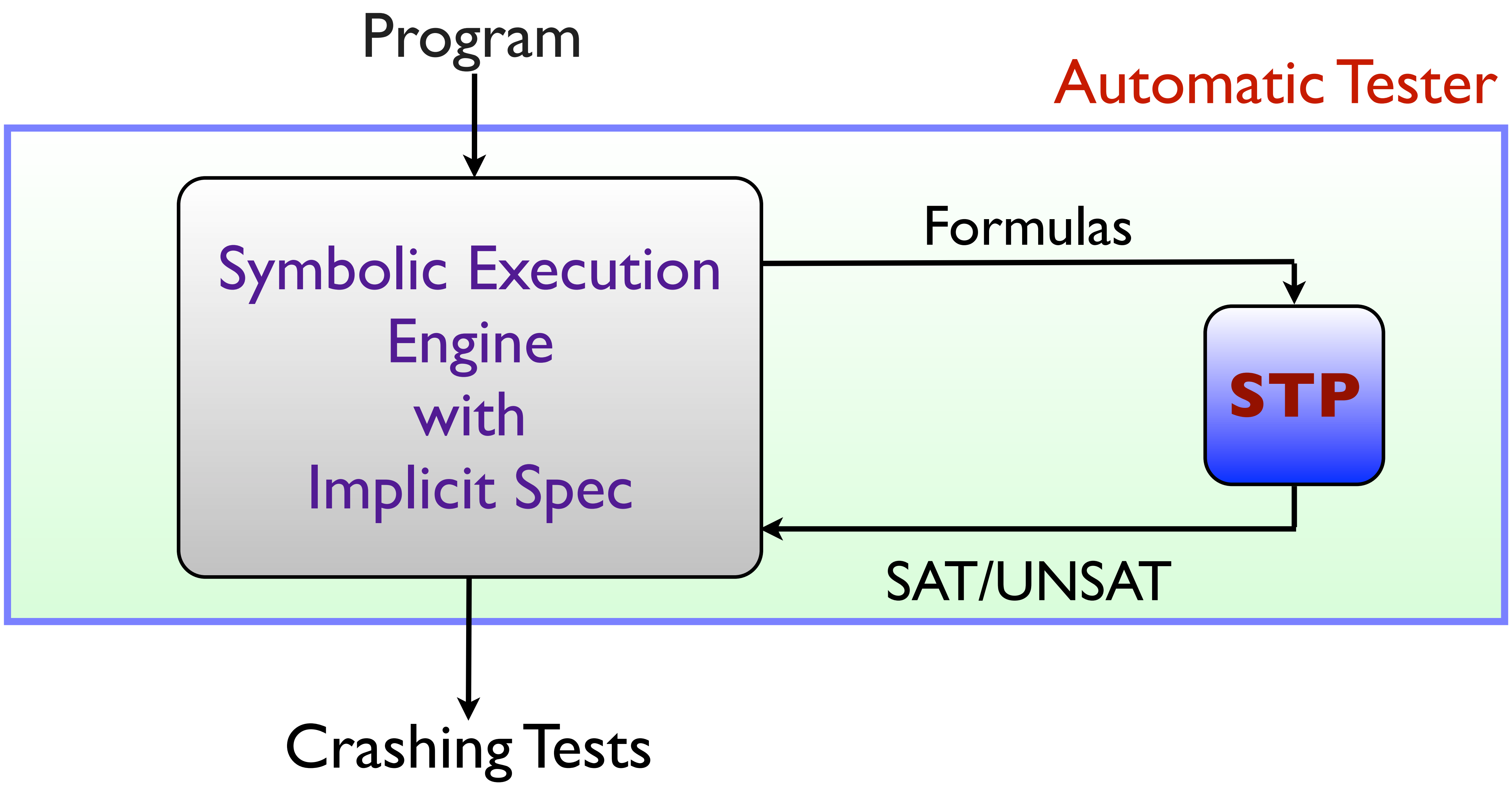
- STP logic tailored for software reliability applications
- Support symbolic execution/program analysis

C/C++/Java/...	Bit-vectors and Arrays
Int Var Char Var	32 bit variable 8 bit variable
Arithmetic operation (x+y, x-y, x*y, x/y,...)	Arithmetic function (x+y,x-y,x*y,x/y,...)
assignments x = expr;	equality x = expr;
if conditional if(cond) x = $expr^1$ else x = $expr^2$	if-then-else construct x = if(cond) $expr^1$ else $expr^2$
inequality	inequality predicate
Memory read/write x = *ptr + i;	Array read/write ptr[]; x = Read(ptr,i);
Structure/Class	Serialized bit-vector expressions
Function	Symbolic execution
Loops	Bounding

How to Automatically Crash Programs?

Concolic Execution & STP

Problem: Automatically generate crashing tests given only the code



How to Automate Testing?

Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player,...

```
Buggy_C_Program(int* data_field, int len_field) {

    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
      //1. Integer overflow causing NULL deref
      //2. Buffer overflow
      *(ptr+i) = process_data(*(data_field+i));
    }
}
```

- Formula captures computation
- Tester attaches formula to capture spec

How to Automate Testing?
Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player,...

```
Buggy_C_Program(int* data_field, int len_field) {

    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
      //1. Integer overflow causing NULL deref
      //2. Buffer overflow
      *(ptr+i) = process_data(*(data_field+i));
    }
}
```

Equivalent Logic Formula derived using symbolic execution

```
data_field, mem_ptr :  ARRAY;
len_field : BITVECTOR(32); //symbolic
i,  j, ptr    : BITVECTOR(32);//symbolic
.
.
mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);
.
.
```

- Formula captures computation
- Tester attaches formula to capture spec

How to Automate Testing?

Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player,...

```
Buggy_C_Program(int* data_field, int len_field) {

    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
      //1. Integer overflow causing NULL deref
      //2. Buffer overflow
      *(ptr+i) = process_data(*(data_field+i));
    }
}
```

↔

Equivalent Logic Formula derived using symbolic execution

```
data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr   : BITVECTOR(32);//symbolic
.
.
mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);
.
.
```

- Formula captures computation
- Tester attaches formula to capture spec

How to Automate Testing?

Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player,...

```
Buggy_C_Program(int* data_field, int len_field) {

    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
      //1. Integer overflow causing NULL deref
      //2. Buffer overflow
      *(ptr+i) = process_data(*(data_field+i));
    }
}
```

↔

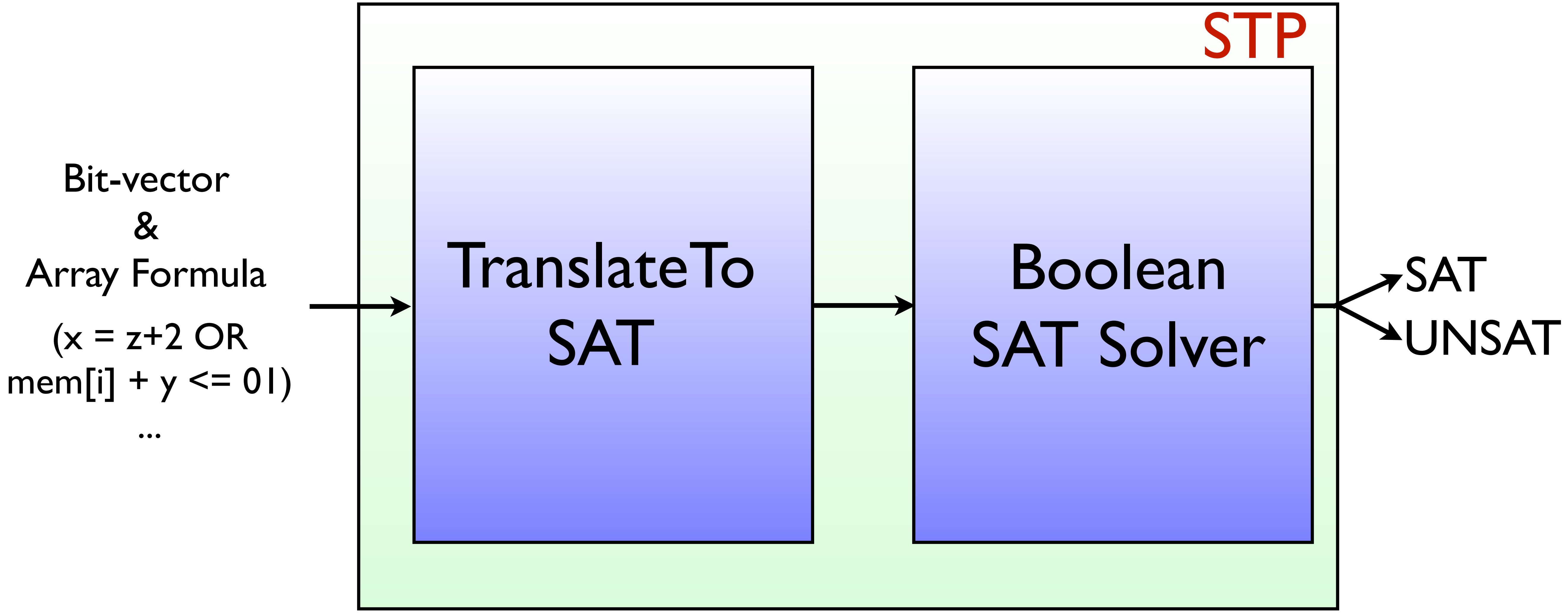
Equivalent Logic Formula derived using symbolic execution

```
data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr   : BITVECTOR(32);//symbolic
.
.
mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);
.
.
//INTEGER OVERFLOW QUERY
0 <= j <= process(len_field);
ptr + i + j = 0?
```

- Formula captures computation
- Tester attaches formula to capture spec

How STP Works

Bird's Eye View: Translate to SAT

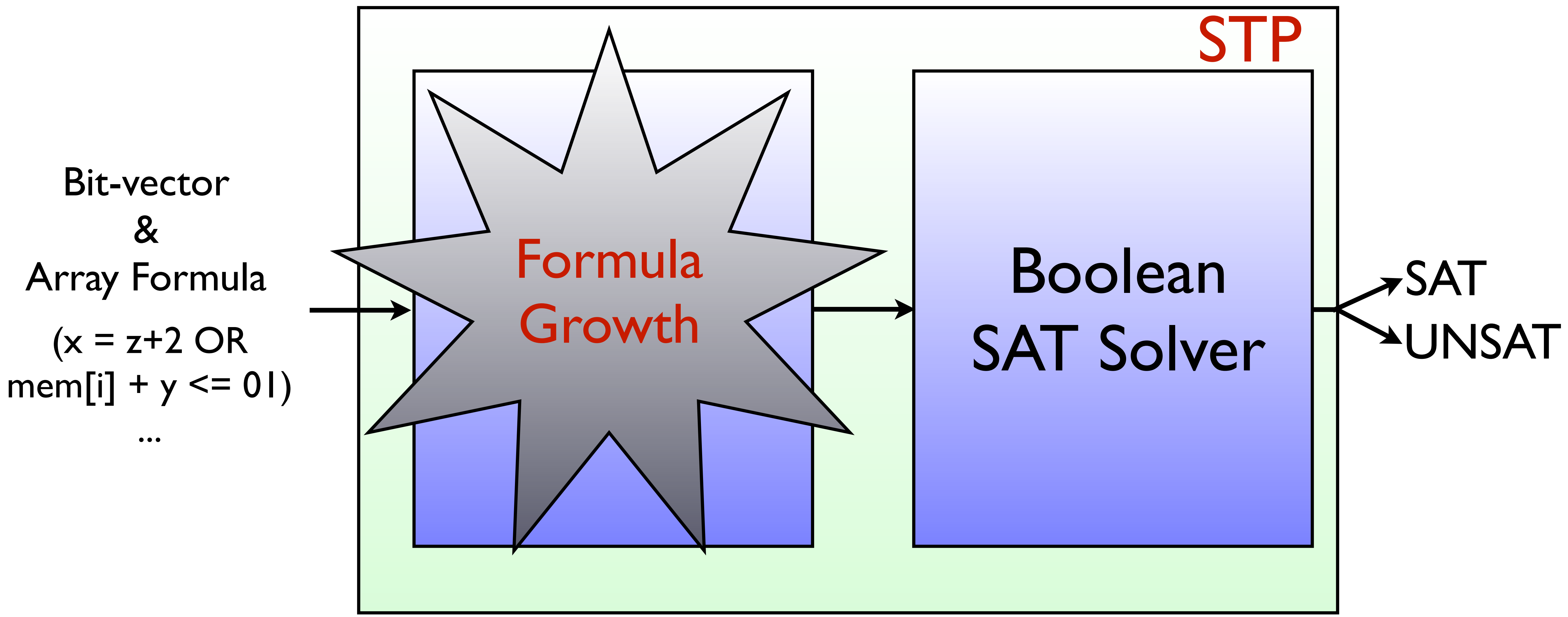


Why Translate to SAT?

- Both theories NP-complete
- Non SAT approaches didn't work
- Translation to SAT leverages solid engineering

How STP Works

Rich Theories cause MEM Blow-up



- Making information explicit
  - Space cost
  - Time cost

Explicit Information causes Blow-up

Array Memory Read Problem

Logic Formula derived using
symbolic execution

```
data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr    : BITVECTOR(32);//symbolic
.
.
mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);
.
.
if(ptr+i = ptr+j) then mem_ptr[ptr+i] = mem_ptr[ptr+j];

//INTEGER OVERFLOW QUERY
0 <= j <= process(len_field);
ptr + i + j < ptr?
```

- Array Aliasing is implicit
- Need to make information explicit during solving
- Cannot be avoided

How STP Works

Array-read MEM Blow-up Problem

- Problem: $O(n^2)$ axioms added, n is number of read indices
- Lethal, if n is large, say, n = 100,000; # of axioms is 10 Billion

Formula Growth

$Read(Mem,i_0) = expr_0$
$Read(Mem,i_1) = expr_1$
$Read(Mem,i_2) = expr_2$
.
.
.
$Read(Mem,i_n) = expr_n$

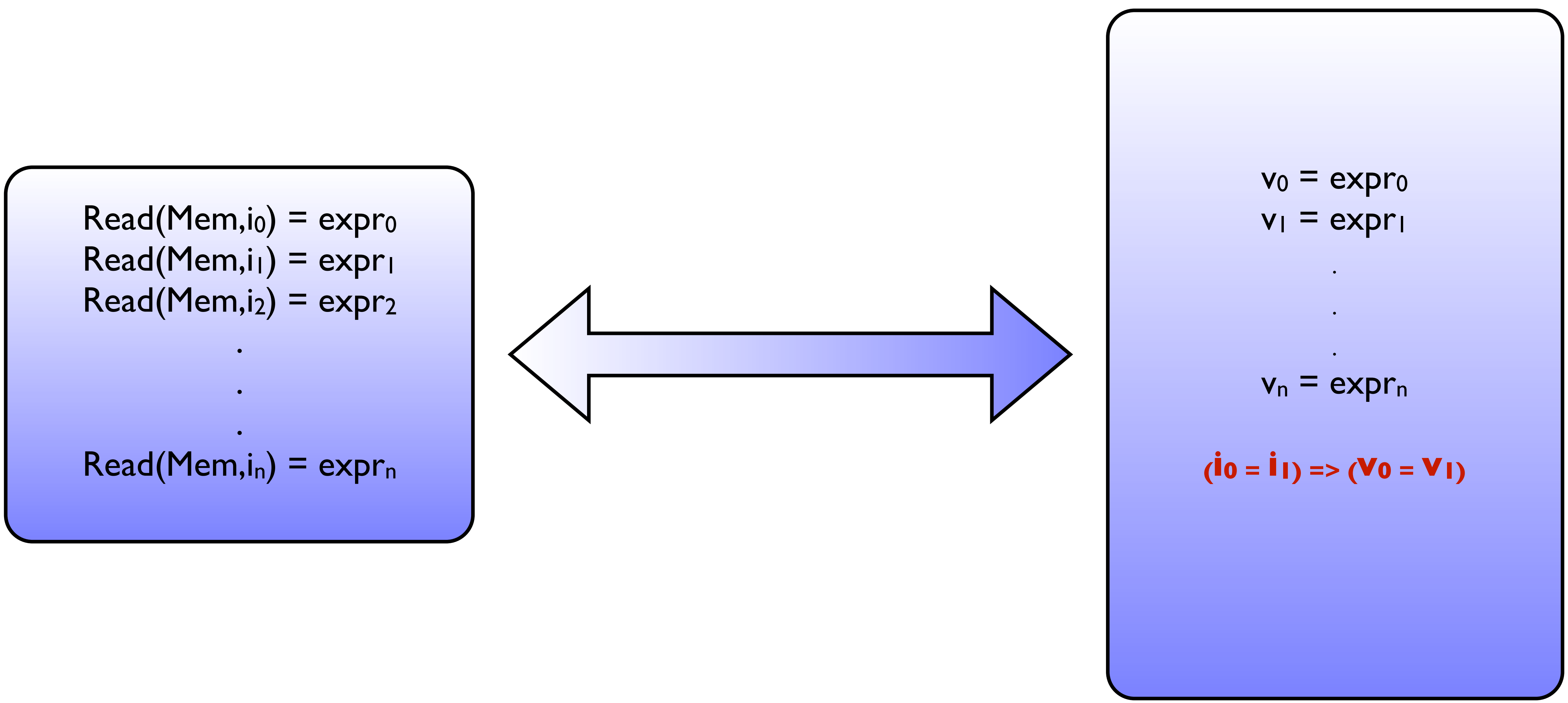
$v_0 = expr_0$
$v_1 = expr_1$
.
.
.
$v_n = expr_n$

$(i_0 = i_1) \Rightarrow (v_0 = v_1)$
$(i_0 = i_2) \Rightarrow (v_0 = v_2)$
...
$(i_1 = i_2) \Rightarrow (v_1 = v_2)$
...

