SAT/SMT Solvers for Software Engineering and Security

A Short Course

by

Vijay Ganesh

Assistant Professor, University of Waterloo, Canada.

Date: April 4-8, 2016

Venue: Moscow State University, Russia.

Lecture 1
Symbolic Execution based Testing
An Application of Solvers

Vijay Ganesh
Affiliation: University of Waterloo

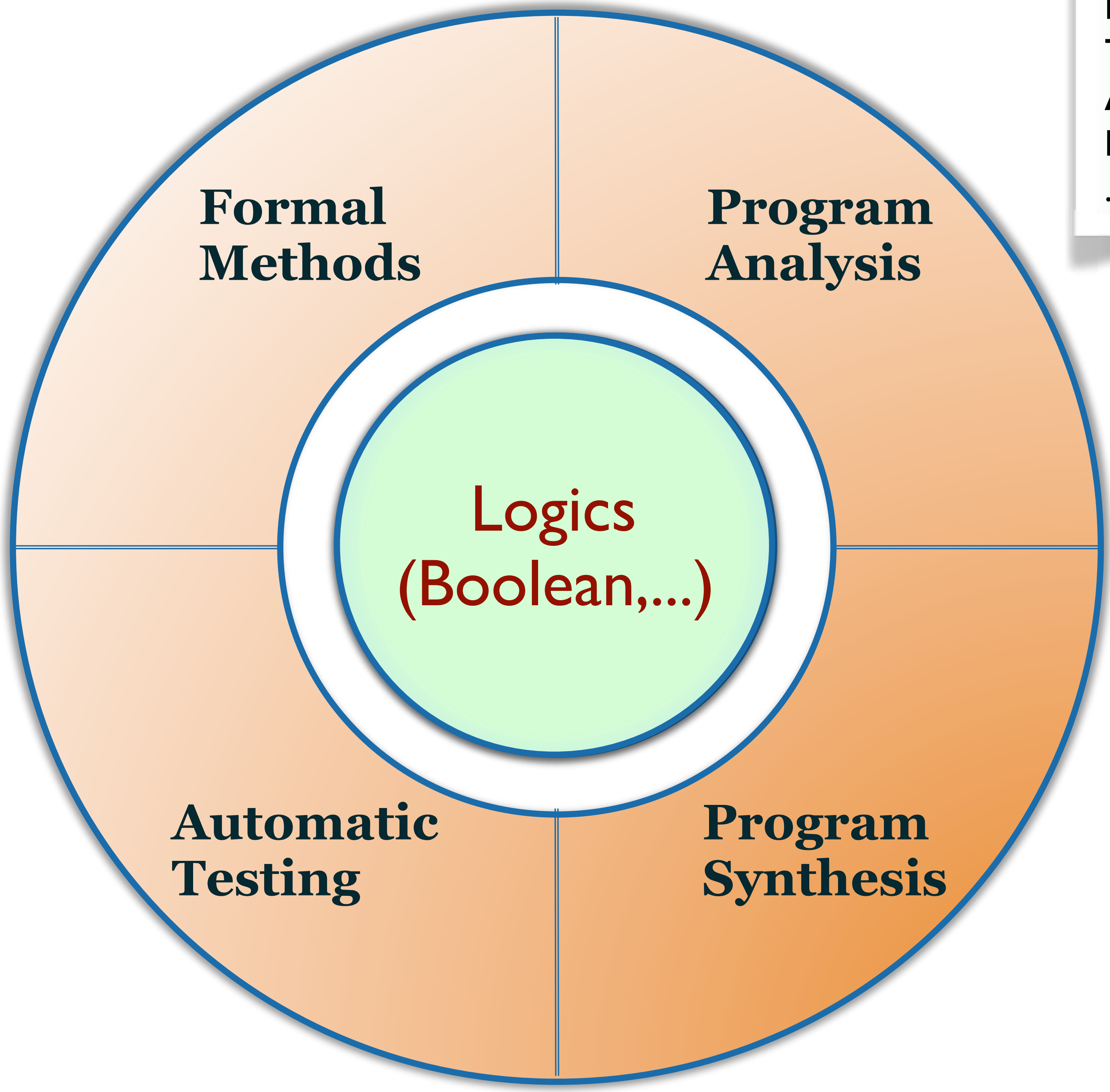
Goals of this Course

An introduction to SAT/SMT Solvers and Apps

- On the importance of logic in software engineering and security
- What are constraint solvers (Boolean SAT and SMT solvers)
- Symbolic execution + solvers: a powerful combination
- Dynamic symbolic testing (aka, concolic testing)
- Anatomy of modern CDCL solvers
- Conclusions

A Foundation for Software Engineering

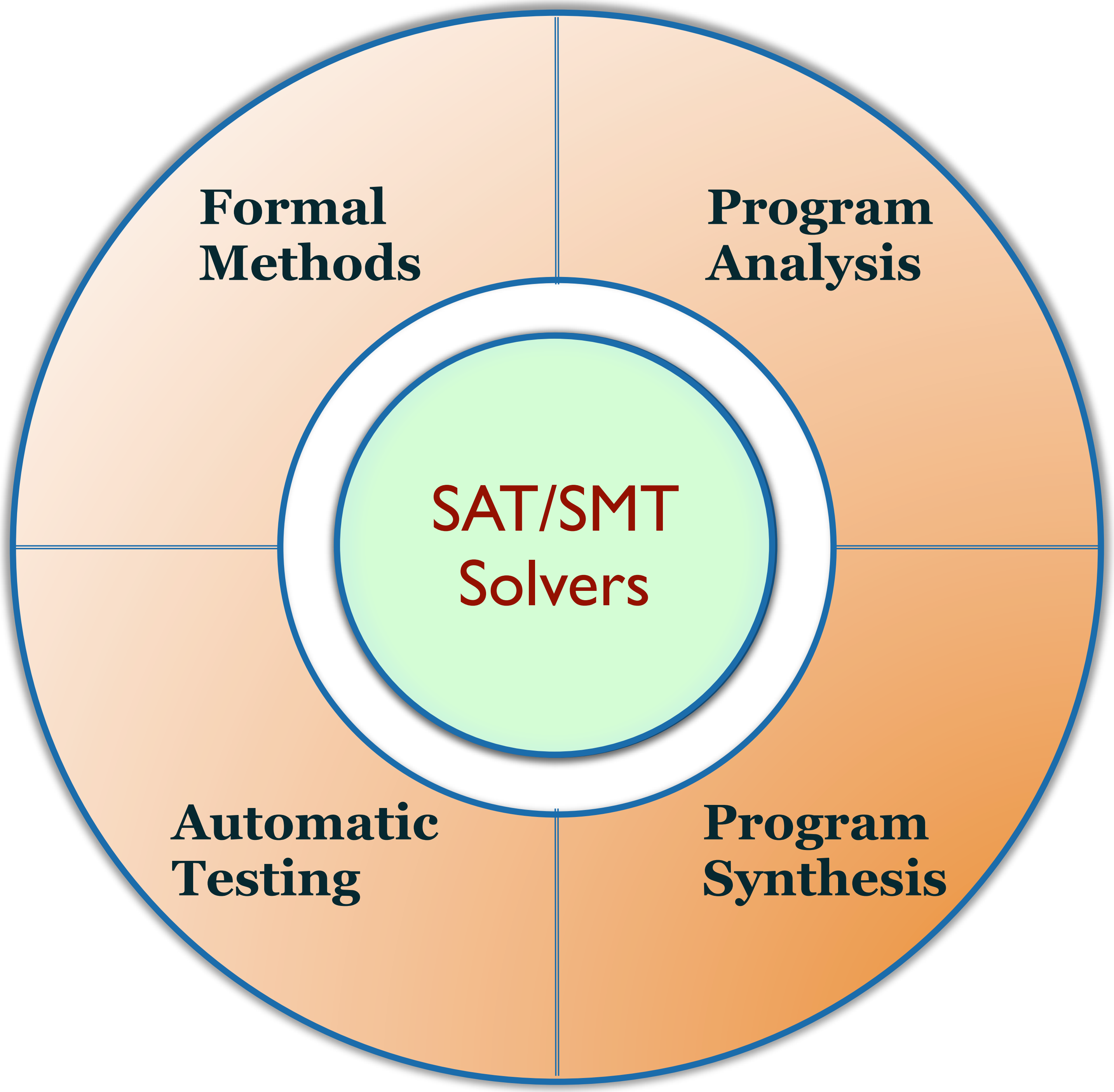
Logic Abstractions of Computation



Bob Floyd (1967)
Tony Hoare (1968,70)
Amir Pnueli (1977)
Ed Clarke (1982)
...