How STP Works

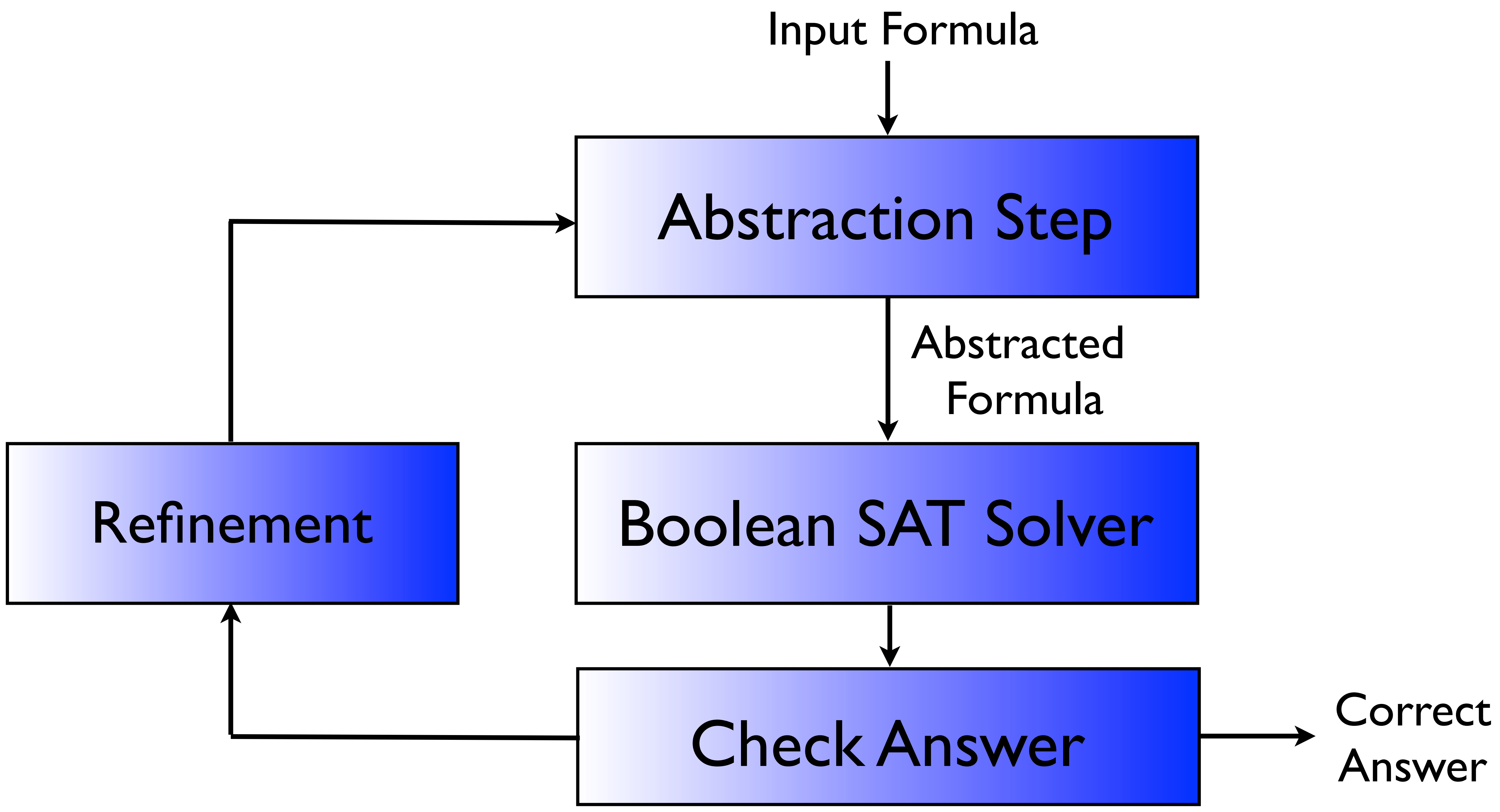
The Array-read Solution

- Key Observation
  - Most indices don’t alias in practice
  - Exploit locality of memory access in typical programs
  - Need only a fraction of array axioms for equivalence



STP Key Conceptual Contribution

Abstraction-refinement Principle



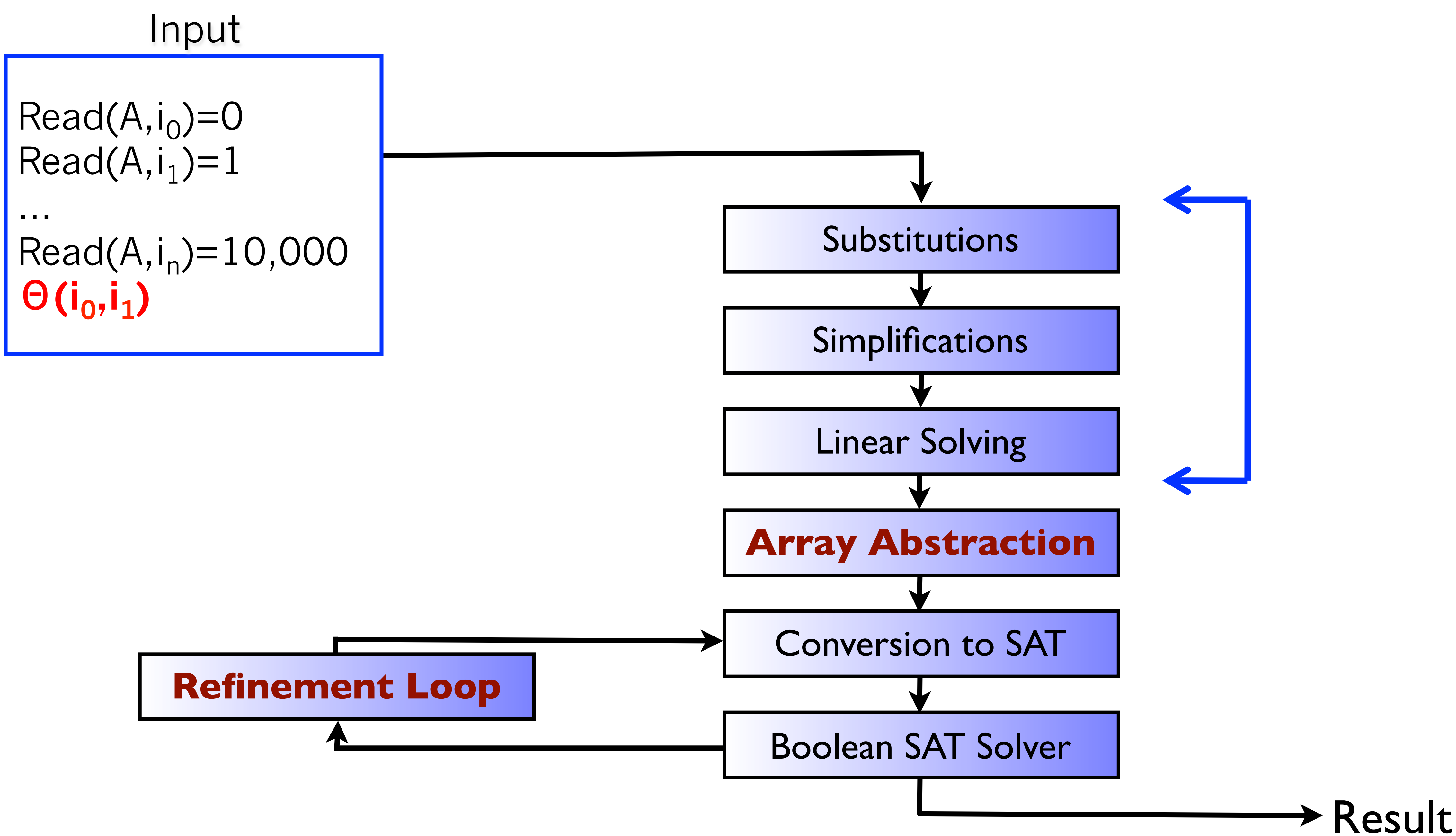
How STP Works

What to Abstract & How to Refine?

Abstraction	Refinement
1. Less essential parts 2. Causes MEM blow-up	1. Guided 2. Must remember
Abstraction manages formula growth hardness	Refinement manages search-space hardness

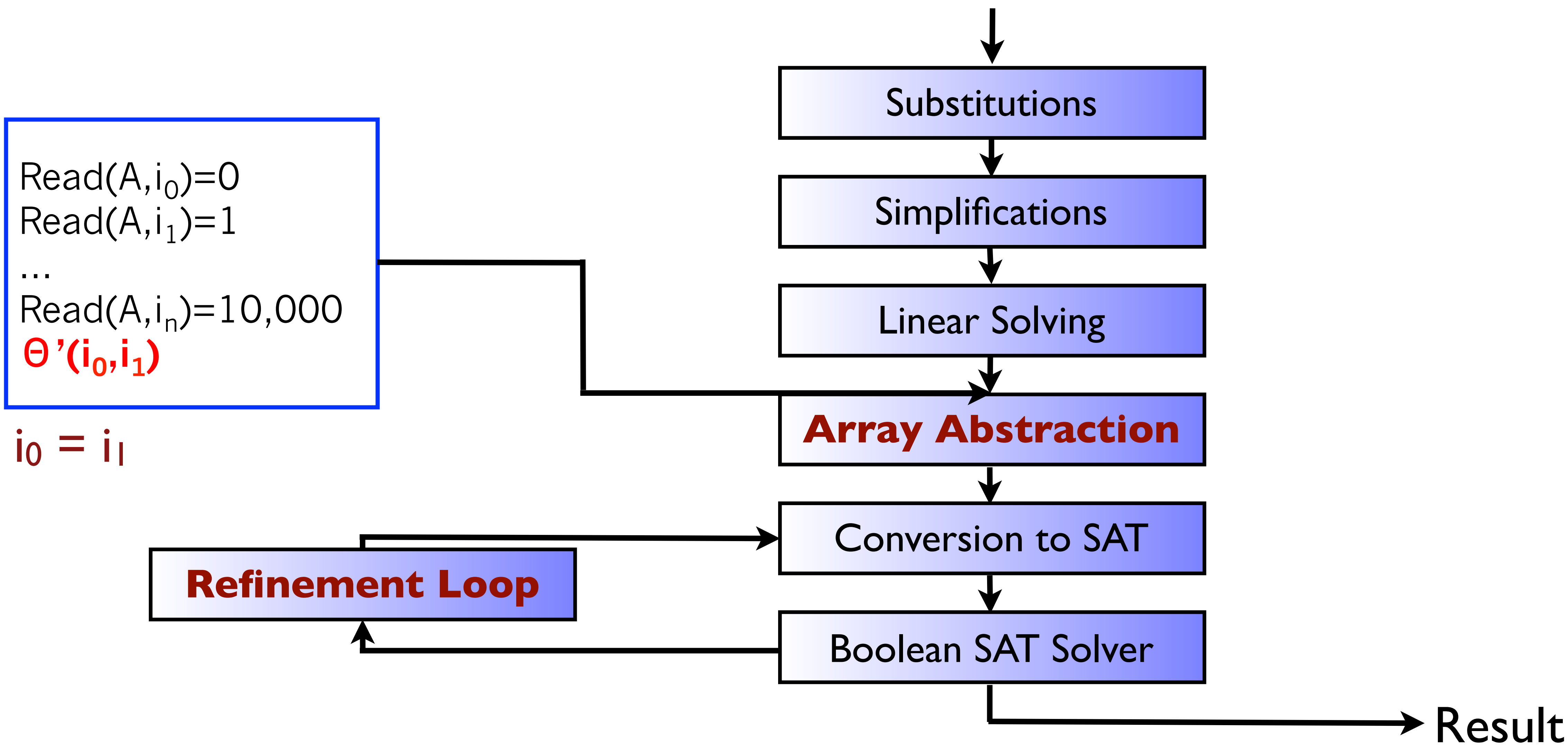
How STP Works

Abstraction-refinement for Array-reads



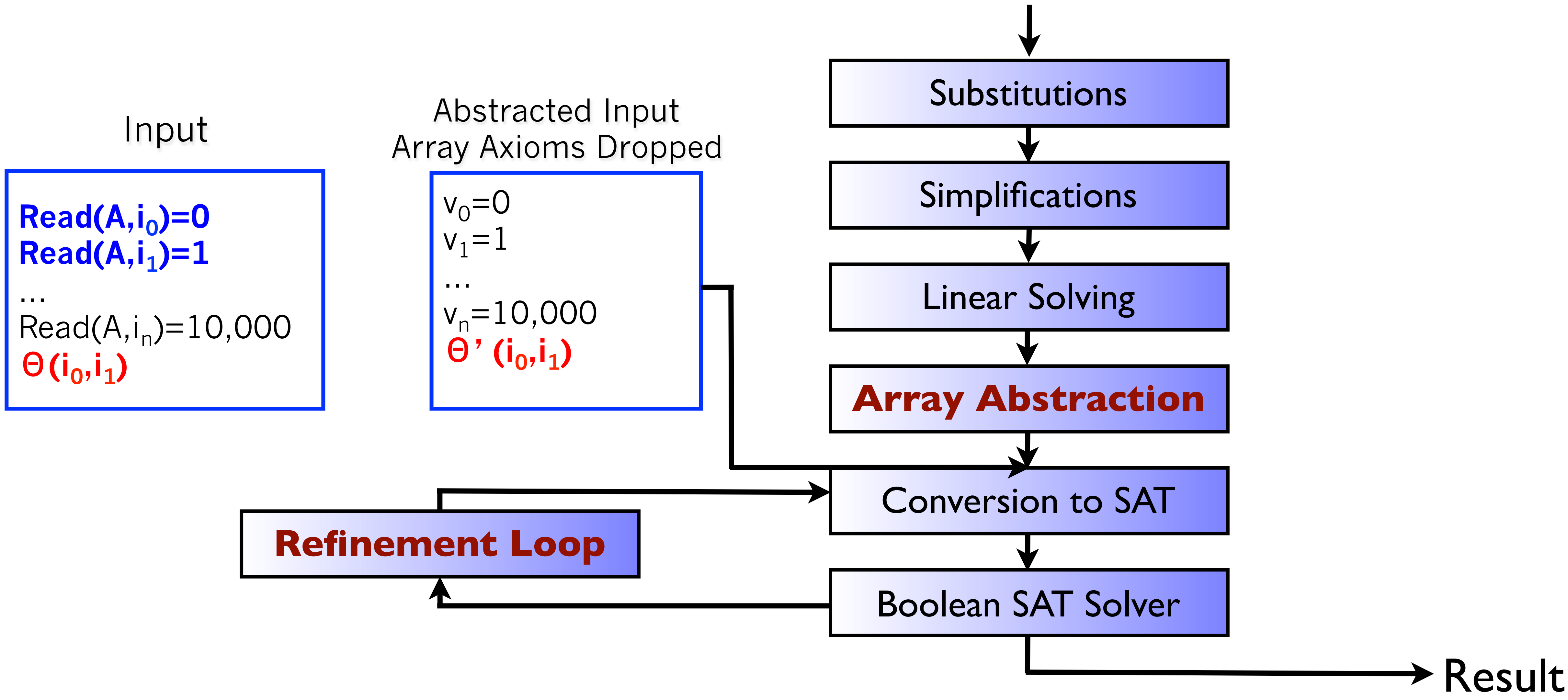
How STP Works

Abstraction-refinement for Array-reads



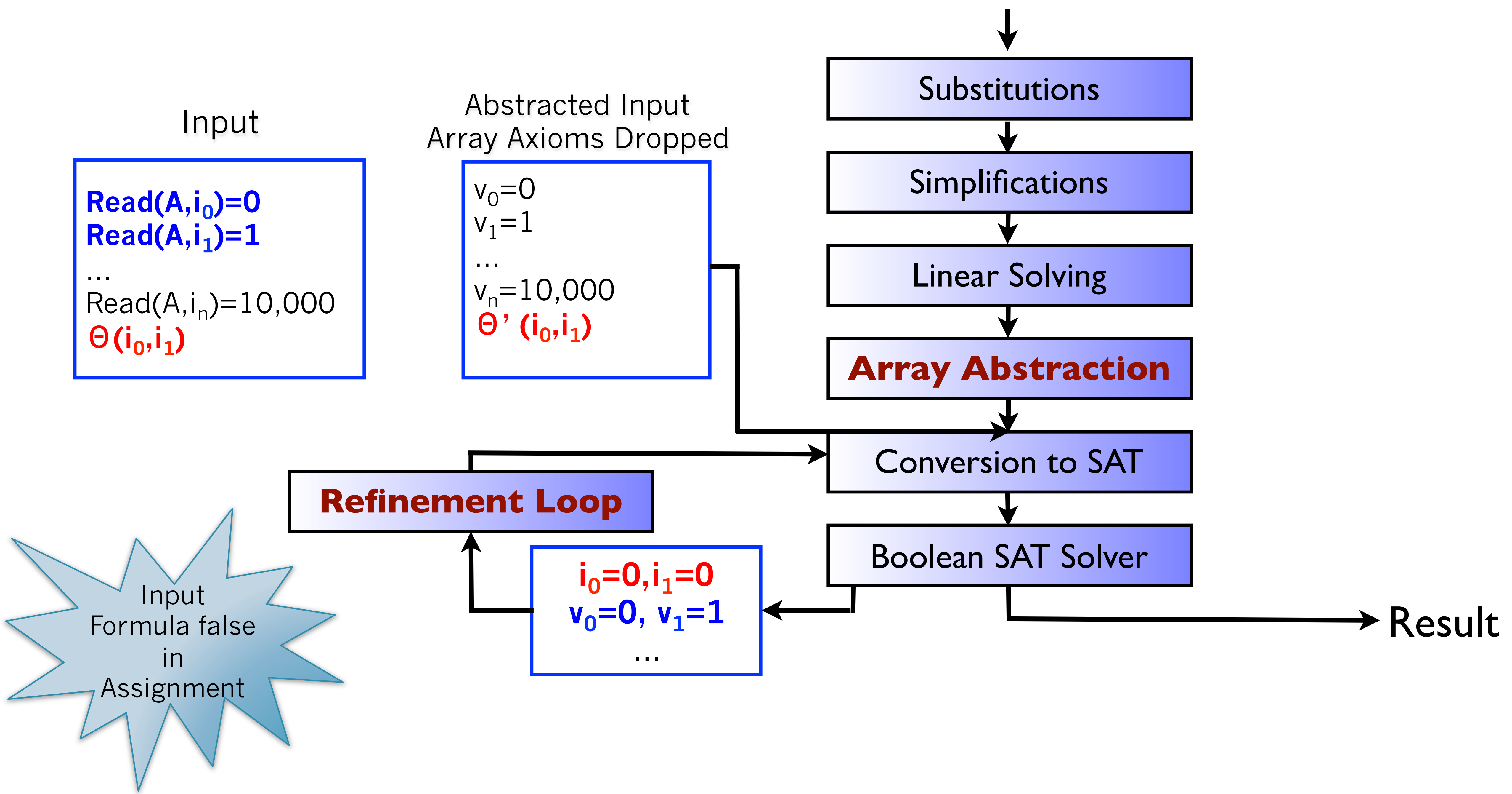
How STP Works

Abstraction-refinement for Array-reads



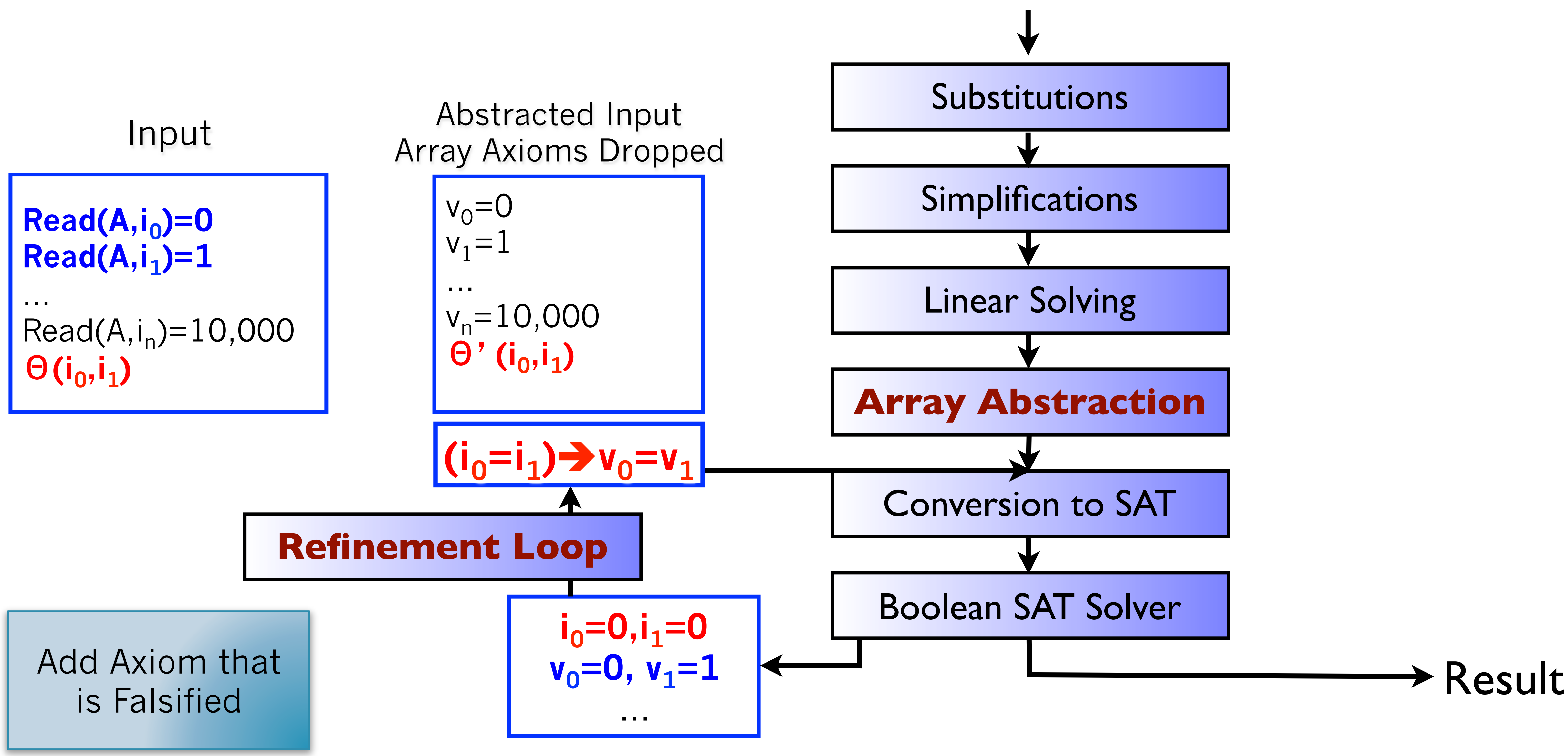
How STP Works

Abstraction-refinement for Array-reads



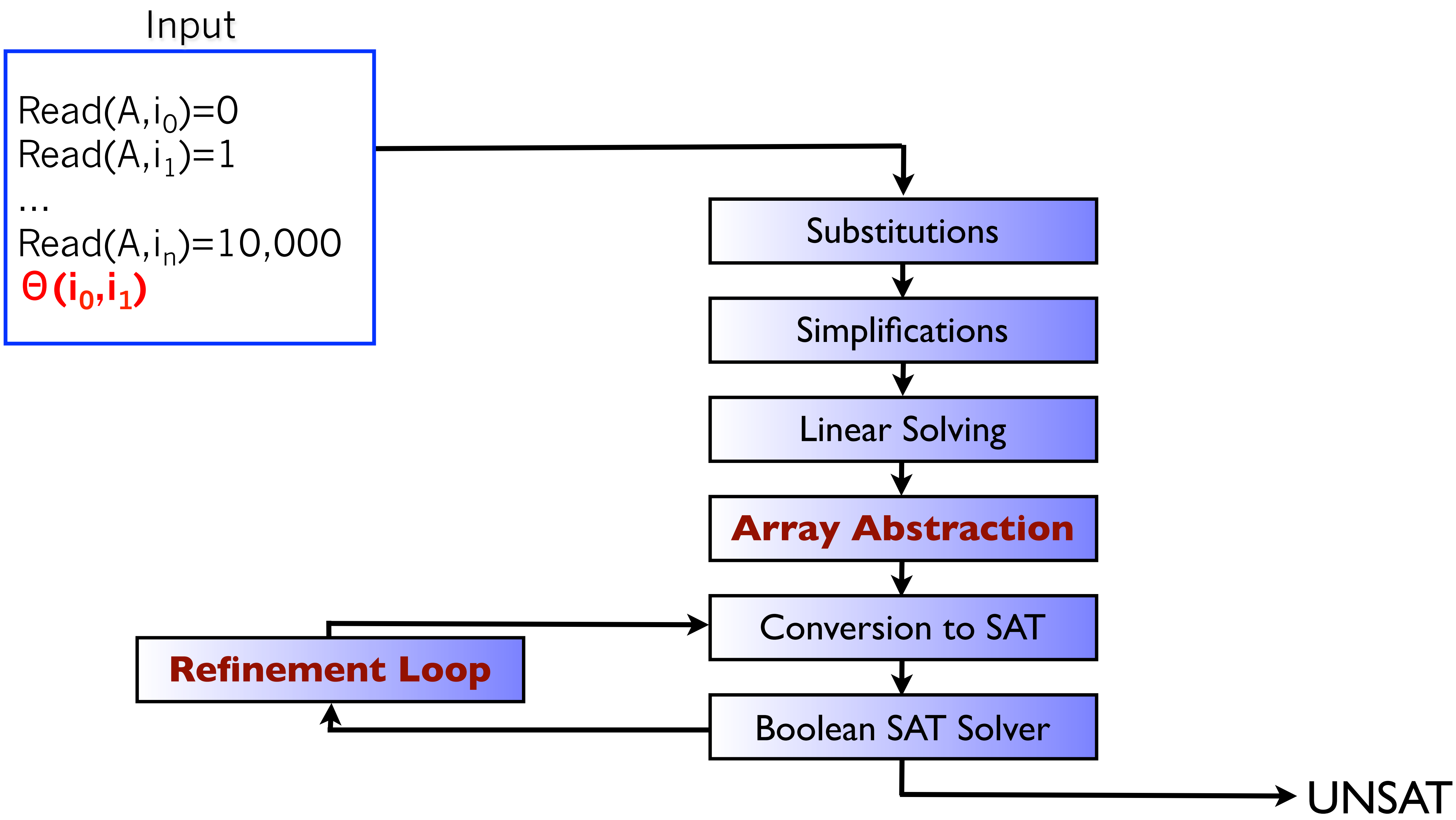
How STP Works

Abstraction-refinement for Array-reads



How STP Works

Abstraction-refinement for Array-reads



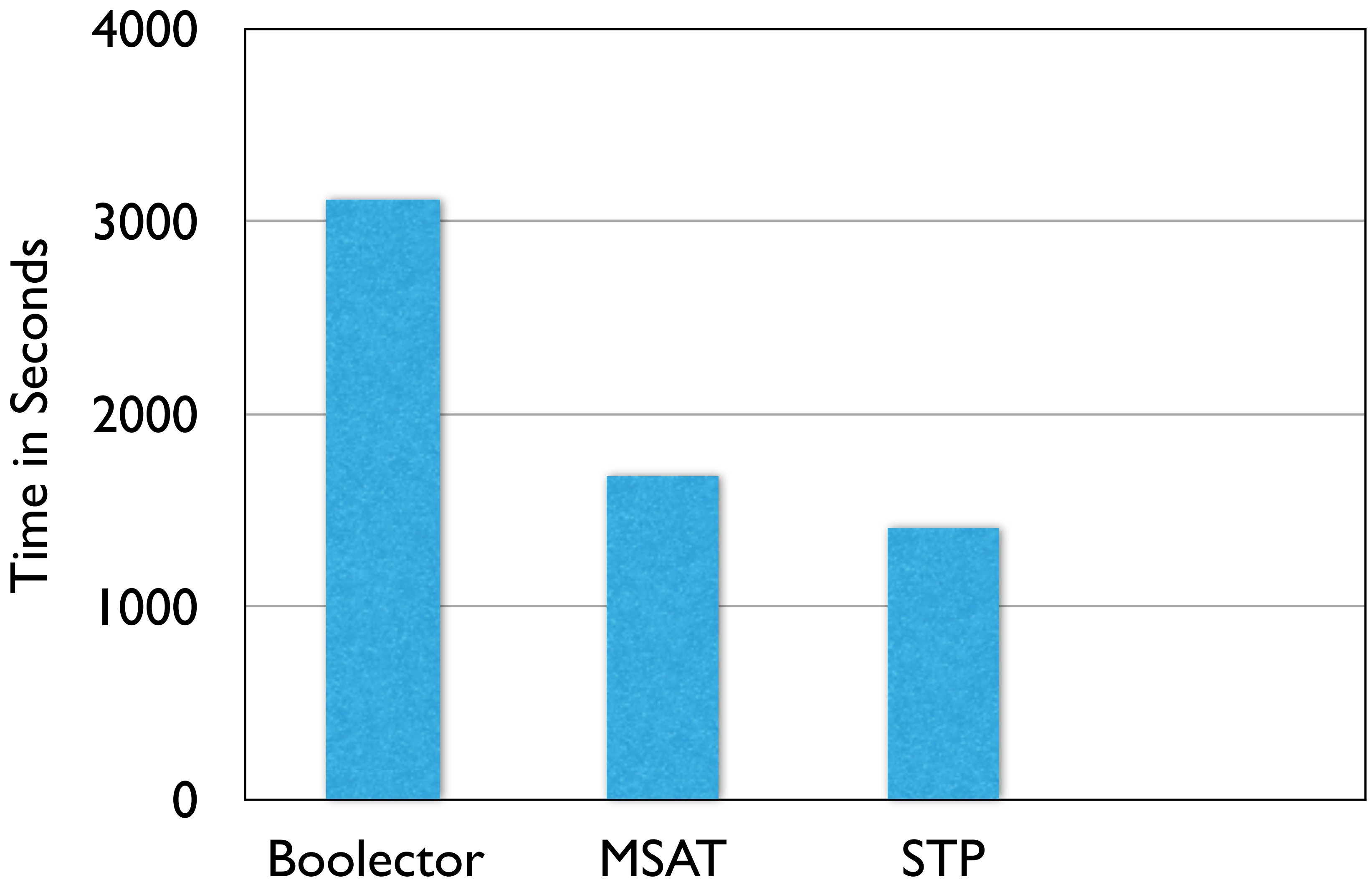
STP vs. Other Solvers

Testcase (Formula Size)	Result	Z3 (sec)	Yices (sec)	STP (sec)
610dd9c (~15K)	SAT	TimeOut	MemOut	37
Grep65 (~60K)	UNSAT	0.3	TimeOut	4
Grep84 (~69K)	SAT	176	TimeOut	18
Grep106 (~69K)	SAT	130	TimeOut	227
Blaster4 (~262K)	UNSAT	MemOut	MemOut	10
Testcase20 (~1.2M)	SAT	MemOut	MemOut	56
Testcase21 (~1.2M)	SAT	MemOut	MemOut	43

* All experiments on 3.2 GHz, 512 Kb cache
* MemOut: 3.2 GB (Memory used by STP much smaller), TimeOut: 1800 seconds
* Examples obtained from Dawn Song at Berkeley, David Molnar at Berkeley and Dawson Engler at Stanford
* Experiments conducted in 2007

STP vs. Other Leading Solvers

- STP vs. Boolector & MathSAT on 615 SMTCOMP 2007 - 2010 examples



* All experiments on 2.4 GHz, 1 GB RAM
* Timeout: 500 seconds/example

Impact of STP

- Enabled existing SE technologies to scale
  - Bounded model checkers, e.g., Chang and Dill

- Easier to engineer SE technologies
  - Formal tools (ACL2+STP) for verifying Crypto, Smith & Dill

- Enabled new SE technologies
  - Concolic testing (EXE,Klee,...) by Engler et al., Binary Analysis by Song et al.