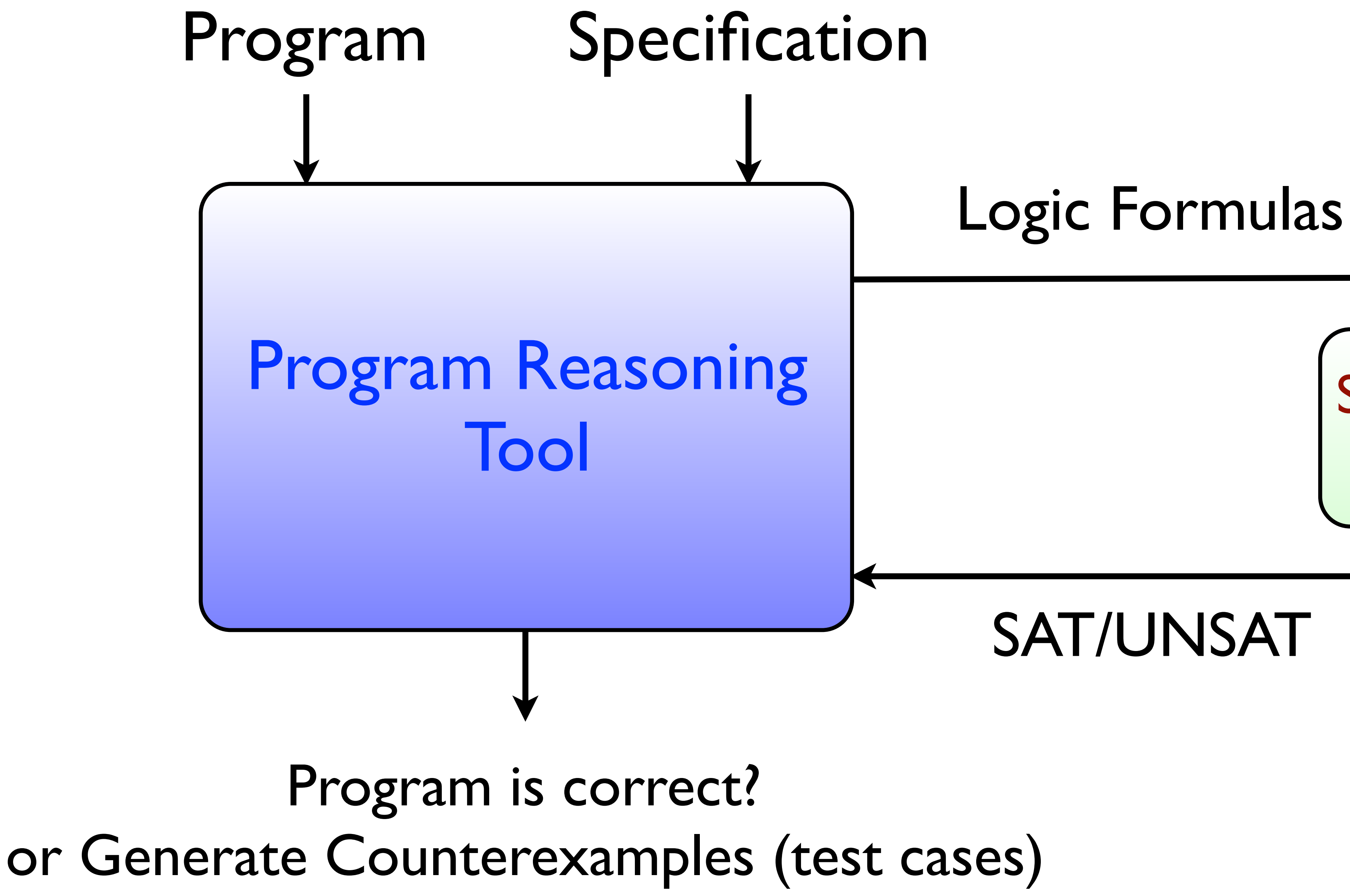
Software Engineering & SAT/SMT Solvers

An Indispensable Tactic for Any Strategy



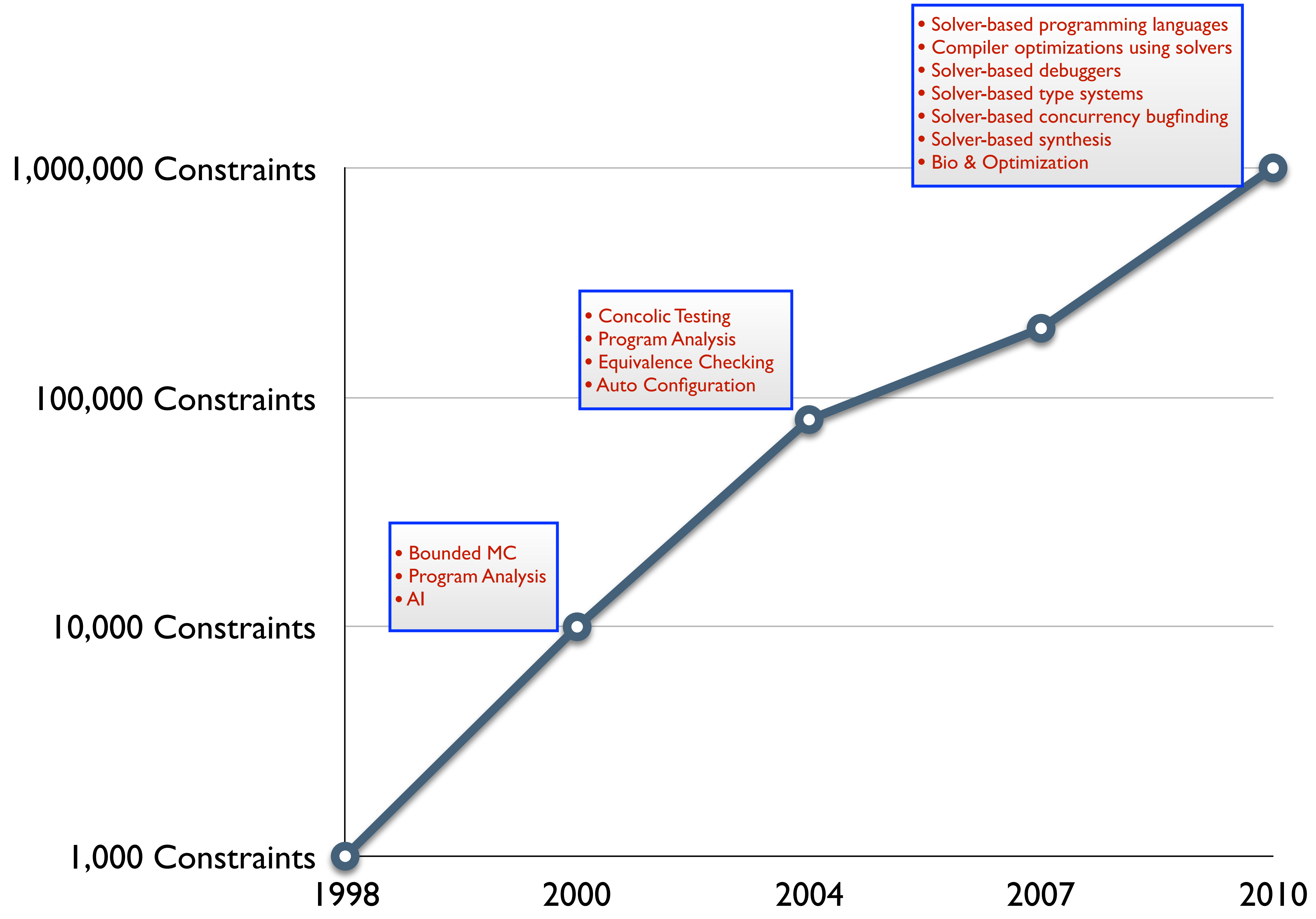
Software Engineering using Solvers

Engineering, Usability, Novelty

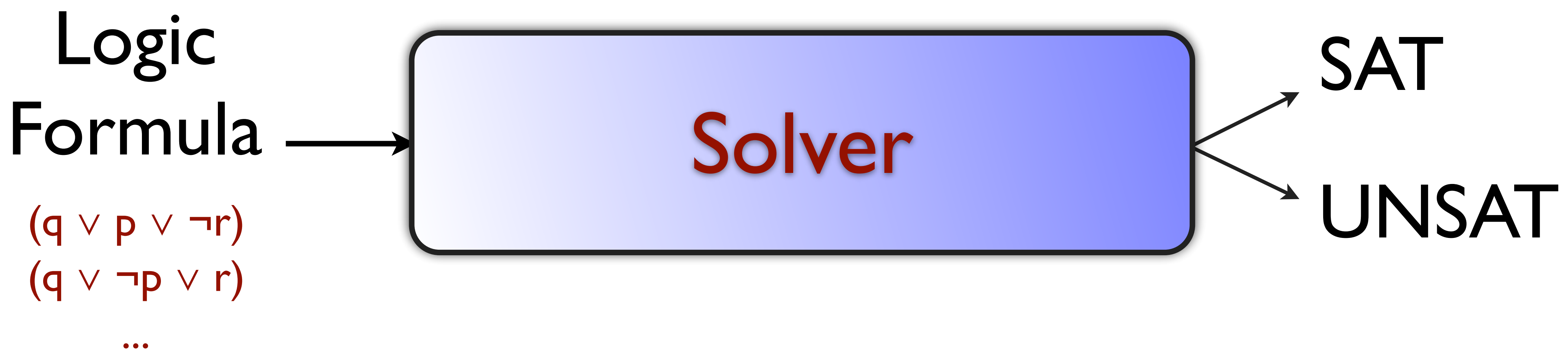


SAT/SMT Solver Research Story

A 1000x Improvement: Democratization of Logic



The SAT/SMT Problem



- Rich logics (Modular arithmetic, Arrays, Strings,...)
- NP-complete, PSPACE-complete,...
- Practical, scalable, usable, automatic
- Enable novel software reliability approaches