Impact of STP: Notable Projects

- Enabled Concolic Testing
- 100+ reliability and security projects

Category	Research Project	Project Leader/Institution
Formal Methods	ACL2 Theorem Prover + STP Verification-aware Design Checker Java PathFinder Model Checker	Eric Smith & David Dill/Stanford Jacob Chang & David Dill/Stanford Mehlitz & Pasareanu/NASA
Program Analysis	BitBlaze & WebBlaze BAP	Dawn Song et al./Berkeley David Brumley/CMU
Automatic Testing Security	Klee, EXE SmartFuzz Kudzu	Engler & Cadar/Stanford Molnar & Wagner/Berkeley Saxena & Song/Berkeley
Hardware Bounded Model-cheking (BMC)	Blue-spec BMC BMC	Katelman & Dave/MIT Haimed/NVIDIA

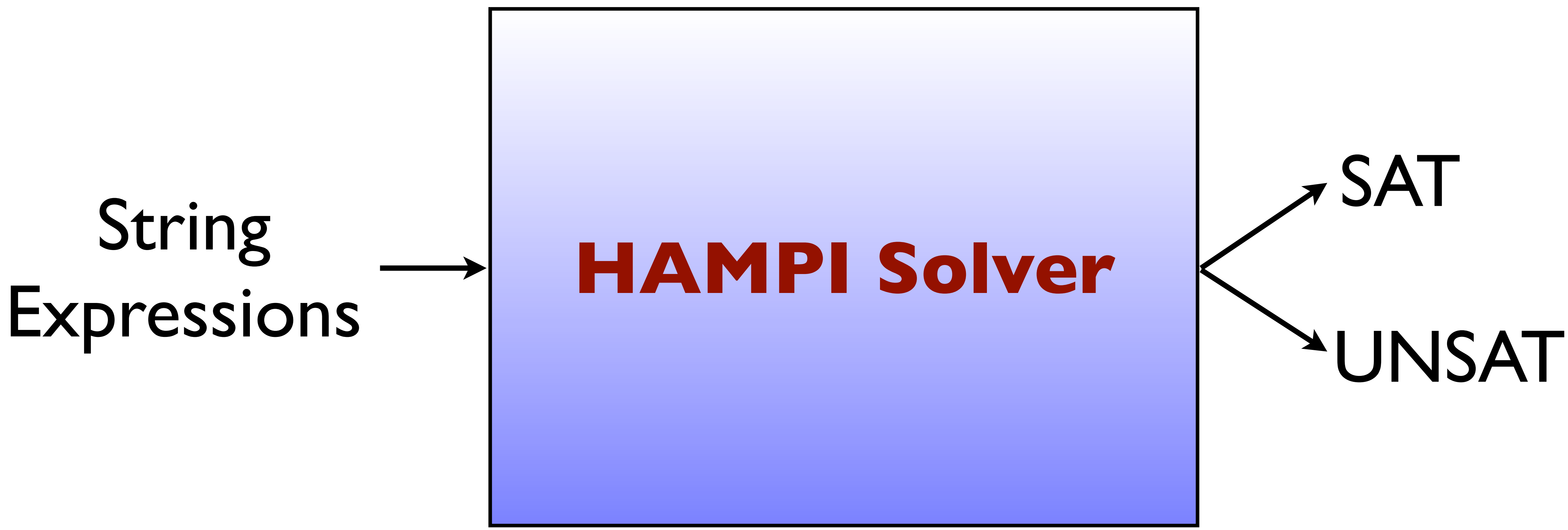
Impact of STP
http://www.metafuzz.com

Program Name	Lines of Code	Number of Bugs Found	Team
Mplayer	~900,000	Hundreds	David Molnar/Berkeley & Microsoft Research
Evince	~90,000	Hundreds	David Molnar/Berkeley & Microsoft Research
Unix Utilities	1000s	Dozens	Dawson Engler et al./Stanford
Crypto Hash Implementations	1000s	Verified	Eric Smith & David Dill/Stanford

Rest of the Talk

- STP Bit-vector and Array Solver
  - Why Bit-vectors and Arrays?
  - How does STP scale: Abstraction-refinement
  - Impact: Concolic testing
  - Experimental Results
- HAMPI String Solver
  - Why Strings?
  - How does HAMPI scale: Bounding
  - Impact: String-based program analysis
  - Experimental Results
- Future Work
  - Multicore SAT
  - SAT-based Languages

HAMPI String Solver



- X = concat("SELECT...",v) AND (X $\in$ SQL_grammar)
- JavaScript and PHP Expressions
- Web applications, SQL queries
- NP-complete

Theory of Strings
The Hampi Language

PHP/JavaScript/C++...	HAMPI: Theory of Strings	Notes
Var a; $a = ‘name’	Var a : 1...20; a = ‘name’	Bounded String Variables String Constants
string_expr.” is ”	concat(string_expr,“ is “);	Concat Function
substr(string_expr,1,3)	string_expr[1:3]	Extract Function

assignments/strcmp
a = string_expr;
a /= string_expr;
	equality
	a = string_expr;
	a /= string_expr;
		Equality Predicate

Sanity check in regular expression RE Sanity check in context-free grammar CFG	string_expr in RE string_expr in SQL string_expr NOT in SQL	Membership Predicate
string_expr contains a sub_str string_expr does not contain a sub_str	string_expr contains sub_str string_expr NOT?contains sub_str	Contains Predicate (Substring Predicate)

Theory of Strings

The Hampi Language

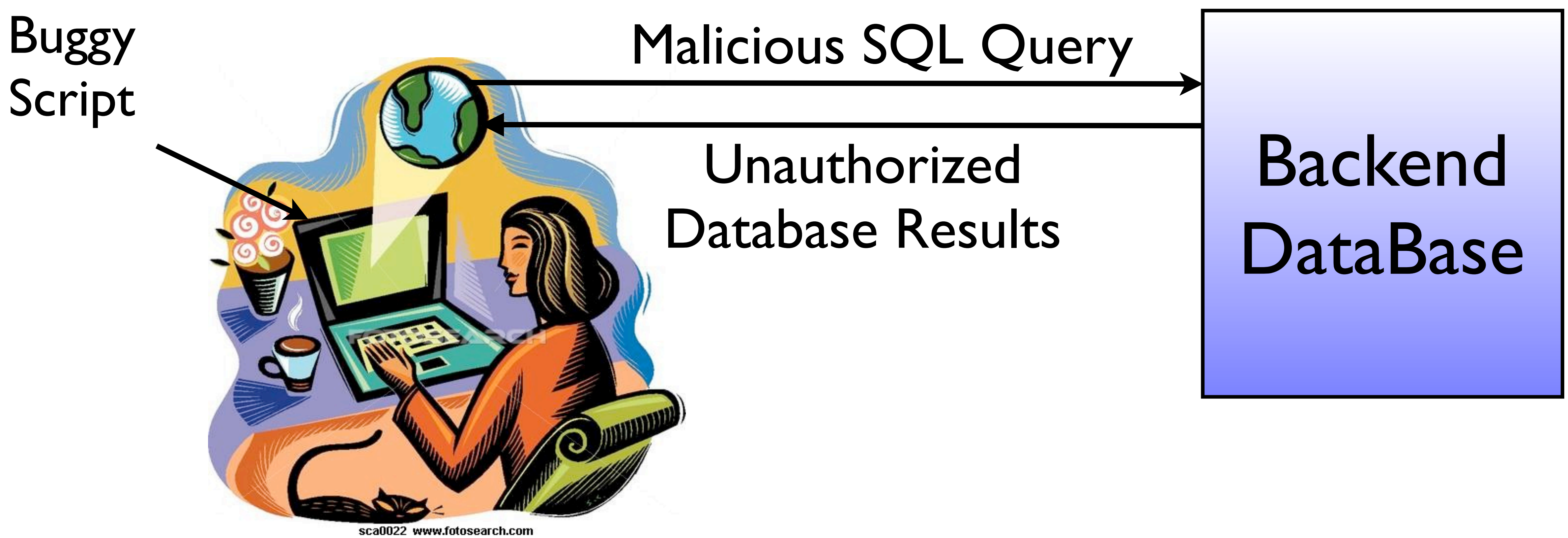
- X = concat("SELECT msg FROM msgs WHERE topicid = ",v)
  AND
  (X ∈ SQL_Grammar)

- input ∈ RegExp([0-9]+)

- X = concat (str_term1, str_term2, "c")[1:42]
  AND
  X contains "abc"

HAMPI Solver Motivating Example

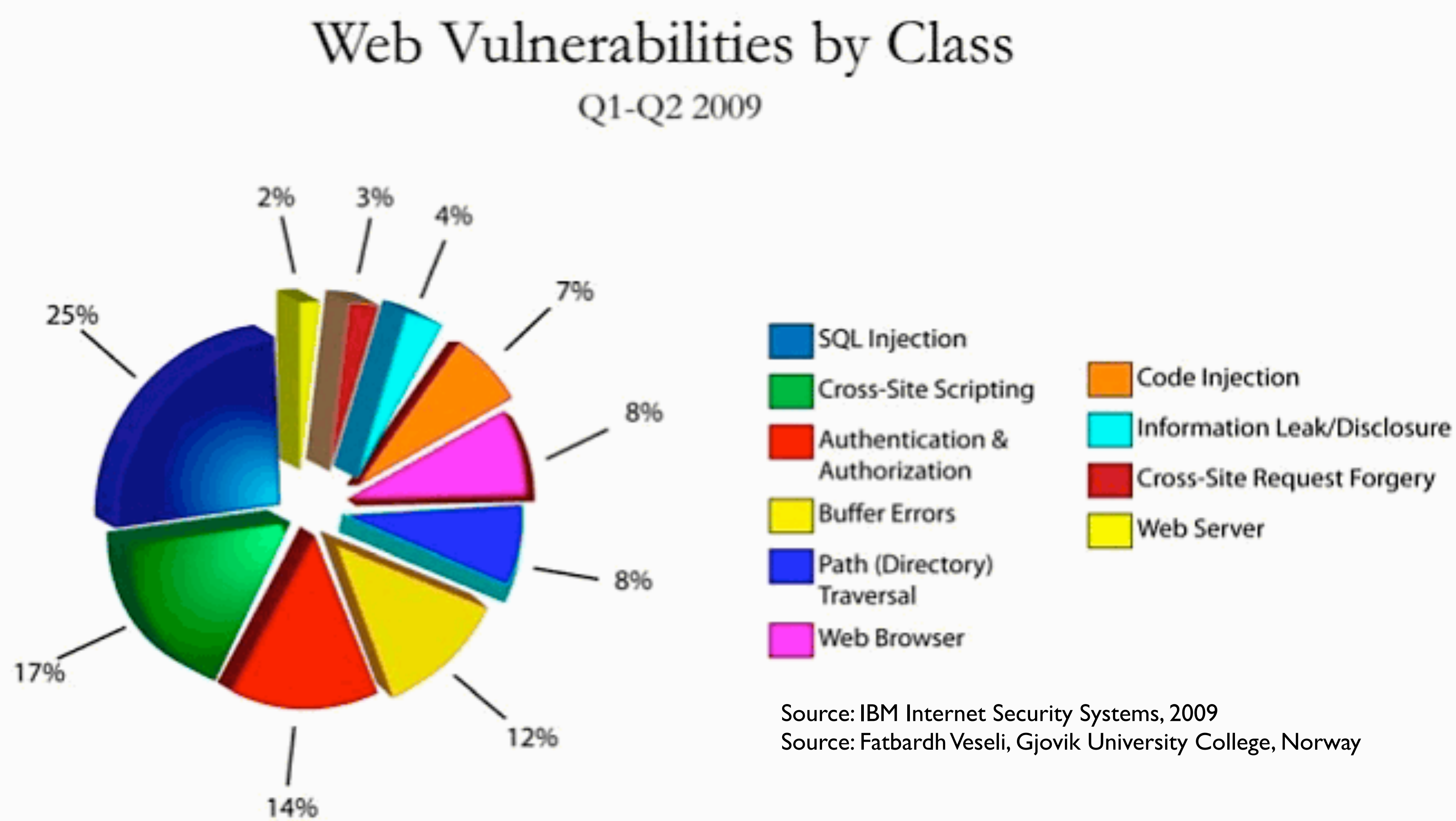
SQL Injection Vulnerabilities



SELECT m FROM messages WHERE id='1' OR 1 = 1

HAMPI Solver Motivating Example

SQL Injection Vulnerabilities



Source: IBM Internet Security Systems, 2009
Source: Fatbardh Veseli, Gjovik University College, Norway

HAMPI Solver Motivating Example

SQL Injection Vulnerabilities

Buggy Script

```
if (input in regexp("[0-9]+"))
   query := "SELECT m FROM messages WHERE id=' " + input + " ' ")
```

- input passes validation (regular expression check)
- query is syntactically-valid SQL
- query can potentially contain an attack substring (e.g., 1' OR '1' = '1)

HAMPI Solver Motivating Example

SQL Injection Vulnerabilities

Should be: “^[0-9]+$”

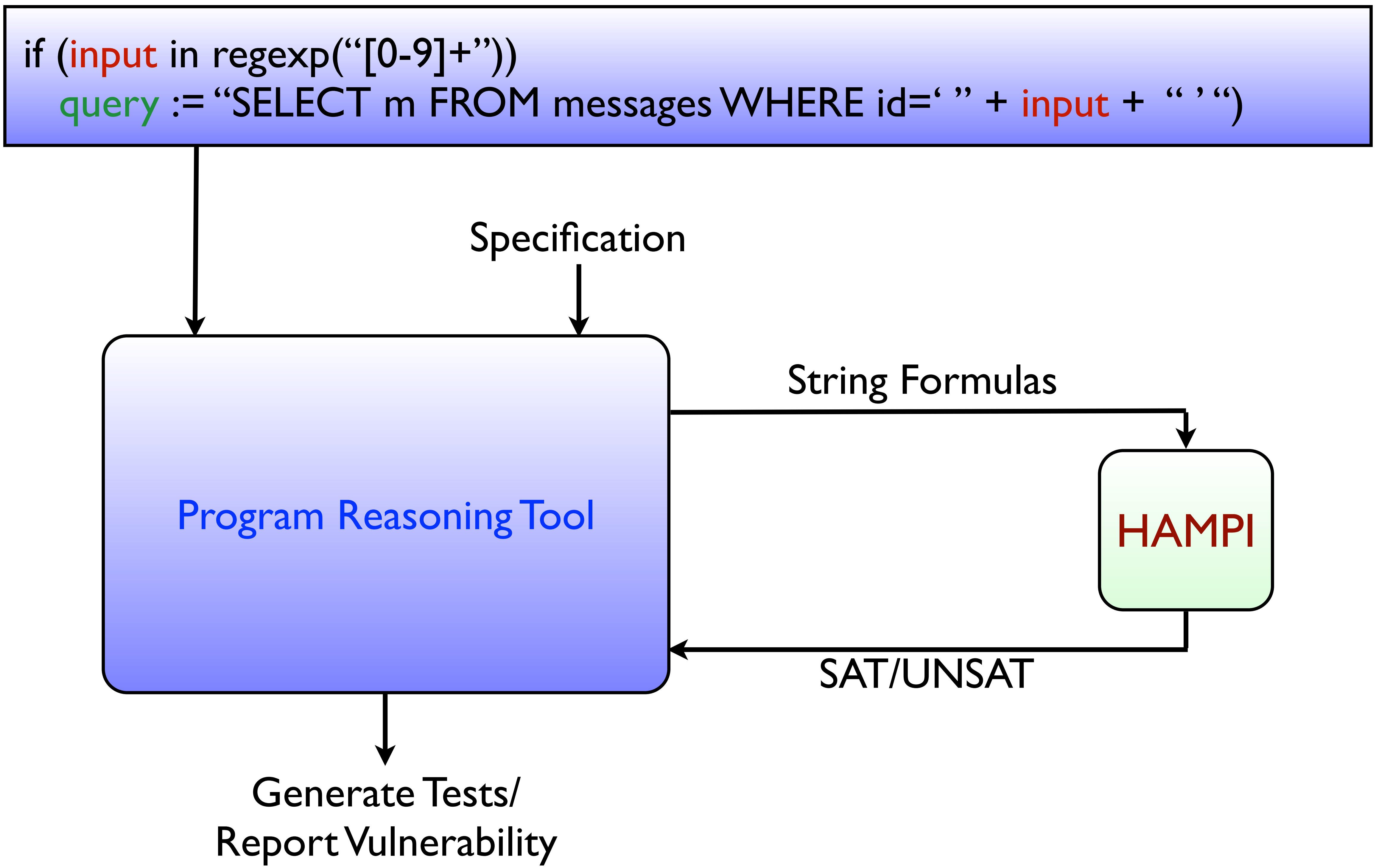
Buggy Script

```
if (input in regexp(“[0-9]+”))
   query := “SELECT m FROM messages WHERE id=‘ ” + input +  “ ’ “)
```