Lecture Outline

Points already covered

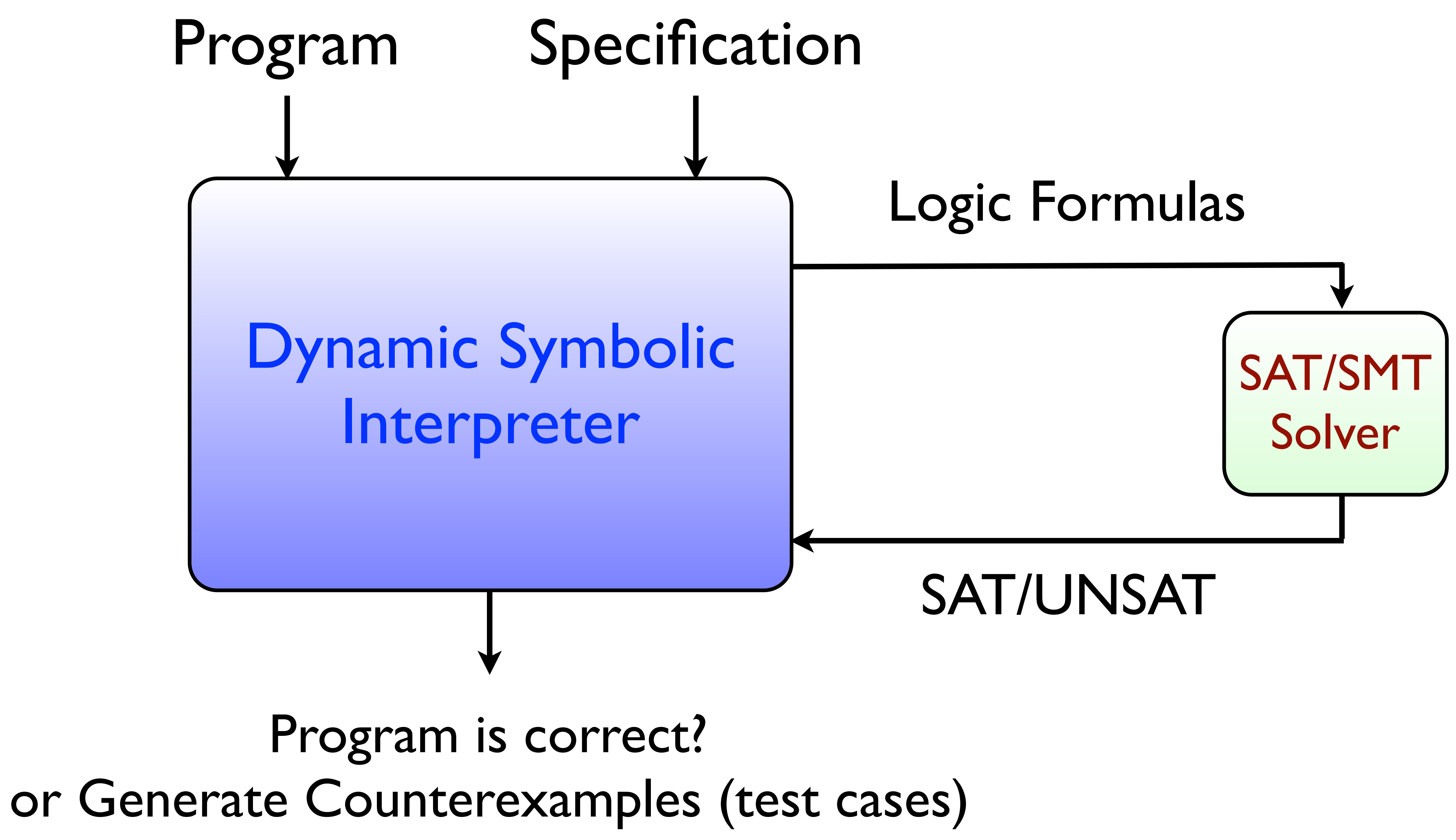
- ✓ Motivation for SAT/SMT solvers in software engineering
- ✓ High-level description of the SAT/SMT problem & logics
- ✓ Why you should care

Rest of the lecture

- Dynamic symbolic testing (aka concolic testing): A classic use of solvers
- Modern CDCL SAT solver architecture & techniques
- SAT/SMT-based applications
- Future of SAT/SMT solvers
- Some history (who, when,...) and references sprinkled throughout the talk

Dynamic Symbolic Testing

Symbolic/Concrete Execution + Solvers

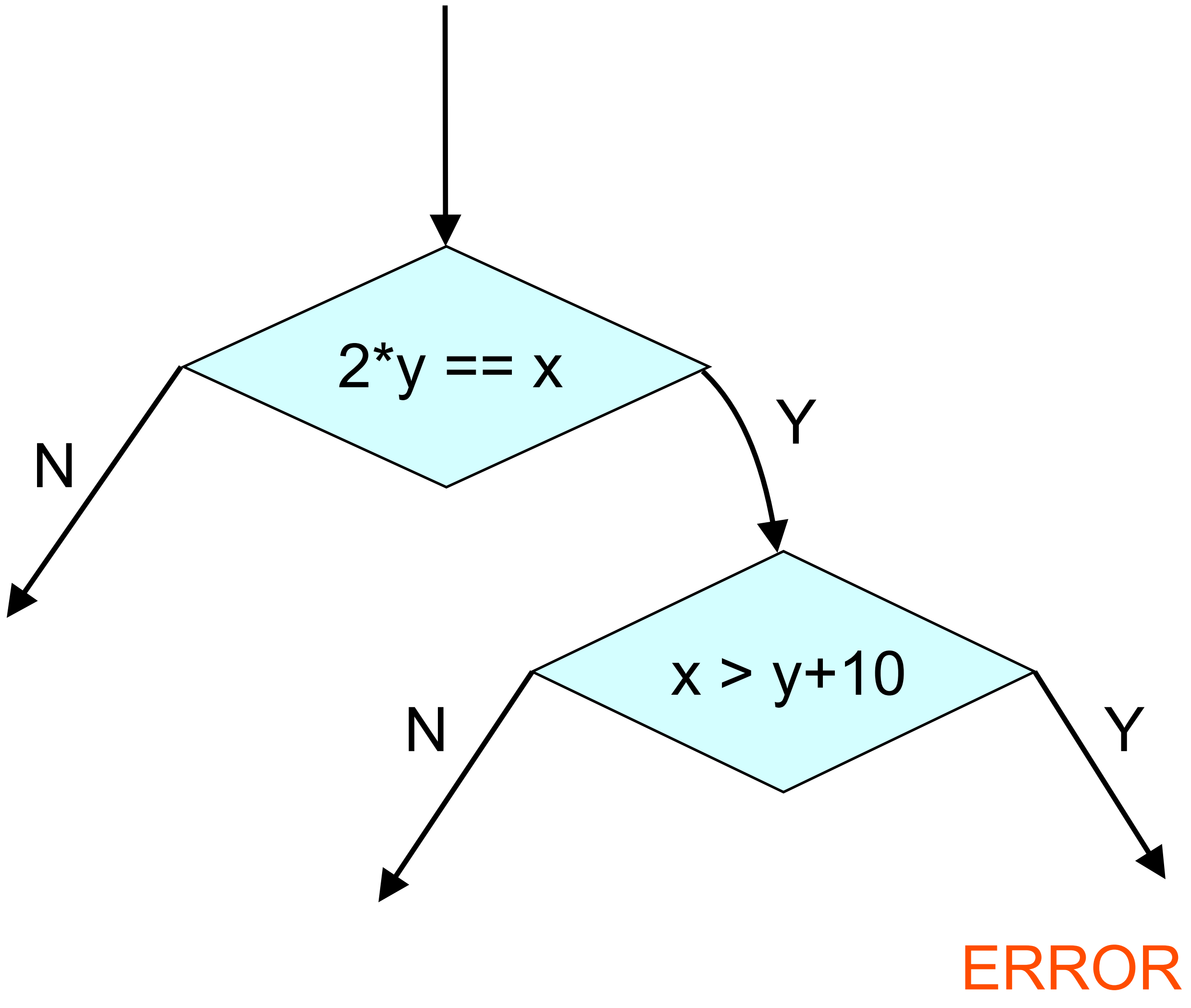


Concolic Testing: Example

```
int double (int v) {
     return 2*v;
}


void testme (int x, int y) {
     z = double (y);
     if (z == x) {
          if (x > y+10) {
               ERROR;
          }
     }
}
```

Concolic Testing: Example

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```



Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}


void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
	$x = x_0$, $y = y_0$	

Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}


void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
	$x = x_0$, $y = y_0$,	
	$z = 2^*y_0$	