- input passes validation (regular expression check)
- query is syntactically-valid SQL
- query can potentially contain an attack substring (e.g., 1’ OR ‘1’ = ‘1)

HAMPI Solver Motivating Example

SQL Injection Vulnerabilities



Rest of the Talk

- HAMPI Logic: A Theory of Strings
- Motivating Example: HAMPI-based Vulnerability Detection App
- How HAMPI works
- Experimental Results
- Related Work: Theory and Practice
- HAMPI 2.0
- SMTization: Future of Strings

Expressing the Problem in HAMPI

SQL Injection Vulnerabilities

Input String

Var v : 12;

SQL Grammar

cfg *SqlSmall* := "SELECT ” [a-z]+ " FROM ” [a-z]+ " WHERE " *Cond*;

cfg *Cond* := *Val* "=" *Val* | *Cond* " OR " *Cond*;

cfg *Val* := [a-z]+ | "'” [a-z0-9]* "'" | [0-9]+;

SQL Query

val q := concat("SELECT msg FROM messages WHERE topicid='", v, "'");

assert v in [0-9]+;

assert q in *SqlSmall*;

SQLI attack conditions

assert q contains "OR ‘1'=‘1'";

“q is a valid SQL query”

“q contains an attack vector”

Hampi Key Conceptual Idea

Bounding, expressiveness and efficiency

L_i	Complexity of $\varnothing = L_1 \cap \ldots \cap L_n$	Current Solvers
Context-free	Undecidable	n/a
Regular	PSPACE-complete	Quantified Boolean Logic
Bounded	NP-complete	SAT Efficient in practice

Hampi Key Idea: Bounded Logics

Testing, Vulnerability Detection,...

- Finding SAT assignment is key
- Short assignments are sufficient

→

- Bounding is sufficient
- Bounded logics easier to decide

Hampi Key Idea: Bounded Logics
Bounding vs. Completeness

- Bounding leads to incompleteness
- Testing (Bounded MC) vs. Verification (MC)
- Bounding allows trade-off (Scalability vs. Completeness)
- Completeness (also, soundness) as resources

HAMPI Solver Motivating Example

SQL Injection Vulnerabilities

Input String

```
Var v : 12;
```

SQL Grammar

cfg *SqlSmall* := "SELECT ” [a-z]+ " FROM ” [a-z]+ " WHERE " *Cond*;

cfg *Cond* := *Val* "=" *Val* | *Cond* " OR " *Cond*;

cfg *Val* := [a-z]+ | "'” [a-z0-9]* "'" | [0-9]+;

SQL Query

val q := concat("SELECT msg FROM messages WHERE topicid='", v, "'");

assert v in [0-9]+;

SQLI attack conditions

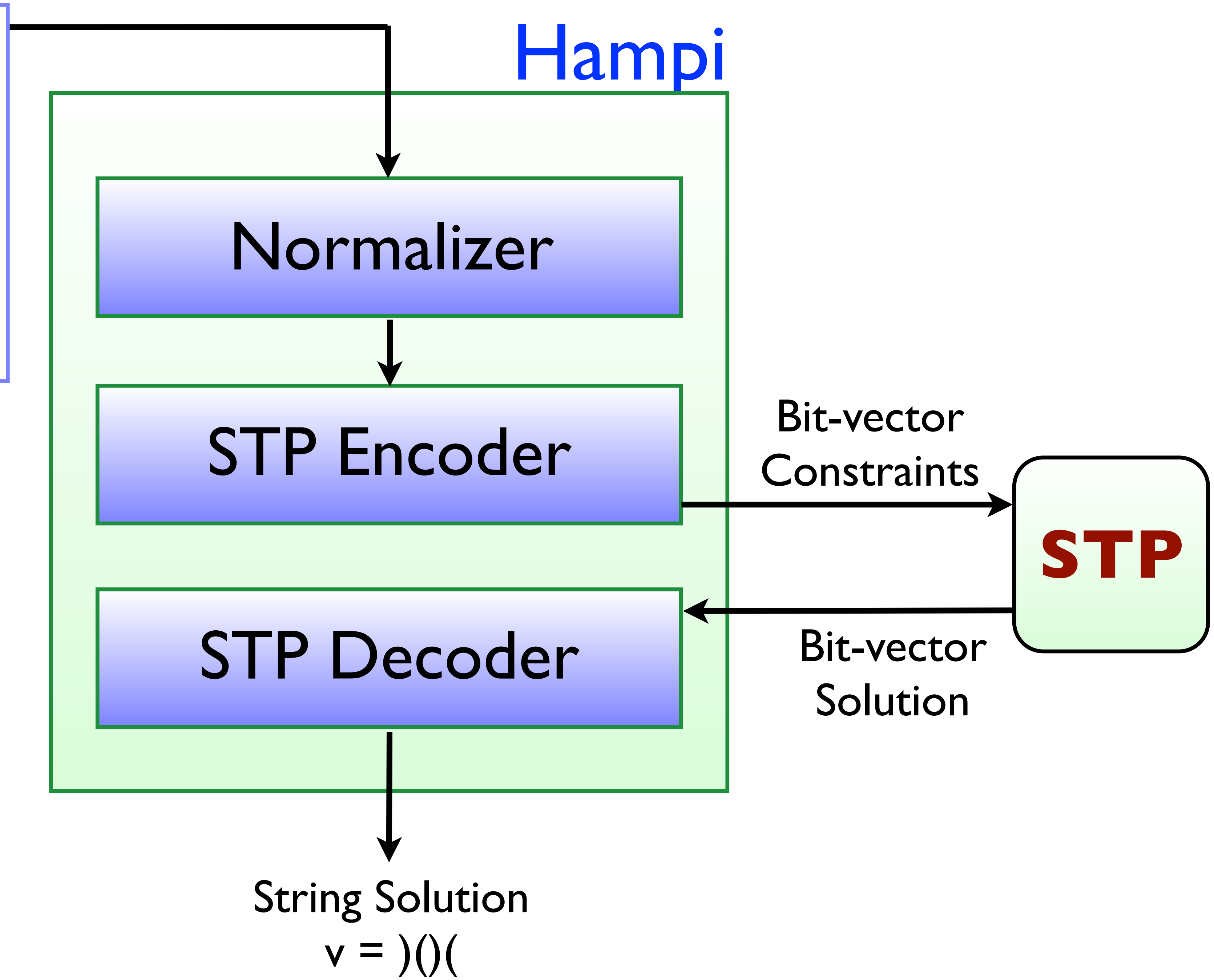
assert q in *SqlSmall*; — **“q is a valid SQL query”**

assert q contains "OR ‘1'=‘1'"; — **“q contains an attack vector”**

How Hampi Works

Bird’s Eye View: Strings into Bit-vectors

```
var v : 4;

cfg E := “()” | E E | “(“ E “)”;

val q := concat( “(“, v, ”)”);

assert q in E;
assert q contains “()()”;
```



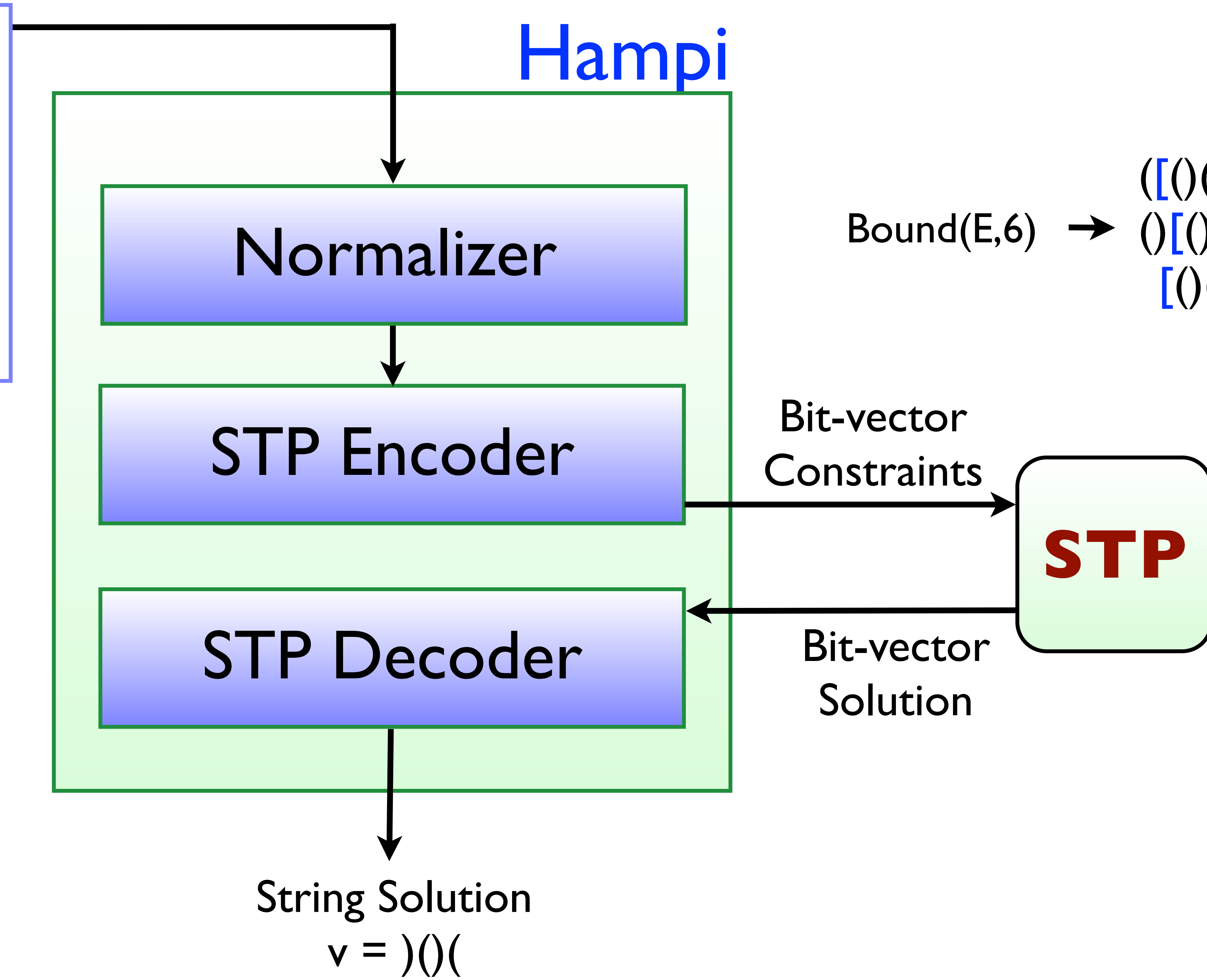
Find a 4-char string v:

- (v) is in E
- (v) contains ()()

How Hampi Works

Unroll Bounded CFGs into Regular Exp.

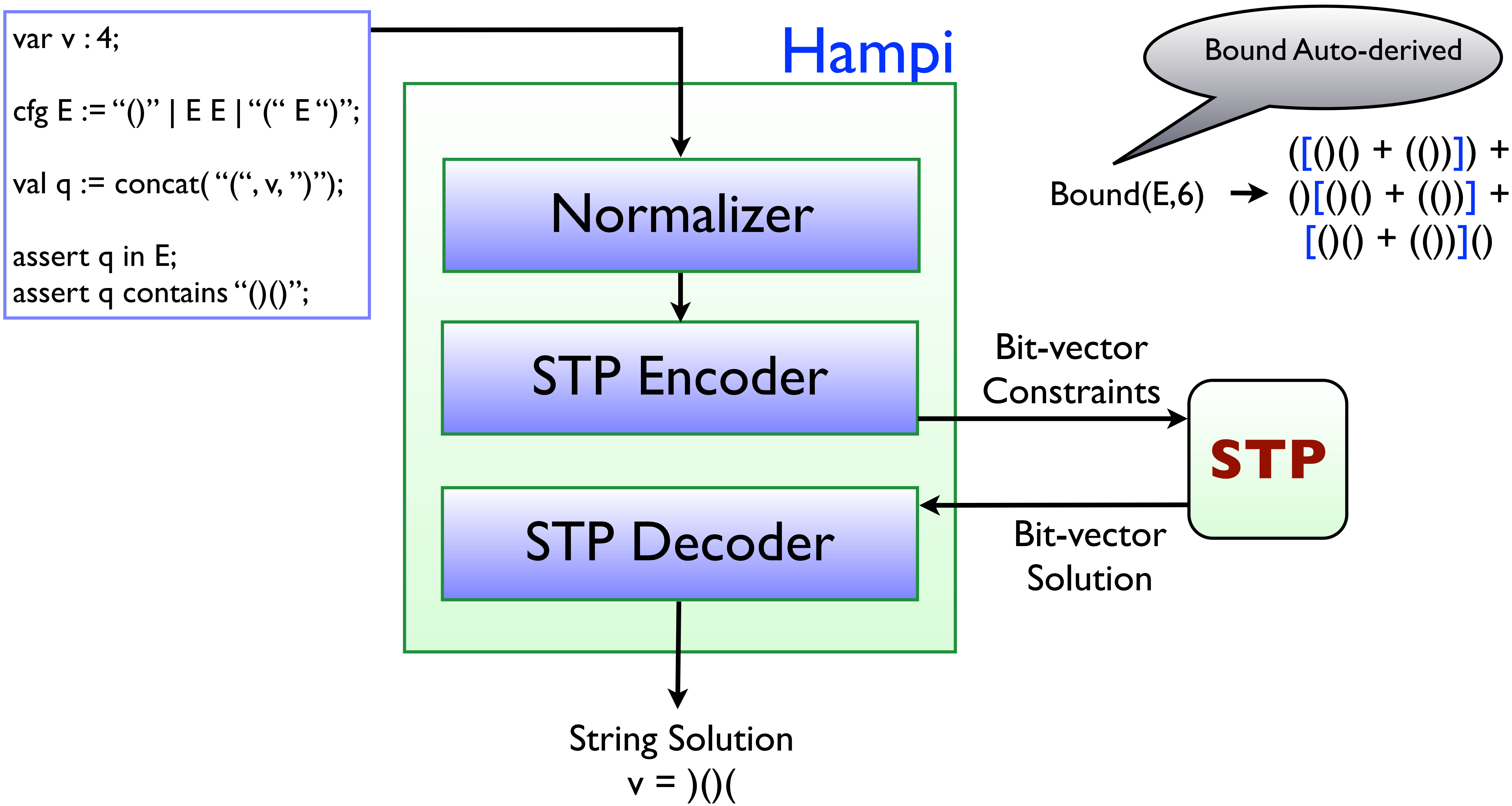
```
var v : 4;

cfg E := "()" | E E | "(" E ")";

val q := concat( "(", v, ")");

assert q in E;
assert q contains "()()";
```



Bound(E,6) → ([()() + (())]) + ()[()() + (())] + [()() + (())]()

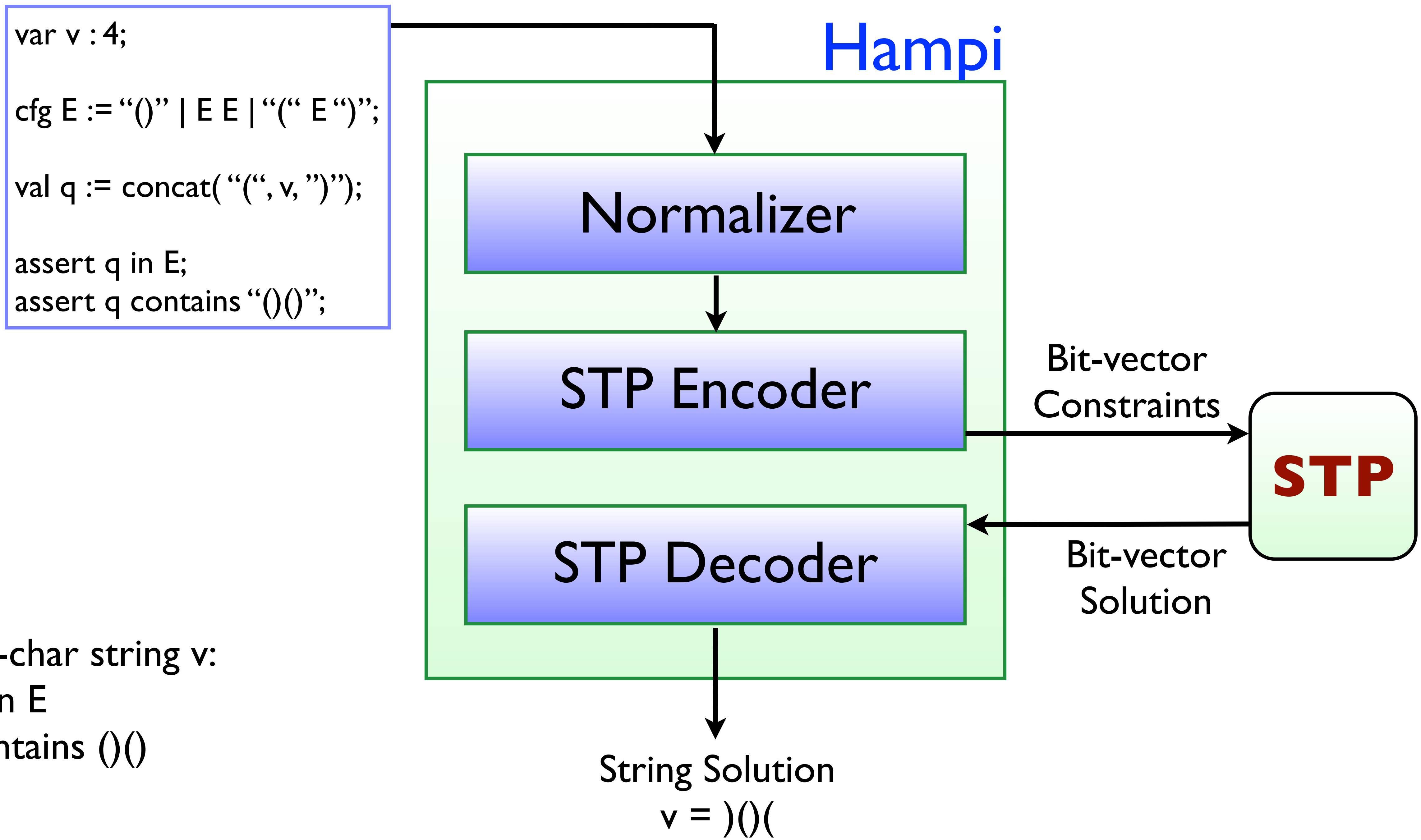
How Hampi Works

Unroll Bounded CFGs into Regular Exp.



How Hampi Works

Bird's Eye View: Strings into Bit-vectors



Find a 4-char string v:

- (v) is in E
- (v) contains ()()

How Hampi Works
Unroll Bounded CFGs into Regular Exp.

Step 1:

```
var v : 4;

cfg E := "()" | E E | "(" E ")";

val q := concat( "(", v, ")");

assert q in E;
assert q contains "()()";
```

→ Auto-derive lower/upper bounds [L,B] on CFG → [6,6]

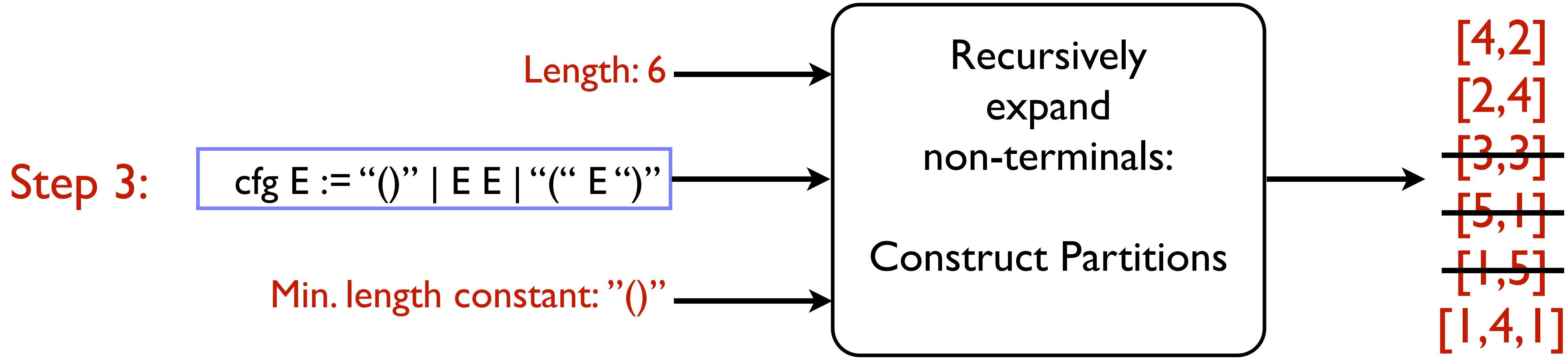
Step 2:

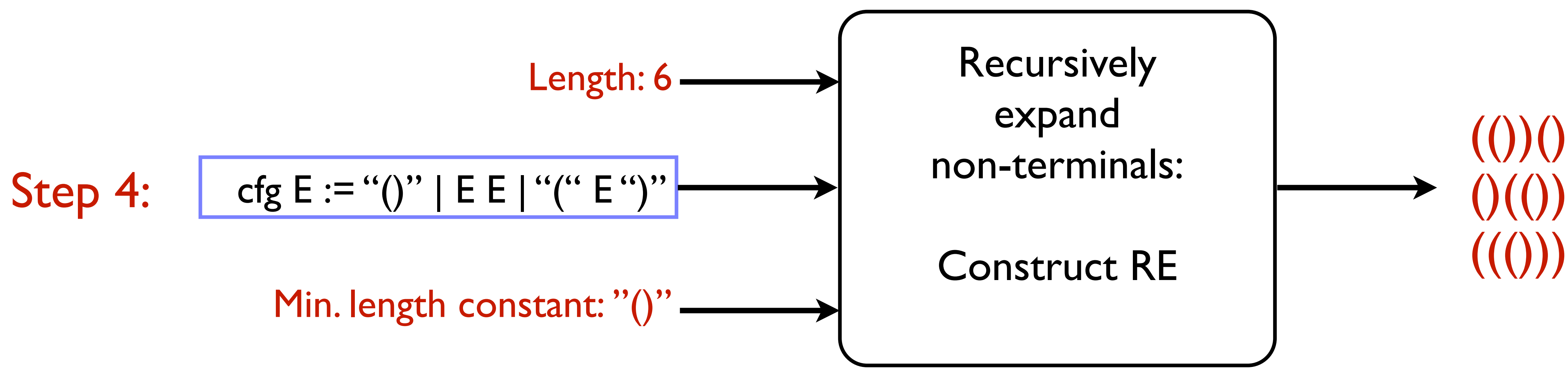
```
cfg E := "()" | E E | "(" E ")"
```

→ Look for minimal length string → "()"

How Hampi Works

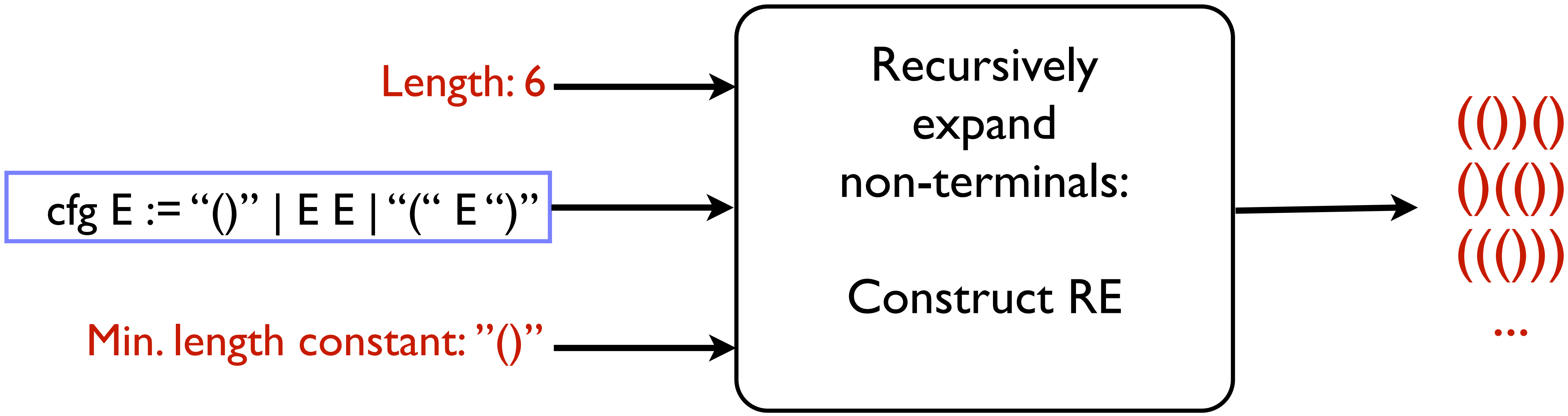
Unroll Bounded CFGs into Regular Exp.





Unroll Bounded CFGs into Regular Exp.

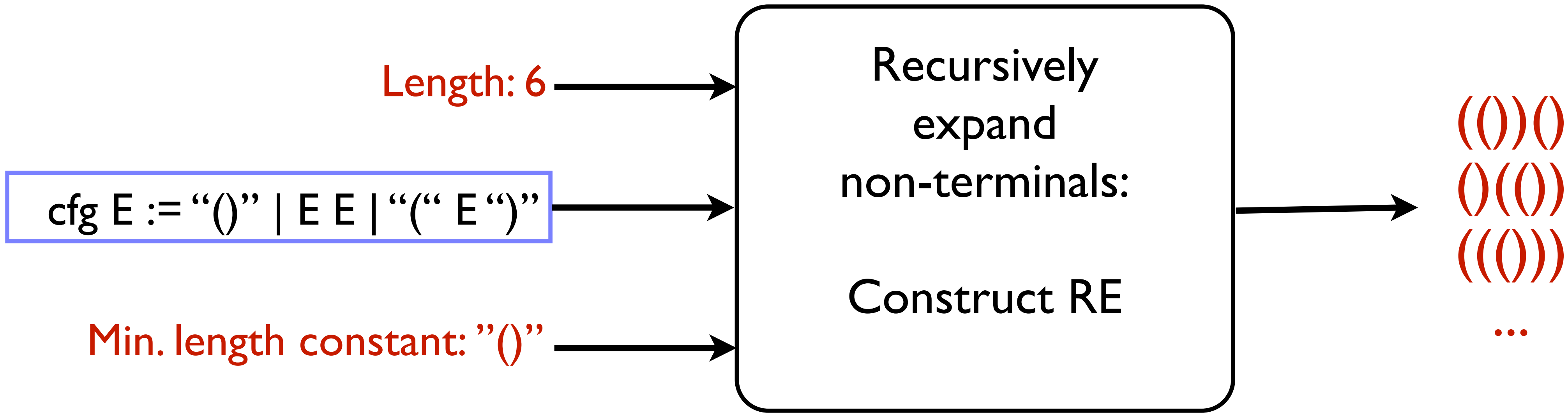
Managing Exponential Blow-up



- Dynamic programming style
- Works well in practice

Unroll Bounded CFGs into Regular Exp.

Managing Exponential Blow-up



Bound(E,6) → ([()() + (())]) + ()[()() + (())] + [()() + (())]()

How Hampi Works
Converting Regular Exp. into Bit-vectors

Encode regular expressions recursively

- Alphabet { (,) } → 0, 1
- constant → bit-vector constant
- union + → disjunction ∨
- concatenation → conjunction ∧
- Kleene star * → conjunction ∧
- Membership, equality → equality

```
( v ) ∈ ()[()() + (())] + [()() + (())]() + ([()() + (())])
```

↓ ↓ ↓

Formula Φ_1 ∨ Formula Φ_2 ∨ Formula Φ_3

$B[0]=0 \wedge B[1]=1 \wedge \{B[2]=0 \wedge B[3]=1 \wedge B[4]=0 \wedge B[5]=1 \vee \ldots$

How Hampi Works

Converting Regular Exp. into Bit-vectors

(v) ∈ ()[()() + (())] + [()() + (())]() + ([()() + (())])

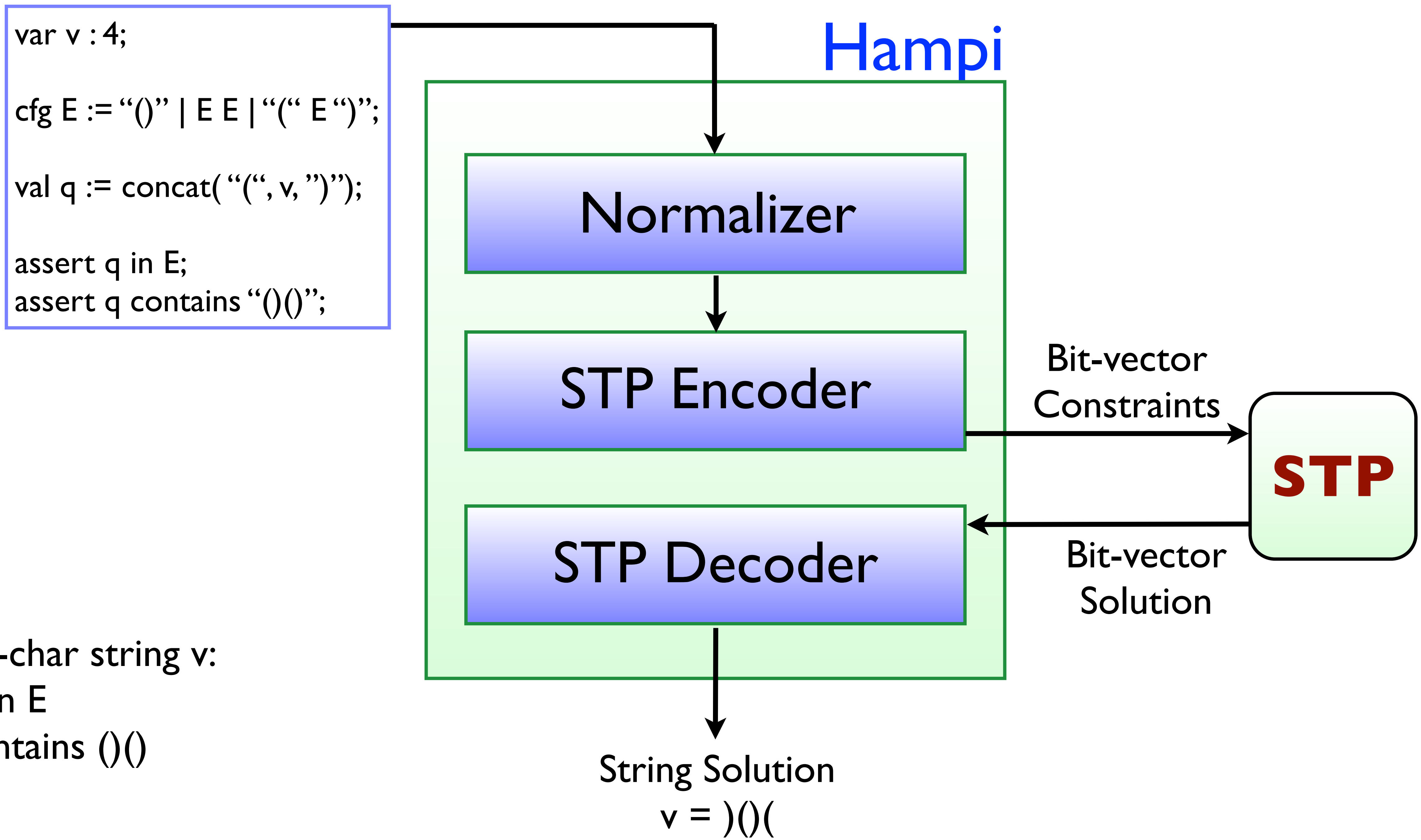
Formula Φ_1 ∨ Formula Φ_2 ∨ Formula Φ_3

B[0]=0 ∧ B[1]=1 ∧ {B[2]=0∧B[3]=1∧B[4]=0∧B[5]=1 v...