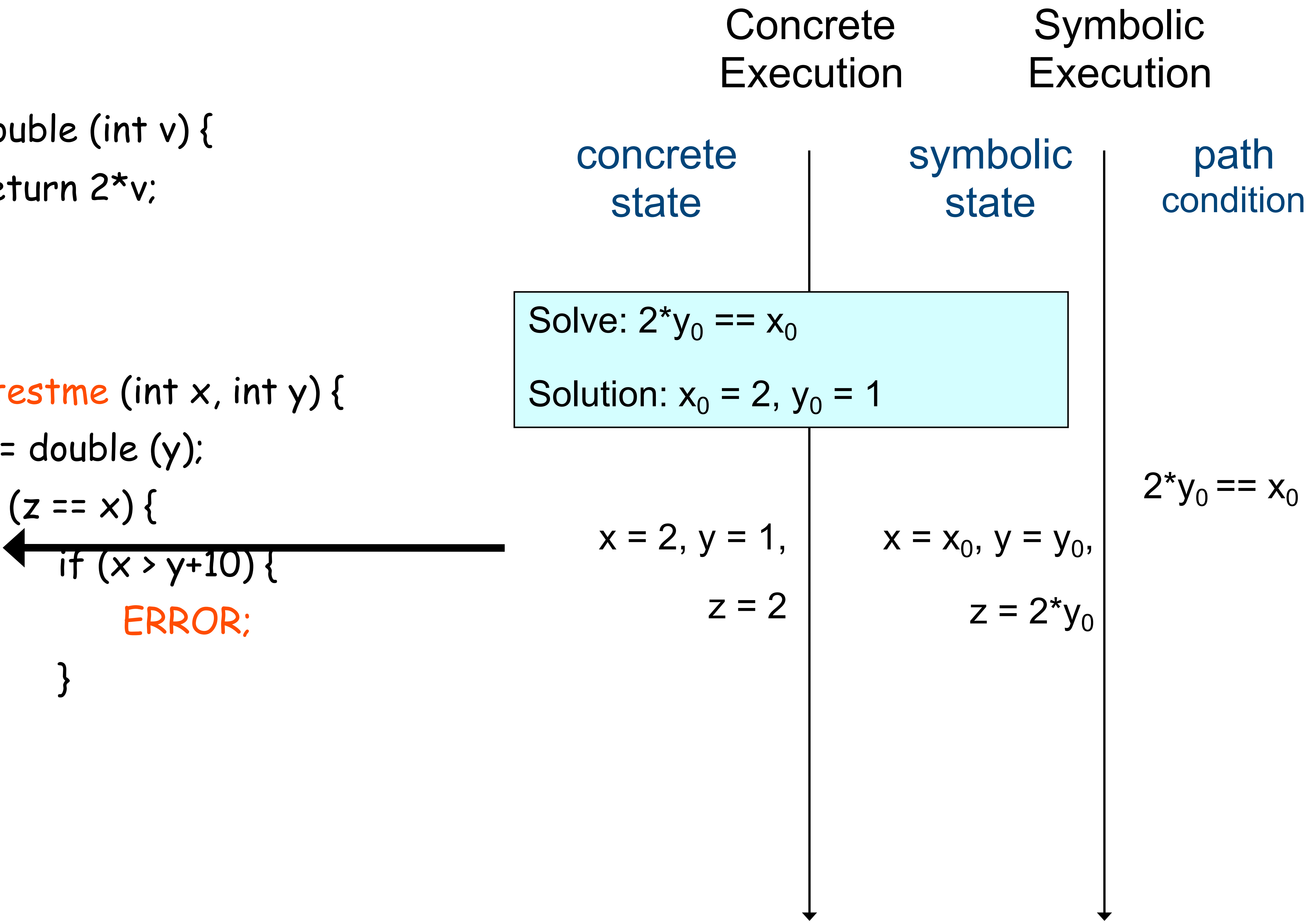
Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}


void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 22, y = 7, z = 14		$x = x_0$, $y = y_0$, $z = 2*y_0$	$2*y_0 \,!= x_0$

Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```



Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}


void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$2*y_0 == x_0$
		$x_0 < y_0+10$
$x = 2, y = 1, z = 2$	$x = x_0, y = y_0, z = 2*y_0$	

Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution | Symbolic Execution

concrete state | symbolic state | path condition

Solve: $(2*y_0 == x_0)$ **AND** $(x_0 > y_0 + 10)$

Solution: $x_0 = 30$, $y_0 = 15$

$2*y_0 == x_0$

$x_0 \cdot y_0+10$

$x = 2$, $y = 1$, $z = 2$

$x = x_0$, $y = y_0$, $z = 2*y_0$

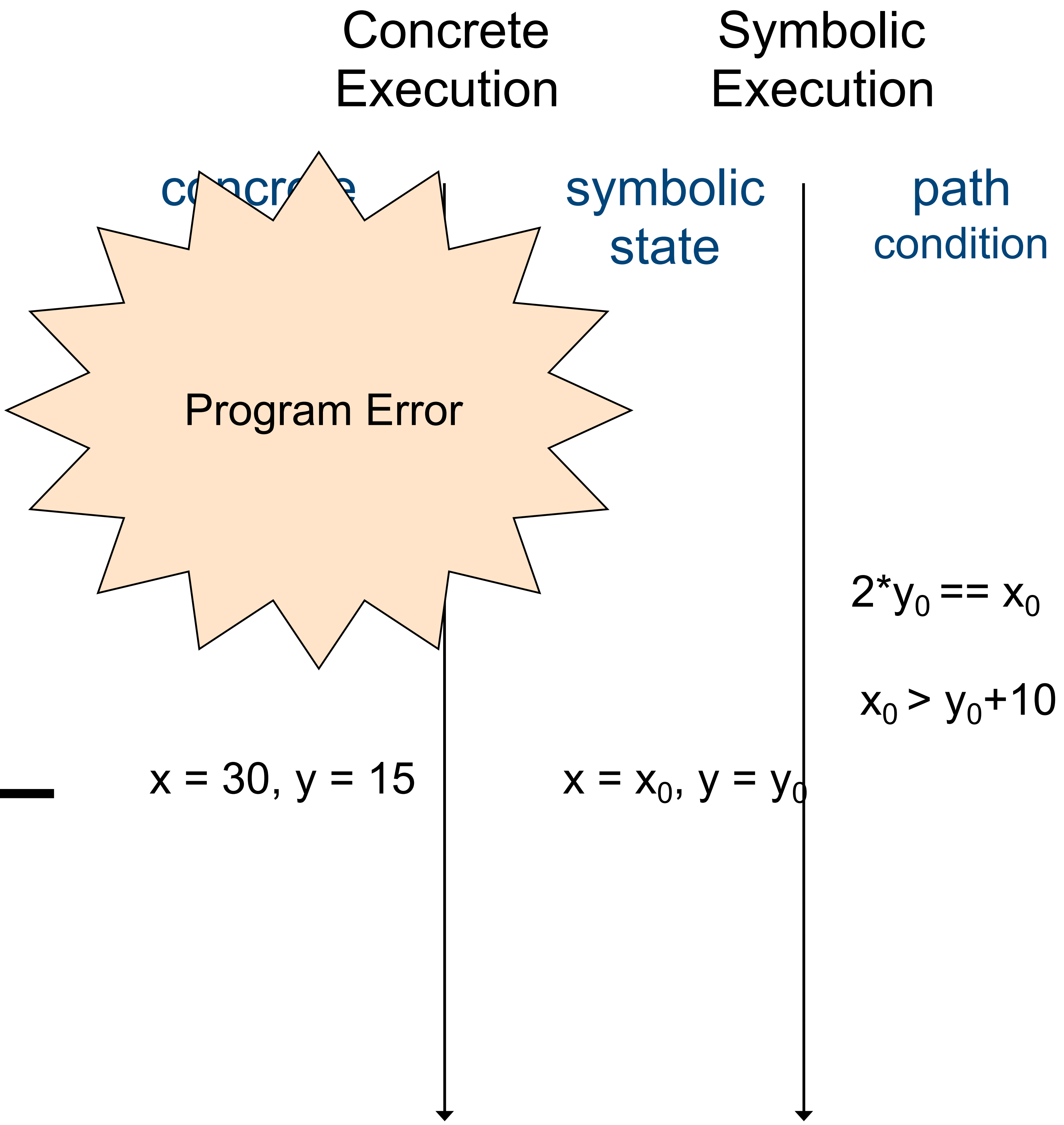
Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 30, y = 15$	$x = x_0, y = y_0$	

Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```



Concolic Testing Approach - Example 2

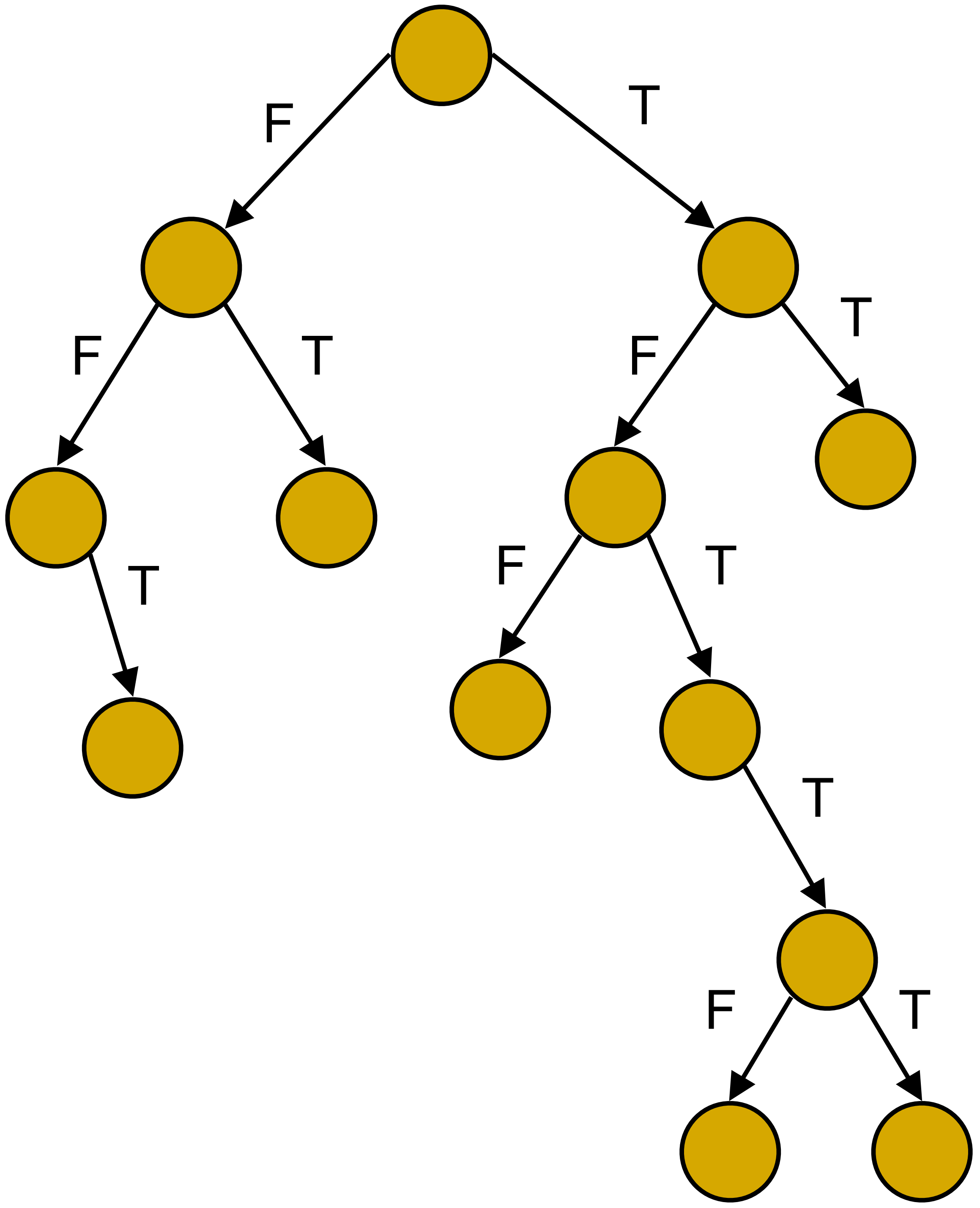
```
void testme (int x, int y) {
    if (hash(y) == x) {
      ERROR;
    }
}
```