- Constraint Templates
- Encode once, and reuse
- On-demand formula generation

How Hampi Works

Decoder converts Bit-vectors to Strings



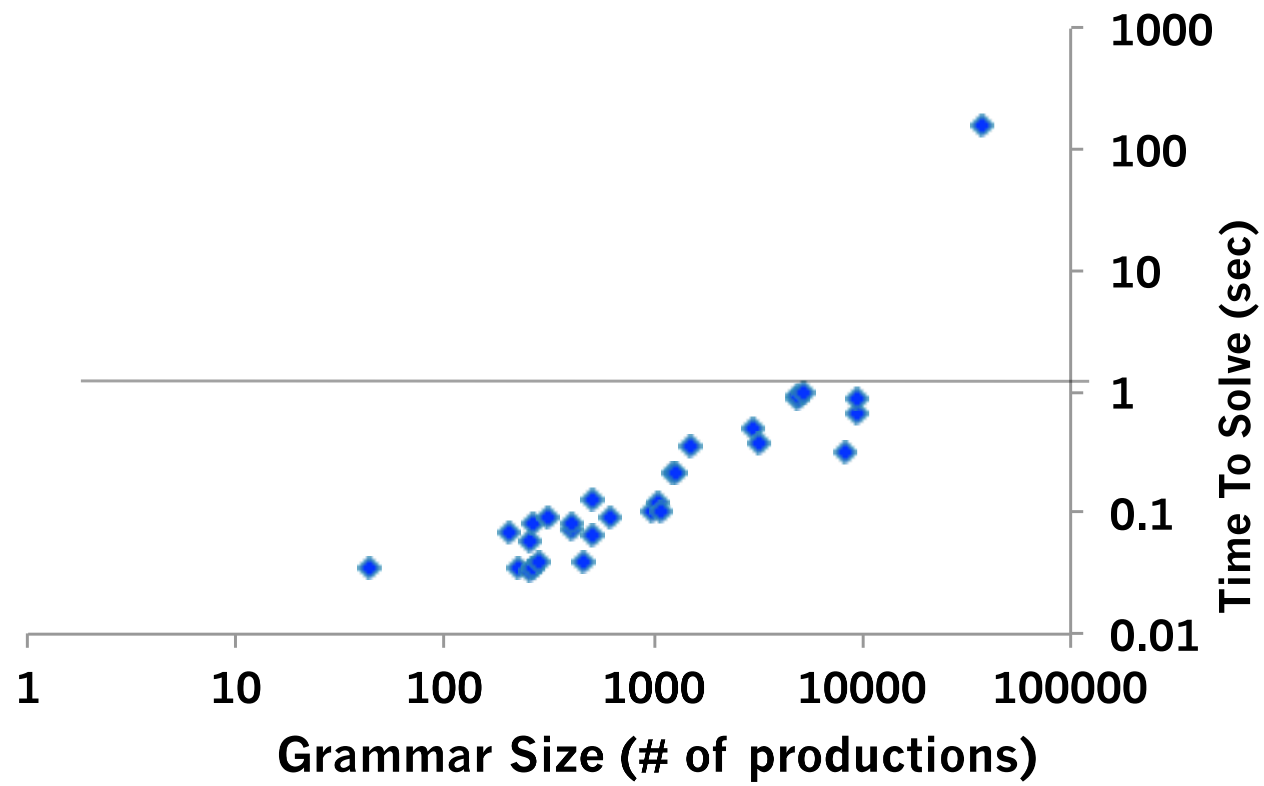
Find a 4-char string v:

- (v) is in E
- (v) contains ()()

Rest of the Talk

- HAMPI Logic: A Theory of Strings
- Motivating Example: HAMPI-based Vulnerability Detection App
- How HAMPI works
- Experimental Results
- Related Work: Theory and Practice
- HAMPI 2.0
- SMTization: Future of Strings

HAMPI: Result 1
Static SQL Injection Analysis



- 1367 string constraints from Wasserman & Su [PLDI'07]
- Hampi scales to large grammars
- Hampi solved 99.7% of constraints in < 1sec
- All solvable constraints had short solutions

HAMPI: Result 2
Security Testing and XSS

- Attackers inject client-side script into web pages
- Somehow circumvent same-origin policy in websites
- echo "Thank you $my_poster for using the message board";
- Unsanitized $my_poster
- Can be JavaScript
- Execution can be bad

HAMPI: Result 2
Security Testing

- Hampi used to build Ardilla security tester [Kiezun et al., ICSE'09]

- 60 new vulnerabilities on 5 PHP applications (300+ kLOC)
  - 23 SQL injection
  - 37 cross-site scripting (XSS)

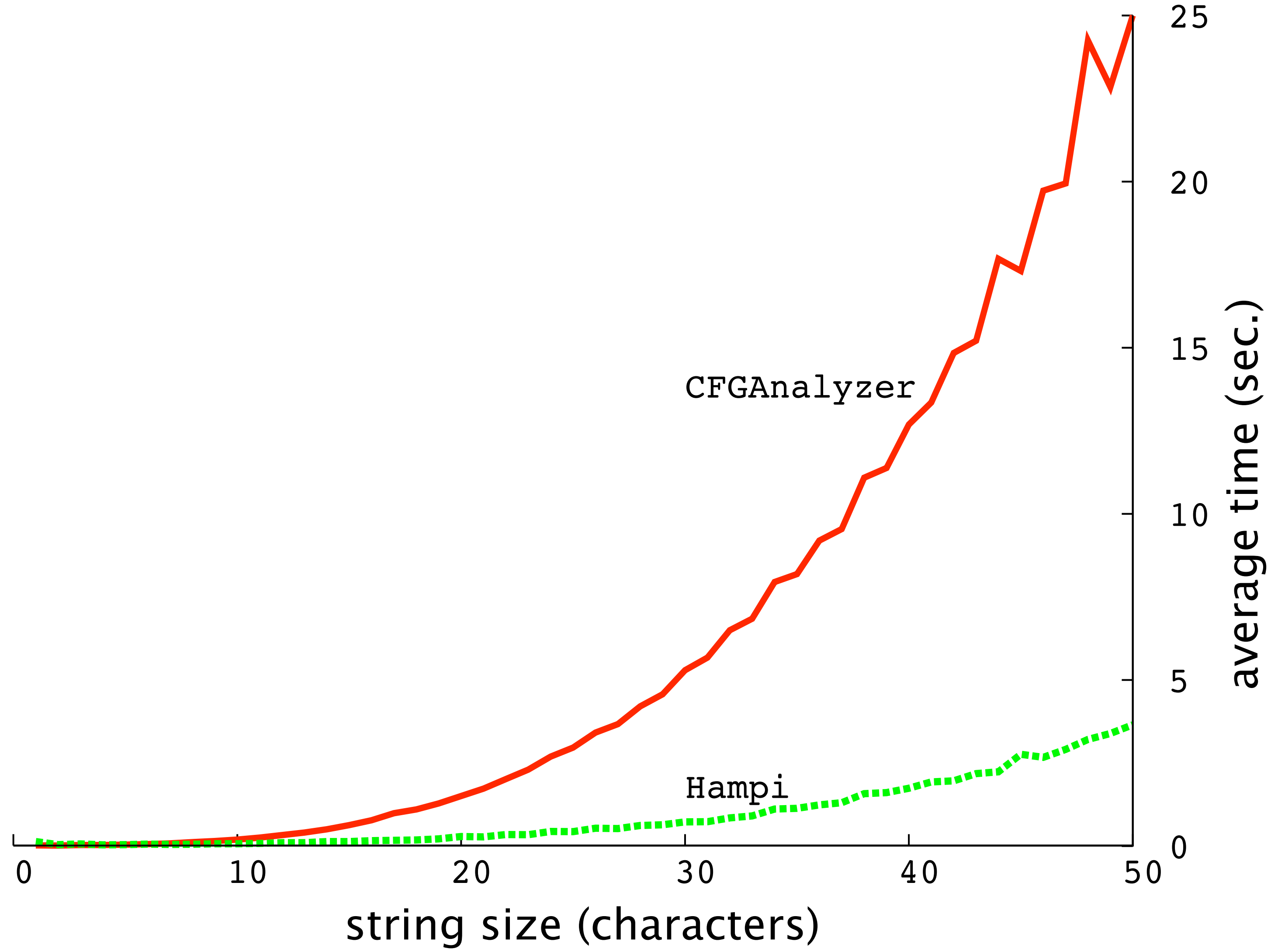
5 added to
US National Vulnerability DB

- 46% of constraints solved in < 1 second per constraint

- 100% of constraints solved in <10 seconds per constraint

HAMPI: Result 3

Comparison with Competing Tools



- HAMPI vs. CFGAnalyzer (U. Munich): HAMPI ~7x faster for strings of size 50+

HAMPI: Result 3

Comparison with Competing Tools

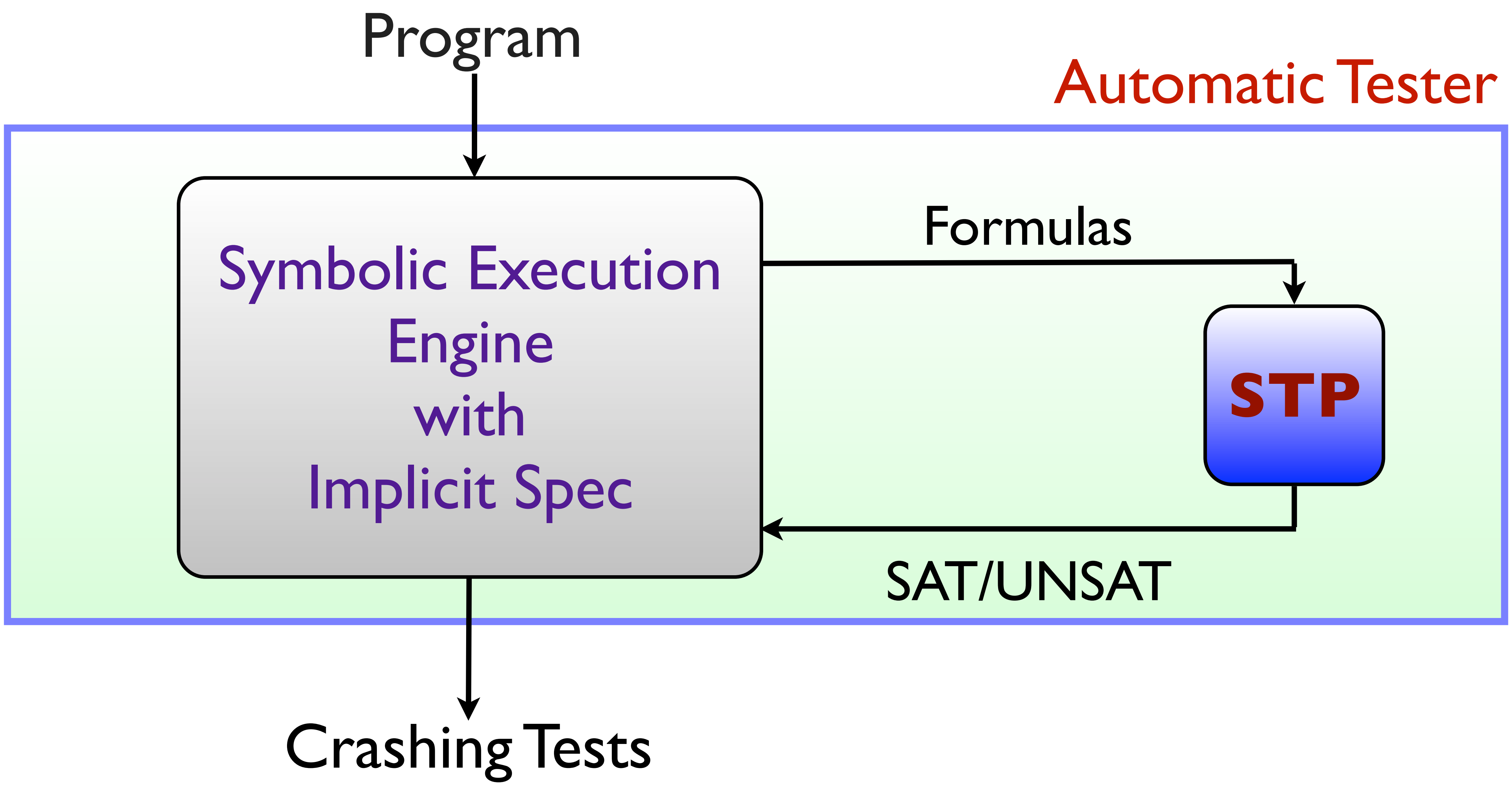
RE intersection problems

- HAMPI 100x faster than Rex (MSR)
- HAMPI 1000x faster than DPRLE (U.Virginia)
- Pieter Hooimeijer 2010 paper titled 'Solving String Constraints Lazily'

How to Automatically Crash Programs?

KLEE: Concolic Execution-based Tester

Problem: Automatically generate crashing tests given only the code



How to Automatically Crash Programs?

KLEE: Concolic Execution-based Tester

Structured input processing code:
PDF Reader, Movie Player,...

```
Buggy_C_Program(int* data_field, int len_field) {

    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
      //1. Integer overflow causing NULL deref
      //2. Buffer overflow
      *(ptr+i) = process_data(*(data_field+i));
    }
}
```

- Formula captures computation
- Tester attaches formula to capture spec

How to Automatically Crash Programs?

KLEE: Concolic Execution-based Tester

Structured input processing code:
PDF Reader, Movie Player,...

```
Buggy_C_Program(int* data_field, int len_field) {

    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
      //1. Integer overflow causing NULL deref
      //2. Buffer overflow
      *(ptr+i) = process_data(*(data_field+i));
    }
}
```

Equivalent Logic Formula derived using
symbolic execution

```
data_field, mem_ptr :  ARRAY;
len_field : BITVECTOR(32); //symbolic
i,  j, ptr     : BITVECTOR(32);//symbolic
.
.
mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);
.
.
```

- Formula captures computation
- Tester attaches formula to capture spec

How to Automatically Crash Programs?

KLEE: Concolic Execution-based Tester

Structured input processing code:
PDF Reader, Movie Player,...

```
Buggy_C_Program(int* data_field, int len_field) {

    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
      //1. Integer overflow causing NULL deref
      //2. Buffer overflow
      *(ptr+i) = process_data(*(data_field+i));
    }
}
```

↔

Equivalent Logic Formula derived using symbolic execution

```
data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr : BITVECTOR(32);//symbolic
.
.
mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);
.
.
```

- Formula captures computation
- Tester attaches formula to capture spec

How to Automatically Crash Programs?