Concolic Execution

concrete state	symbolic state	path condition
set y = 15, choose x randomly	$x = x_0$, $y = y_0$	hash(15) ==x
record hash(y)		
Run again by setting		
y = 15, x= hash(y)		

Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with valgrind to discover memory errors



Dynamic Symbolic Testing
Some History

- ✓ Symbolic execution for testing first proposed by Lori Clarke (1975)
  ACM SIGSOFT Outstanding Researcher Award 2012

- ✓ Follow up work by J.C. King (1976)

- ✓ Rediscovered/modified in the context of powerful solvers, analysis, and appropriate concretizations by independent groups

  - ✓ Patrice Godefroid and Koushik Sen (2005)

  - ✓ Dawson Engler et al. (2005)

  - ✓ Nicky Williams et al. (2004)

- ✓ Many follow up works by George Candea, Dawn Song, David Molnar,...

- ✓ Beyond testing: fault localization, repair, security,...

Dynamic Symbolic Testing and Analysis Tools

- ✓ KLEE: the most well-known open-source symbolic execution tool (web: https://klee.github.io/)

- ✓ SAGE: Microsoft's dynamic symbolic execution tool (closed-source)

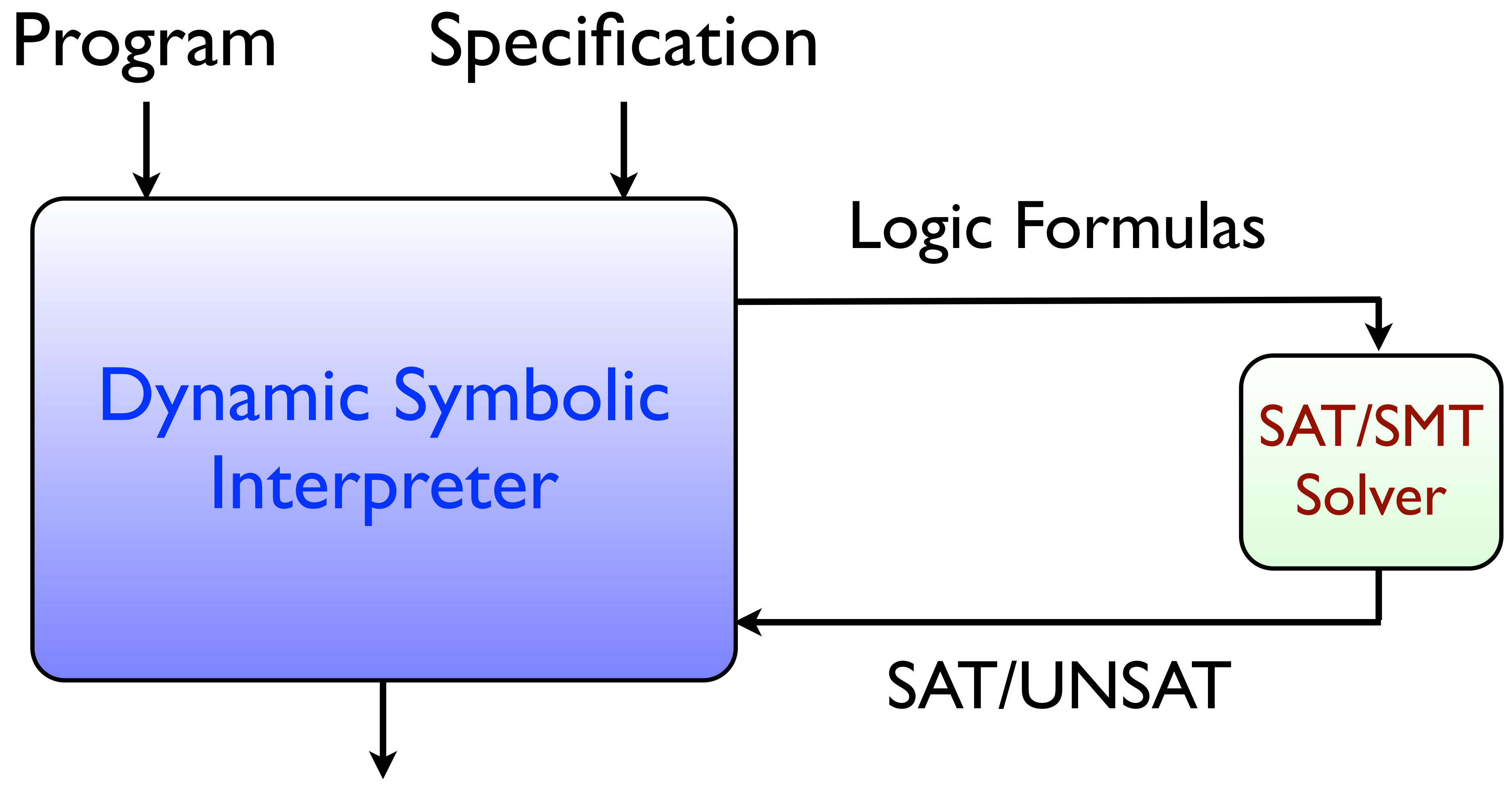
- ✓ S2E: symbolic engine that is built on top of KLEE, but works on binaries (web: http://dslab.epfl.ch/pubs/s2e-tocs.pdf)

- ✓ Jalangi: dynamic symbolic analysis tool for JavaScript (web: https://github.com/Samsung/jalangi2)

- ✓ Other tools: Triton, BAP, Bitblaze, Webblaze

Dynamic Symbolic Testing

Symbolic/Concrete Execution + Solvers



Program is correct?
or Generate Counterexamples (test cases)

Lecture Outline

Points already covered

- ✓ Motivation for SAT/SMT solvers in software engineering
- ✓ High-level description of the SAT/SMT problem & logics
- ✓ Why you should care
- ✓ Dynamic symbolic testing (sometime also called concolic testing)

Rest of the lecture

- Modern CDCL SAT solver architecture & techniques
- An overview of programmatic SAT solvers
- Some history (who, when,...) and references sprinkled throughout the talk

The Boolean SAT Problem
Basic Definitions and Format

A **literal** p is a Boolean variable x or its negation $\neg x$.

A **clause** C is a disjunction of literals: $x_2 \vee \neg x_{41} \vee x_{15}$

A **CNF** is a conjunction of clauses: $(x_2 \vee \neg x_1 \vee x_5) \wedge (x_6 \vee \neg x_2) \wedge (x_3 \vee \neg x_4 \vee \neg x_6)$

All Boolean formulas assumed to be in **CNF**

Assignment is a mapping (binding) from variables to Boolean values **(True, False).**

A **unit clause** C is a clause with a single unbound literal

The **SAT-problem** is:

Find an assignment s.t. each input clause has a true literal (aka input formula has a solution or is SAT)

OR establish input formula has no solution (aka input formula is UNSAT)