KLEE: Concolic Execution-based Tester

Structured input processing code:
PDF Reader, Movie Player,...

```
Buggy_C_Program(int* data_field, int len_field) {

    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
      //1. Integer overflow causing NULL deref
      //2. Buffer overflow
      *(ptr+i) = process_data(*(data_field+i));
    }
}
```

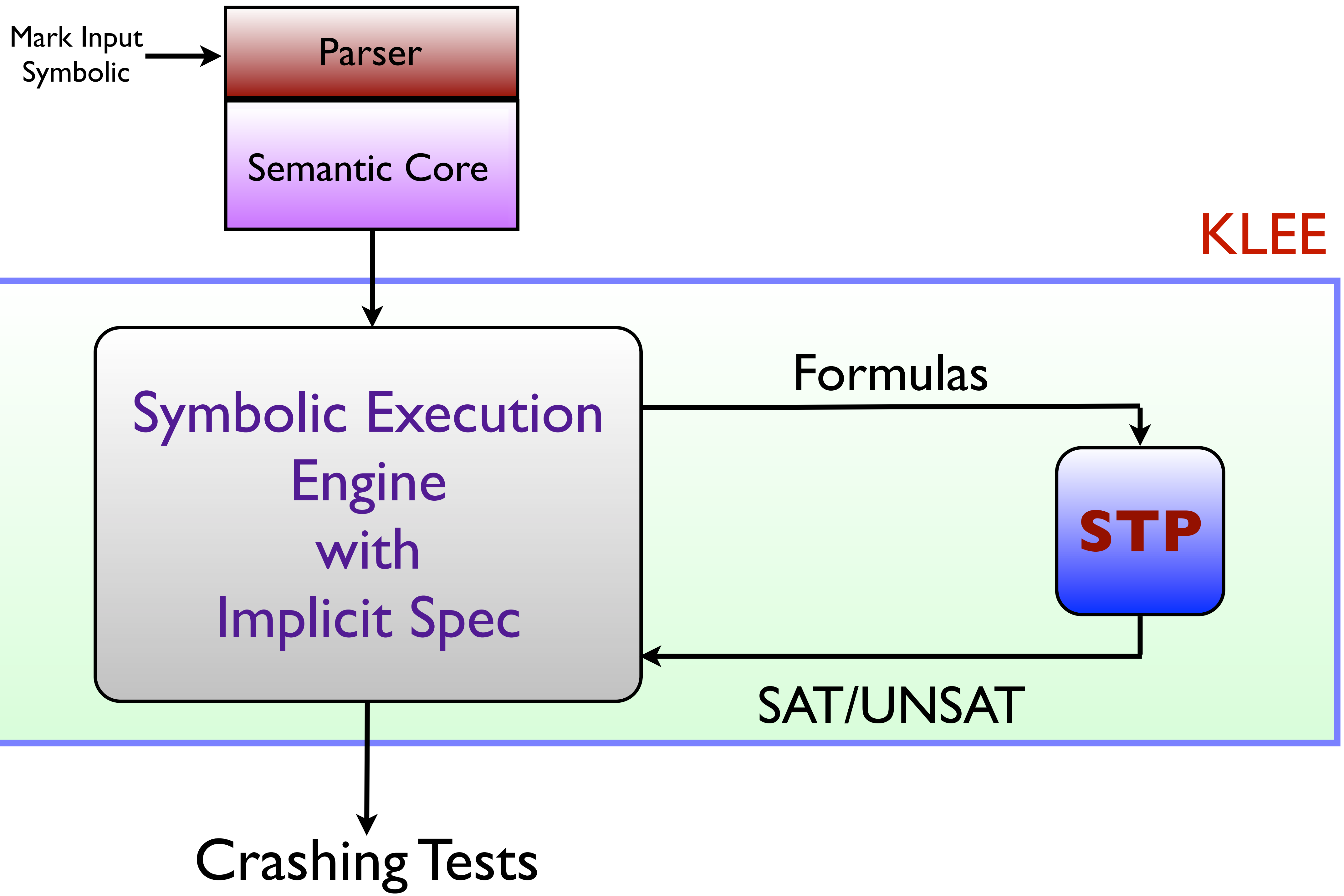
↔

Equivalent Logic Formula derived using symbolic execution

```
data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr    : BITVECTOR(32);//symbolic
.
.
mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);
.
.
//INTEGER OVERFLOW QUERY
0 <= j <= process(len_field);
ptr + i + j = 0?
```

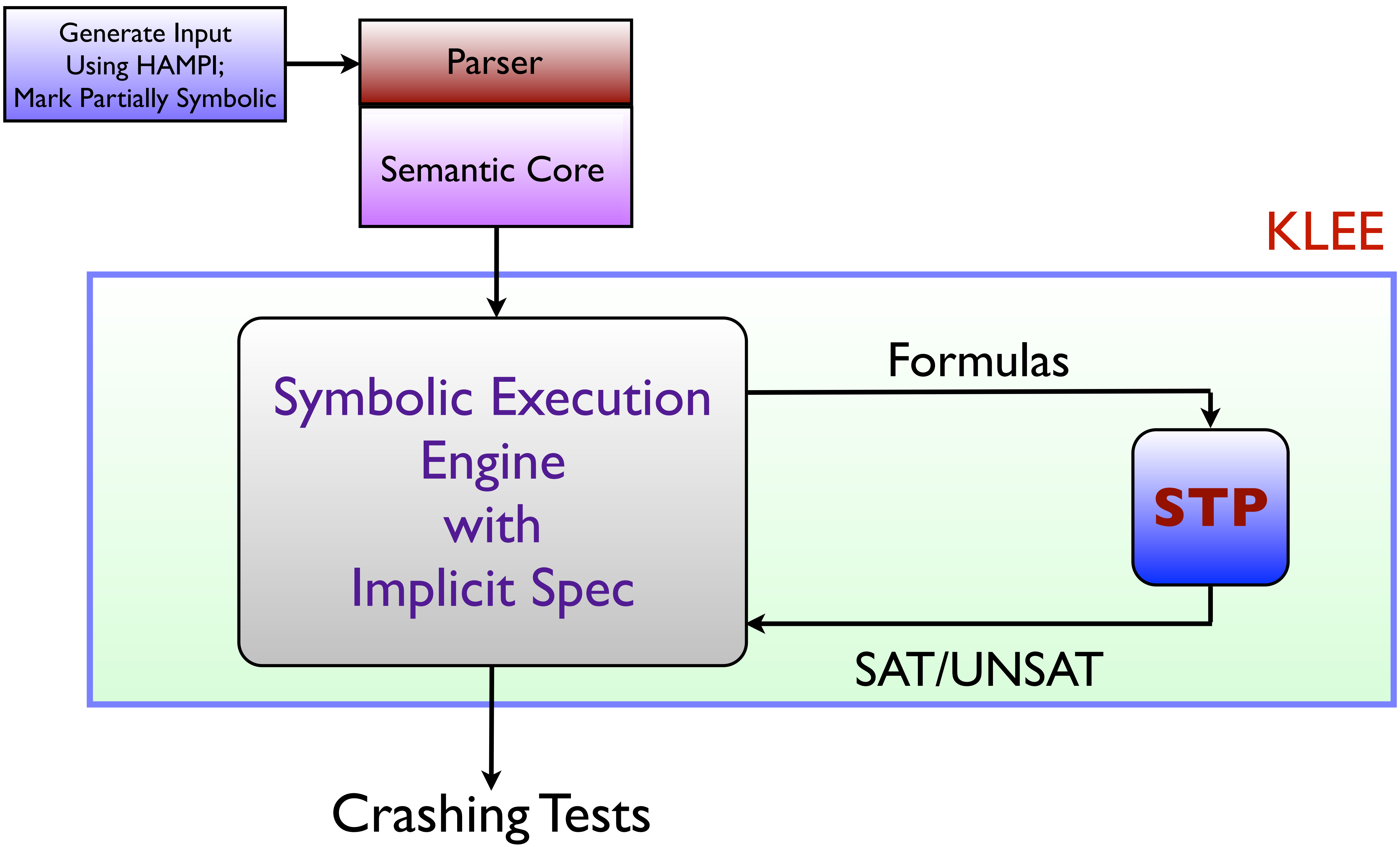
- Formula captures computation
- Tester attaches formula to capture spec

HAMPI: Result 4
Helping KLEE Pierce Parsers



HAMPI: Result 4

Helping KLEE Pierce Parsers



HAMPI: Result 4
Helping KLEE Pierce Parsers

- Klee provides API to place constraints on symbolic inputs
- Manually writing constraints is hard
- Specify grammar using HAMPI, compile to C code
- Particularly useful for programs with highly-structured inputs
- 2-5X improvement in line coverage

Impact of Hampi: Notable Projects

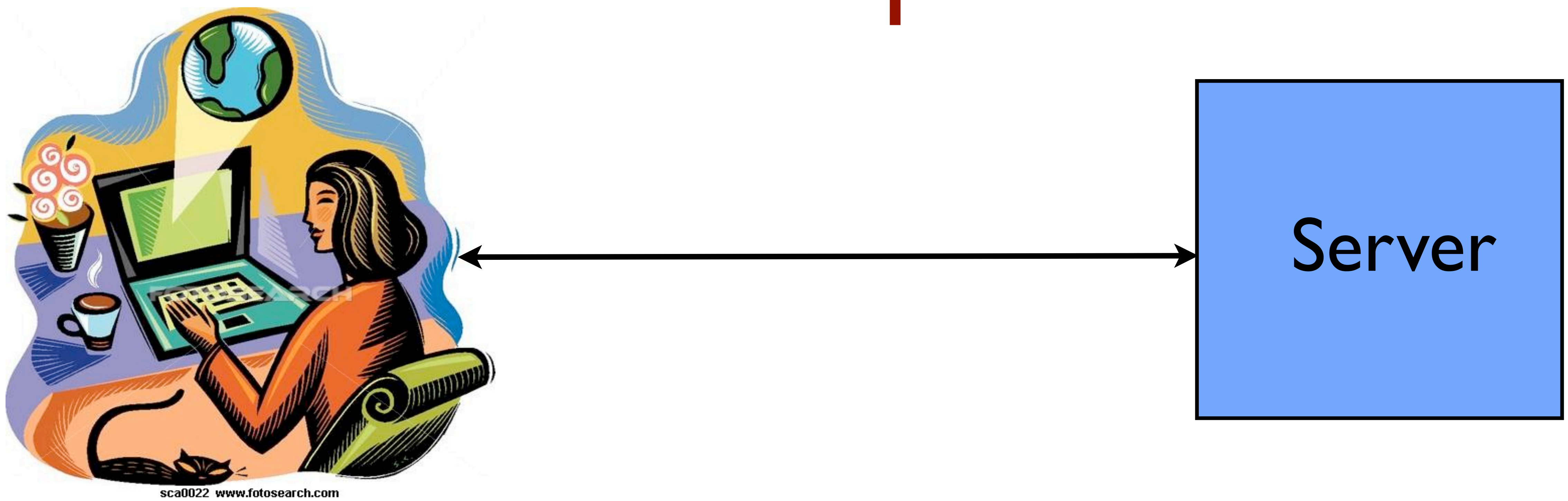
Category	**Research Project**	**Project Leader/Institution**
Static Analysis	SQL-injection vulnerabilities	Wasserman & Su/UC, Davis
Security Testing	Ardilla for PHP (SQL injections, cross-site scripting)	Kiezun & Ernst/MIT
Concolic Testing	Klee Kudzu NoTamper	Engler & Cadar/Stanford Saxena & Song/Berkeley Bisht & Venkatakrishnan/U Chicago
New Solvers	Kaluza	Saxena & Song/Berkeley

Impact of Hampi: Notable Projects

Tool Name	**Description**	**Project Leader/ Institution**
Kudzu	JavaScript Bug Finder & Vulnerability Detector	Saxena Akhawe Hanna Mao McCamant Song/Berkeley
NoTamper	Parameter Tamper Detection	Bisht Hinrichs/U of Chicago Skrupsky Bobrowicz Vekatakrishnan/ U. of Illinois, Chicago

Impact of Hampi: Notable Projects

NoTamper



- Client-side checks (C), no server checks
- Find solutions S_1,S_2,... to C, and solutions E_1,E_2,... to ~C by calling HAMPI
- E_1,E_2,... are candidate exploits
- Submit (S1, E1),... to server
- If server response same, ignore
- If server response differ, report error

Related Work (Practice)

Tool Name	**Project Leader/ Institution**	**Comparison with HAMPI**
Rex	Bjorner, Tillman, Vornkov et al. (Microsoft Research, Redmond)	• HAMPI + Length+Replace(s_1,s_2,s_3) - CFG • Translation to int. linear arith. (Z3)
Mona	Karlund et al. (U. of Aarhus)	• Can encode HAMPI & Rex • User work • Automata-based • Non-elementary
DPRLE	Hooimeijer (U. of Virginia)	• Regular expression constraints

Related Work (Theory)

Result	Person (Year)	Notes
Undecidability of Quantified Word Equations	Quine (1946)	Multiplication reduced to concat
Undecidability of Quantified Word Equations with single alternation	Durnev (1996), G. (2011)	2-counter machines reduced to words with single quantifier alter.
Decidability (PSPACE) of QF Theory of Word Equations	Makanin (1977) Plandowski (1996, 2002/06)	Makanin result very difficult Simplified by Plandowski
Decidability (PSPACE-complete) of QF Theory of Word Equations + RE	Schultz (1992)	RE membership predicate
QF word equations + Length() (?)	Matiyasevich (1971)	Unsolved Reduction to Diophantine
QF word equations in solved form + Length() + RE	G. (2011)	Practical

Future of HAMPI & STP

- HAMPI will be combined with STP
  - Bit-vectors and Arrays
  - Integer/Real Linear Arithmetic
  - Uninterpreted Functions
  - Strings
  - Floating Point
  - Non-linear
- Additional features planned in STP
  - UNSAT Core
  - Quantifiers
  - Incremental
  - DPLL(T)
  - Parallel STP
  - MAXSMT?
- Extensibility and hackability by non-expert

Future of Strings

- Strings SMTization effort started
  - Nikolaj Bjorner, G.
  - Andrei Voronkov, Ruzica Piskac, Ting Zhang
  - Cesare Tinelli, Clark Barrett, Dawn Song, Prateek Saxena, Pieter Hooimeijer, Tim Hinrichs

- SMT Theory of Strings
  - Alphabet (UTF, Unicode,...)
  - String Constants and String Vars (parameterized by length)
  - Concat, Extract, Replace, Length Functions
  - Regular Expressions, CFGs (Extended BNF)
  - Equality, Membership Predicate, Contains Predicate

- Applications
  - Static/Dynamic Analysis for Vulnerability Detection
  - Security Testing using Concolic Idea
  - Formal Methods
  - Synthesis

Conclusions & Take Away

- SMT solvers essential for testing, analysis, verification,...
- Core SMT ideas
  - Combinations
  - DPLL(T)
  - Over/Under approximations (CEGAR,...)
  - SAT solvers
- Future of SMT solvers
  - SMT + Languages
  - SMT + Synthesis
  - Parallel SAT/SMT
- Demand for even richer theories
  - Attribute grammars
  - String theories with length

Modern SMT Solver References

These websites and handbook have all the references you will need

1. Armin Bierre, Marijn Heule, Hans van Maaren, and Toby Walsh (Editors). *Handbook of Satisfiability*. 2009. IOS Press. http://www.st.ewi.tudelft.nl/sat/handbook/

2. SAT Live: http://www.satlive.org/

3. SMT LIB: http://www.smtlib.org/

4. SAT/SMT summer school: http://people.csail.mit.edu/vganesh/summerschool/

Topics Covered

Topics covered in Lecture 1

☑ **Motivation for SAT/SMT solvers in software engineering**
- Software engineering (SE) problems reduced to logic problems
- Automation, engineering, usability of SE tools through solvers

☑ **High-level description of the SAT/SMT problem & logics**
- Rich logics close to program semantics
- Demonstrably easy to solve in many practical cases

☑ **Modern SAT solver architecture & techniques**
- DPLL search, shortcomings
- Modern CDCL SAT solver: propagate (BCP), decide (VSIDS), conflict analysis, clause learn, backJump,
- Termination, correctness
- Big lesson: learning from mistakes

Topics covered in Lecture 2

☑ **Modern SMT solver architecture & techniques**
- Rich logics closer to program semantics
- DPLL(T), Combinations of solvers, Over/under approximations

☑ **My own contributions: STP & HAMPI**
- Abstraction-refinement for solving
- Bounded logics

☑ **SAT/SMT-based applications**
- Dynamic systematic testing
- Static, dynamic analysis for vulnerability detection

☑ **Future of SAT/SMT solvers**

Key Contributions
http://people.csail.mit.edu/vganesh

Name	**Key Concept**	**Impact**	**Pubs**
STP Bit-vector & Array Solver[1,2]	Abstraction-refinement for Solving	Concolic Testing	CAV 2007 CCS 2006 TISSEC 2008
HAMPI String Solver[1]	App-driven Bounding for Solving	Analysis of Web Apps	ISSTA 2009[3] TOSEM 2011 (CAV 2011)
Taint-based Fuzzing	Information flow is cheaper than concolic	Scales better than concolic	ICSE 2009
Automatic Input Rectification	Acceptability Envelope: Fix the input, not the program	New way of approaching SE	Under Submission

1. 100+ research projects use STP and HAMPI
2. STP won the SMTCOMP 2006 and 2010 competitions for bit-vector solvers
3. HAMPI: ACM Best Paper Award 2009
4. Retargetable Compiler (DATE 1999)
5. Proof-producing decision procedures (TACAS 2003)
6. Error-finding in ARBAC policies (CCS 2011)