The Input formula is represented in **DIMACS Format:**

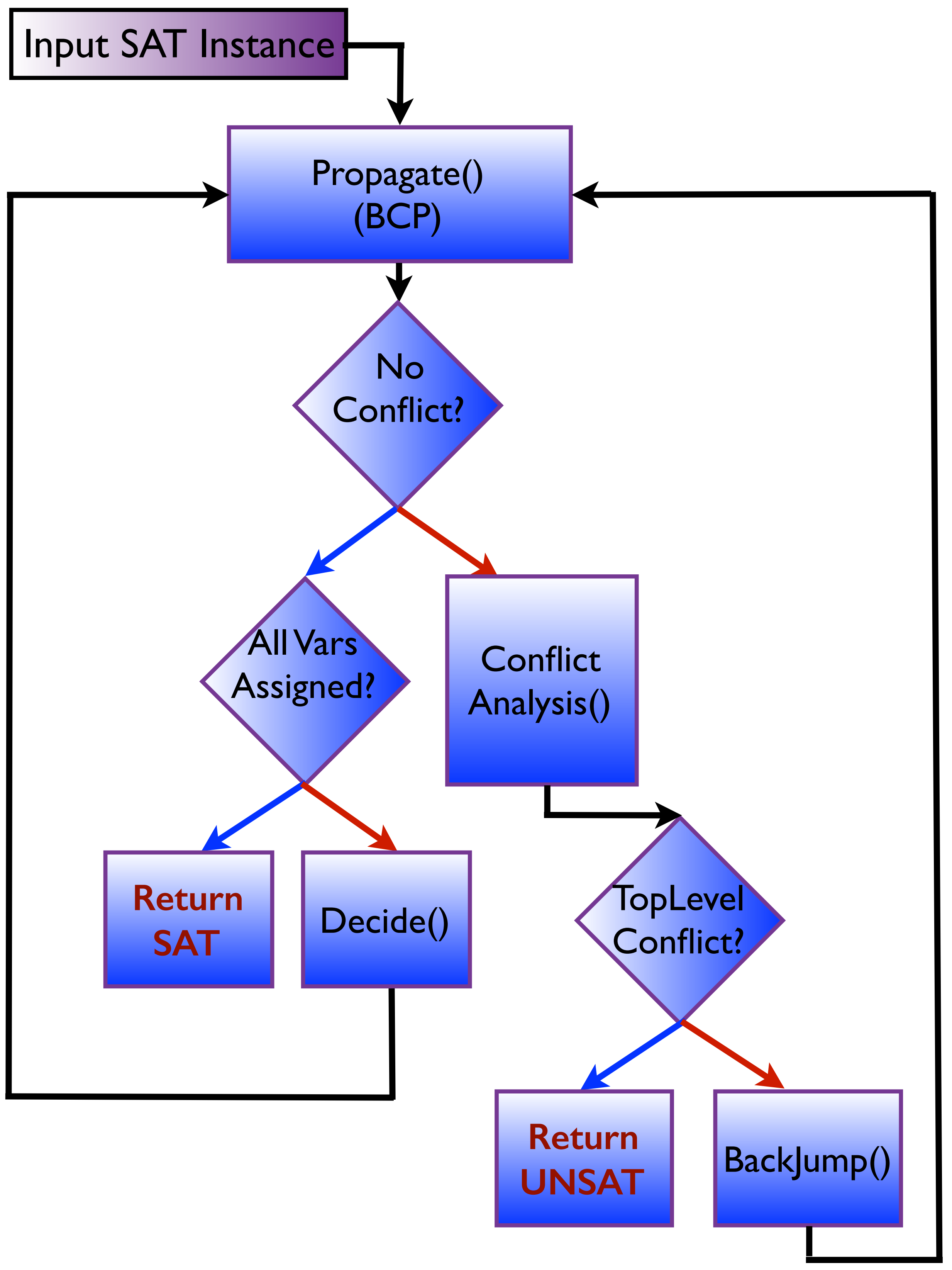
```
c DIMACS
p cnf 6 3
2 -1 5 0
6 -2 0
3 -4 -6 0
```

DPLL SAT Solver Architecture
The Basic Solver

```
DPLL($\Theta_{cnf}$, assign) {

    Propagate unit clauses;

    if "conflict": return FALSE;

    if "complete assign": return TRUE;

    "pick decision variable x";

    return
            DPLL($\Theta_{cnf}|_{x=0}$, assign[x=0])
         || DPLL($\Theta_{cnf}|_{x=1}$, assign[x=1]);
}
```

- Propagate (Boolean Constant Propagation):
  - Propagate inferences due to unit clauses
  - Most time in solving goes into this
- Detect Conflict:
  - Conflict: partial assignment is not satisfying
- Decide (Branch):
  - Choose a variable & assign some value
- Backtracking:
  - Implicitly done by the recursion

Modern CDCL SAT Solver Architecture
Key Steps and Data-structures



Key steps

- Decide()
- Propagate()
  (BCP: Boolean constraint propagation)
- Conflict analysis and learning()
- Backjump()
- Forget()
- Restart()

CDCL: Conflict-Driven Clause-Learning

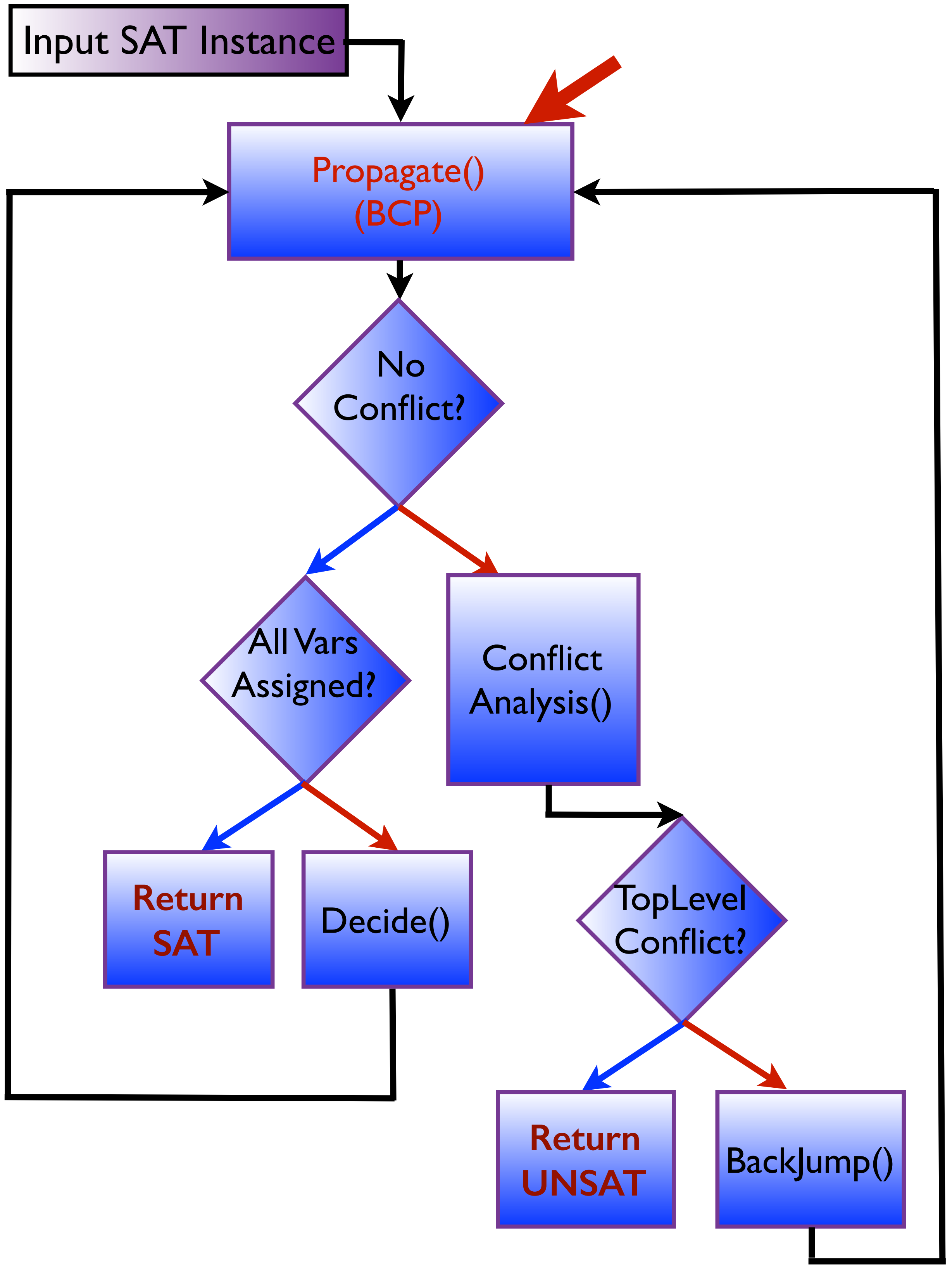
- Conflict analysis is a key step
- Results in learning a conflict clause
- Prunes the search space

Key data-structures (State):

- Stack or trail of partial assignments (AT)
- Input clause database
- Conflict clause database
- Conflict graph
- Decision level (DL) of a variable

Modern CDCL SAT Solver Architecture

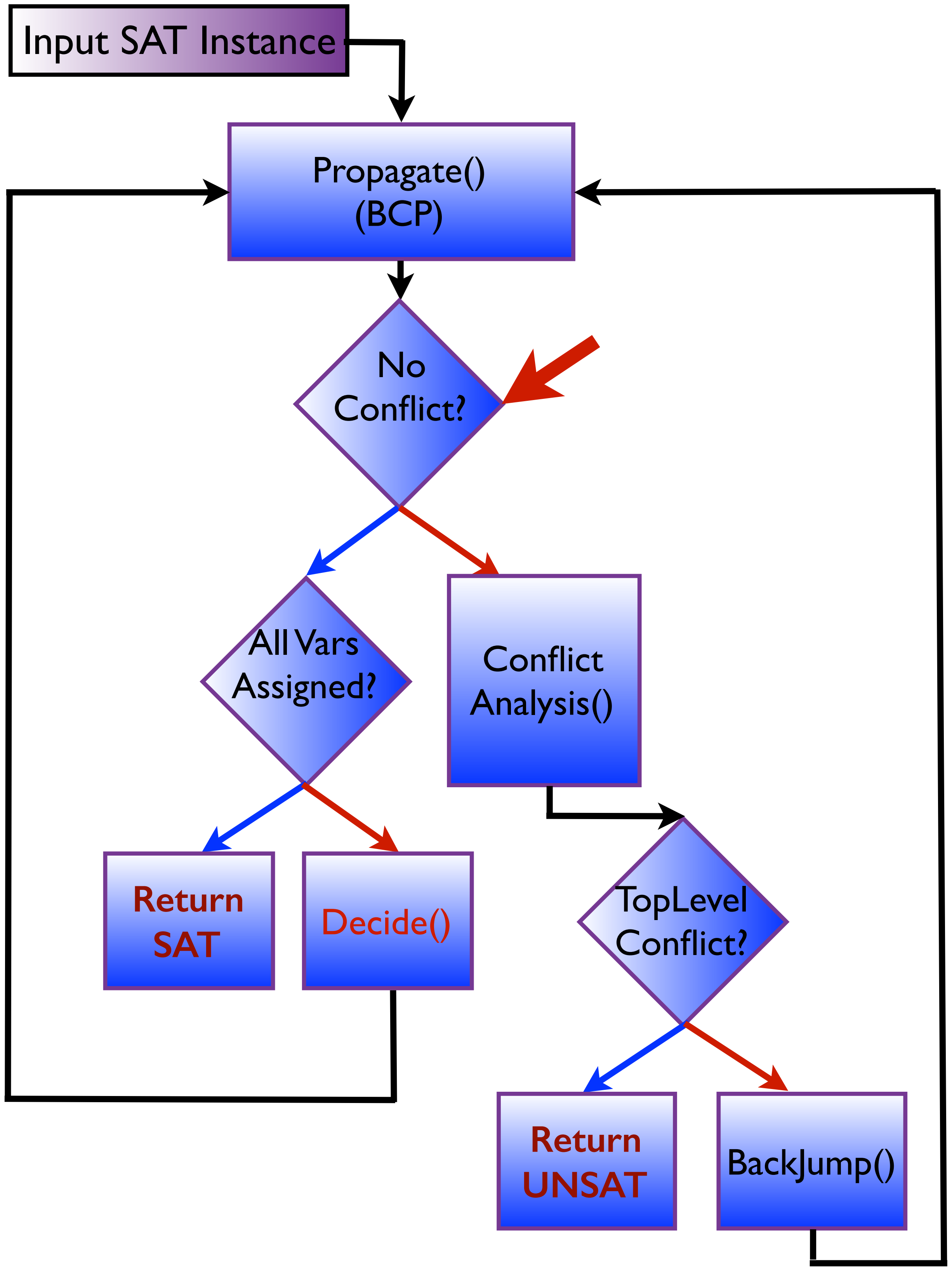
Propagate(), Decide(), Analyze/Learn(), BackJump()



- Propagate (Boolean Constant Propagation):
  - Propagate inferences due to unit clauses
  - Most time in solving goes into this

Modern CDCL SAT Solver Architecture

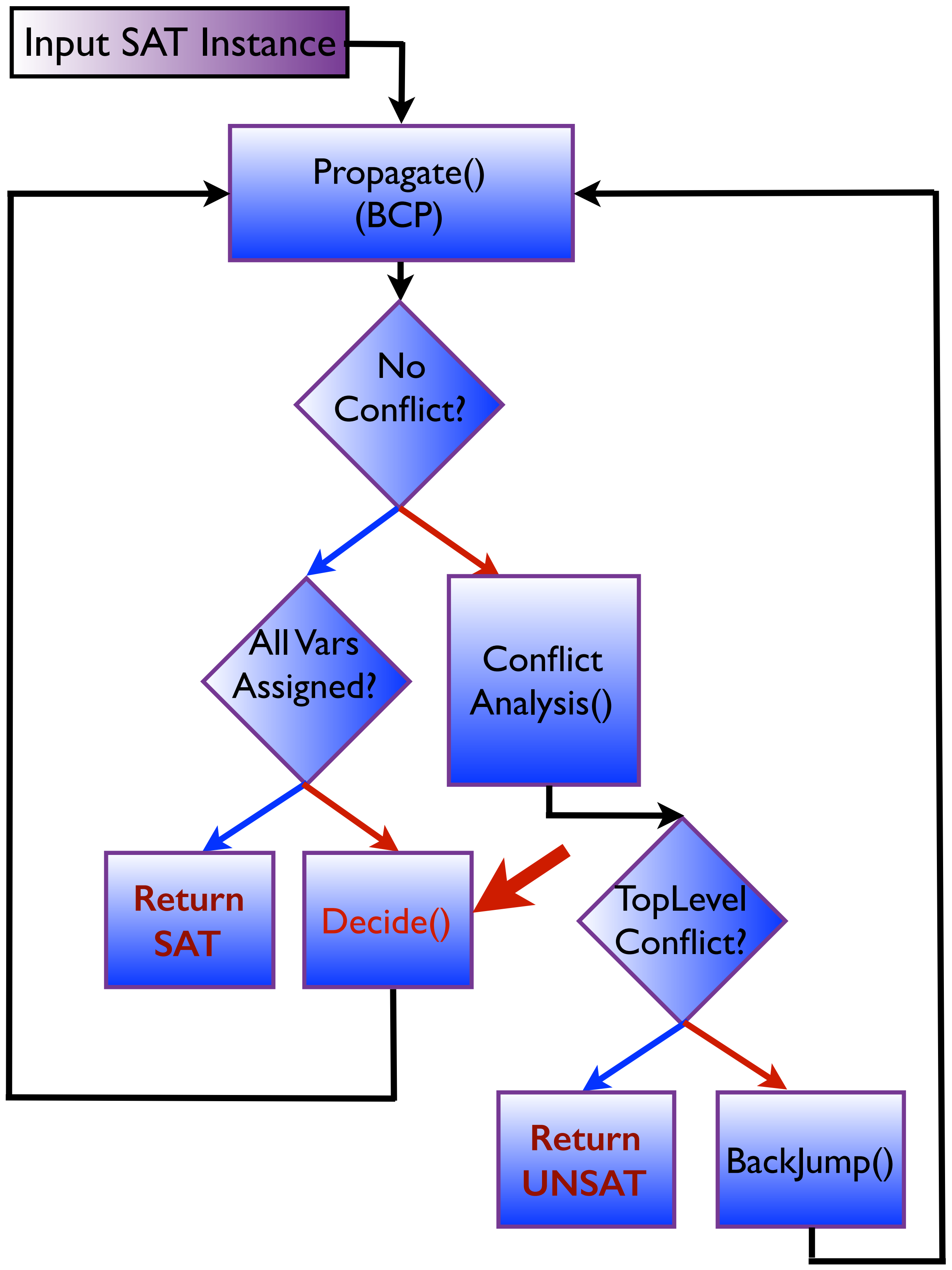
Propagate(), Decide(), Analyze/Learn(), BackJump()



- Propagate (Boolean Constant Propagation):
  - Propagate inferences due to unit clauses
  - Most time in solving goes into this
- Detect Conflict?
  - Conflict: partial assignment is not satisfying

Modern CDCL SAT Solver Architecture

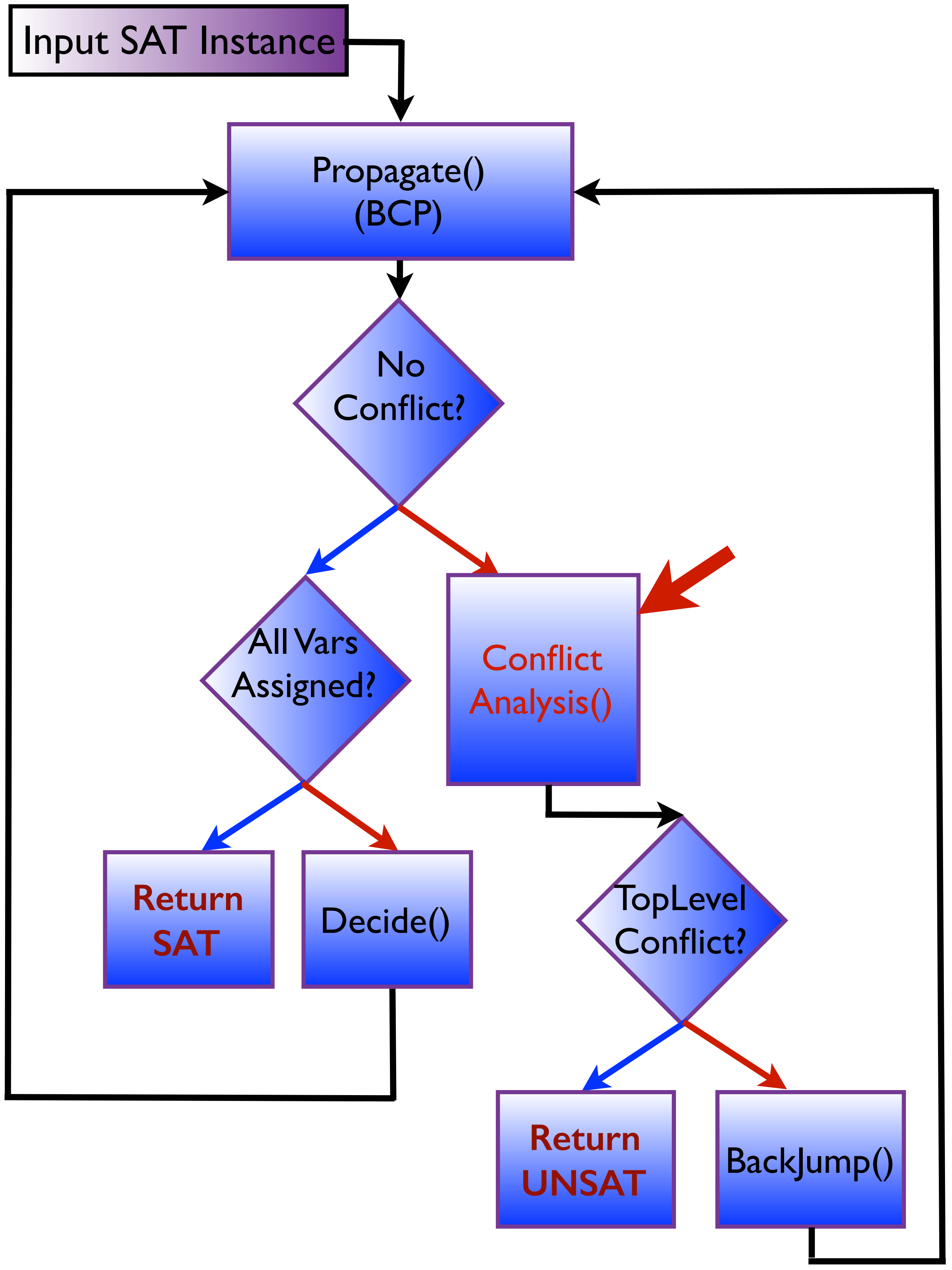
Propagate(), Decide(), Analyze/Learn(), BackJump()



- Propagate (Boolean Constant Propagation):
  - Propagate inferences due to unit clauses
  - Most time in solving goes into this
- Detect Conflict?
  - Conflict: partial assignment is not satisfying
- Decide (Branch):
  - Choose a variable & assign some value (decision)
  - Basic mechanism to do search
  - Imposes dynamic variable order
  - Decision Level (DL): variable ⇒ natural number

Modern CDCL SAT Solver Architecture

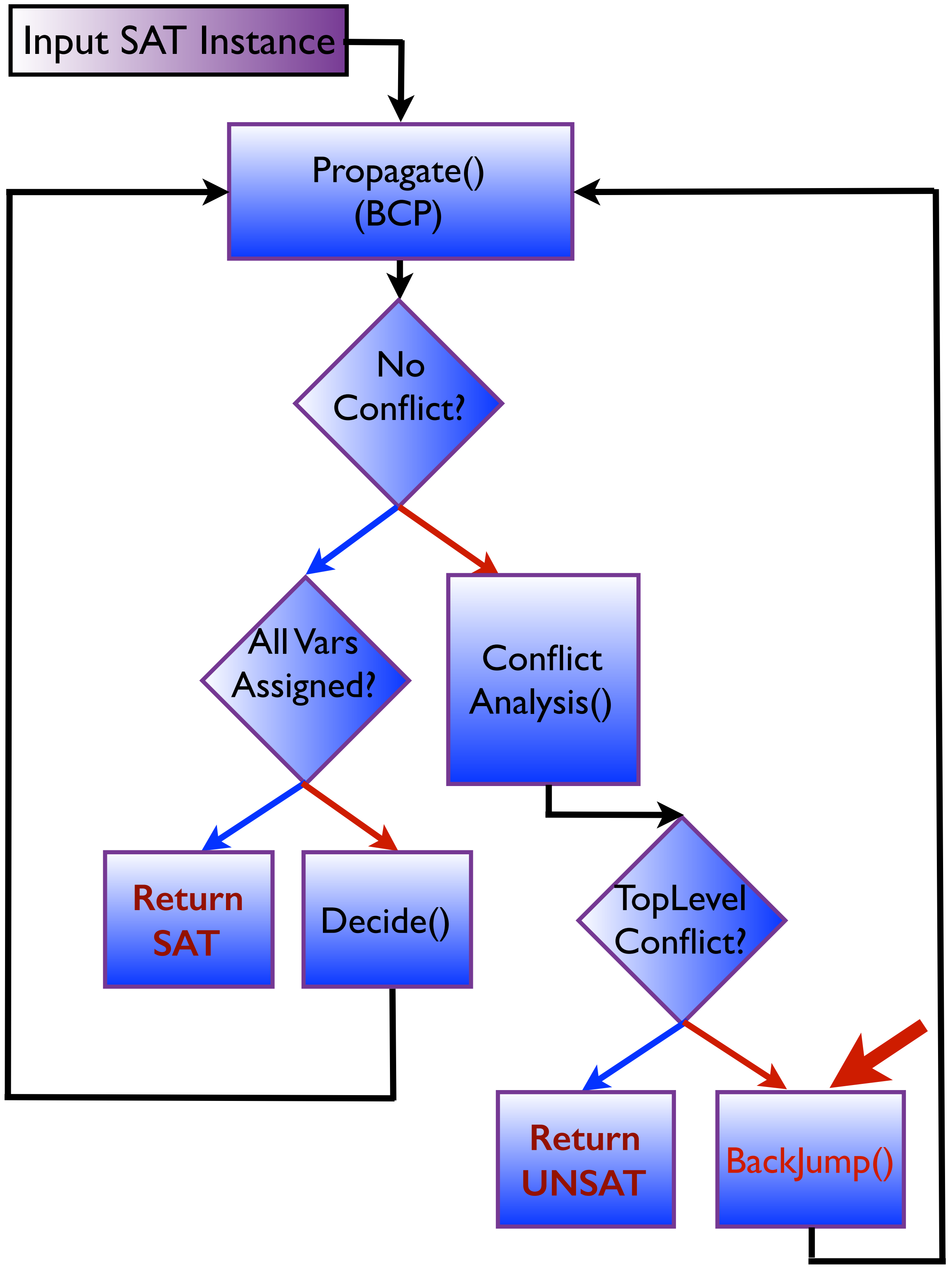
Propagate(), Decide(), Analyze/Learn(), BackJump()



- Propagate:
  - Propagate inferences due to unit clauses
  - Most time in solving goes into this
- Detect Conflict?
  - Conflict: partial assignment is not satisfying
- Decide (Branch):
  - Choose a variable & assign some value (decision)
  - Each decision is a decision level
  - Imposes dynamic variable order
  - Decision Level (DL): variable ⇒ natural number
- Conflict analysis and clause learning:
  - Compute assignments that lead to conflict (analysis)
  - Construct conflict clause blocks the non-satisfying & a large set of other ‘no-good’ assignments (learning)
  - Marques-Silva & Sakallah (1996)

Modern CDCL SAT Solver Architecture

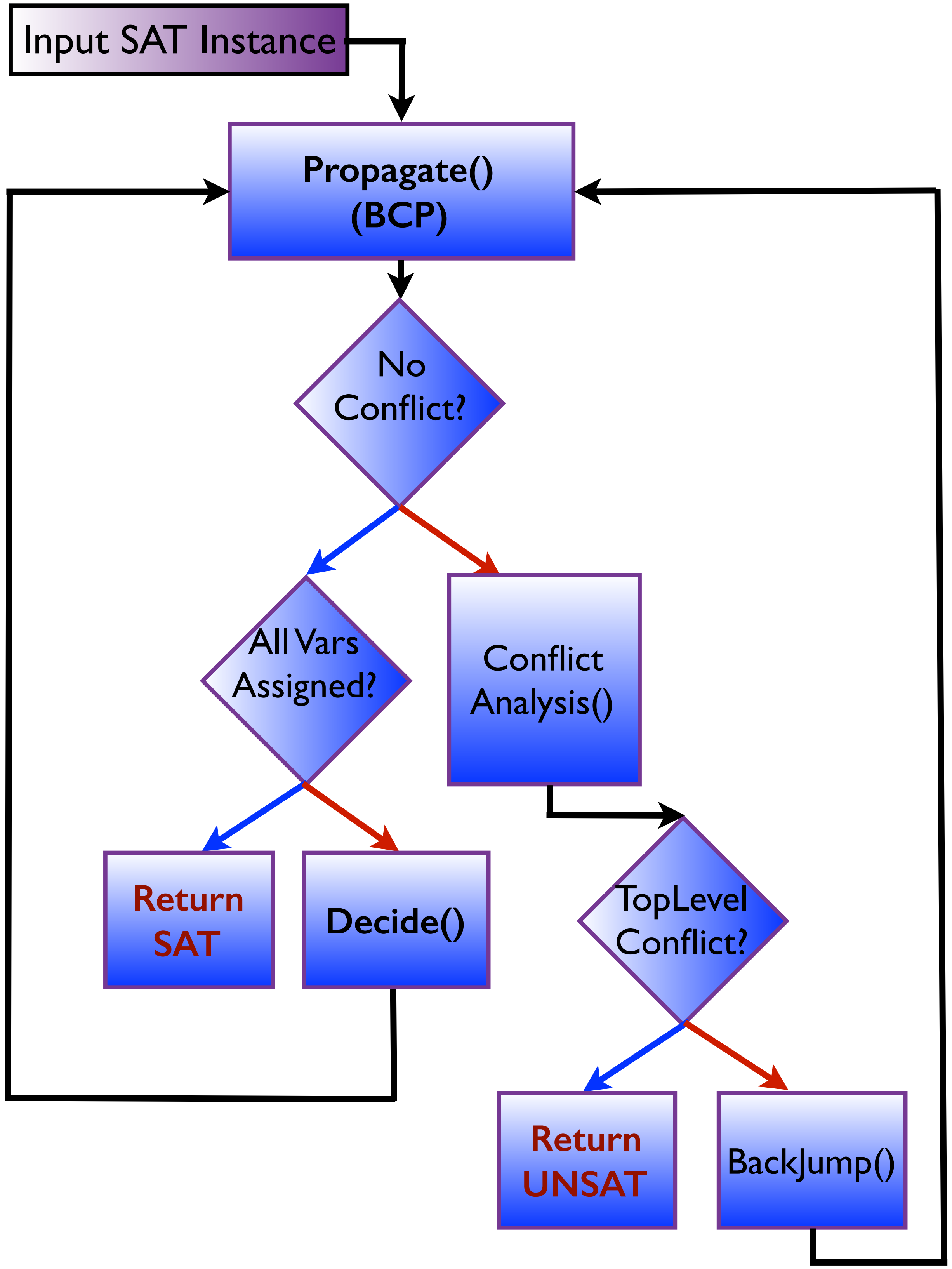
Propagate(), Decide(), Analyze/Learn(), BackJump()

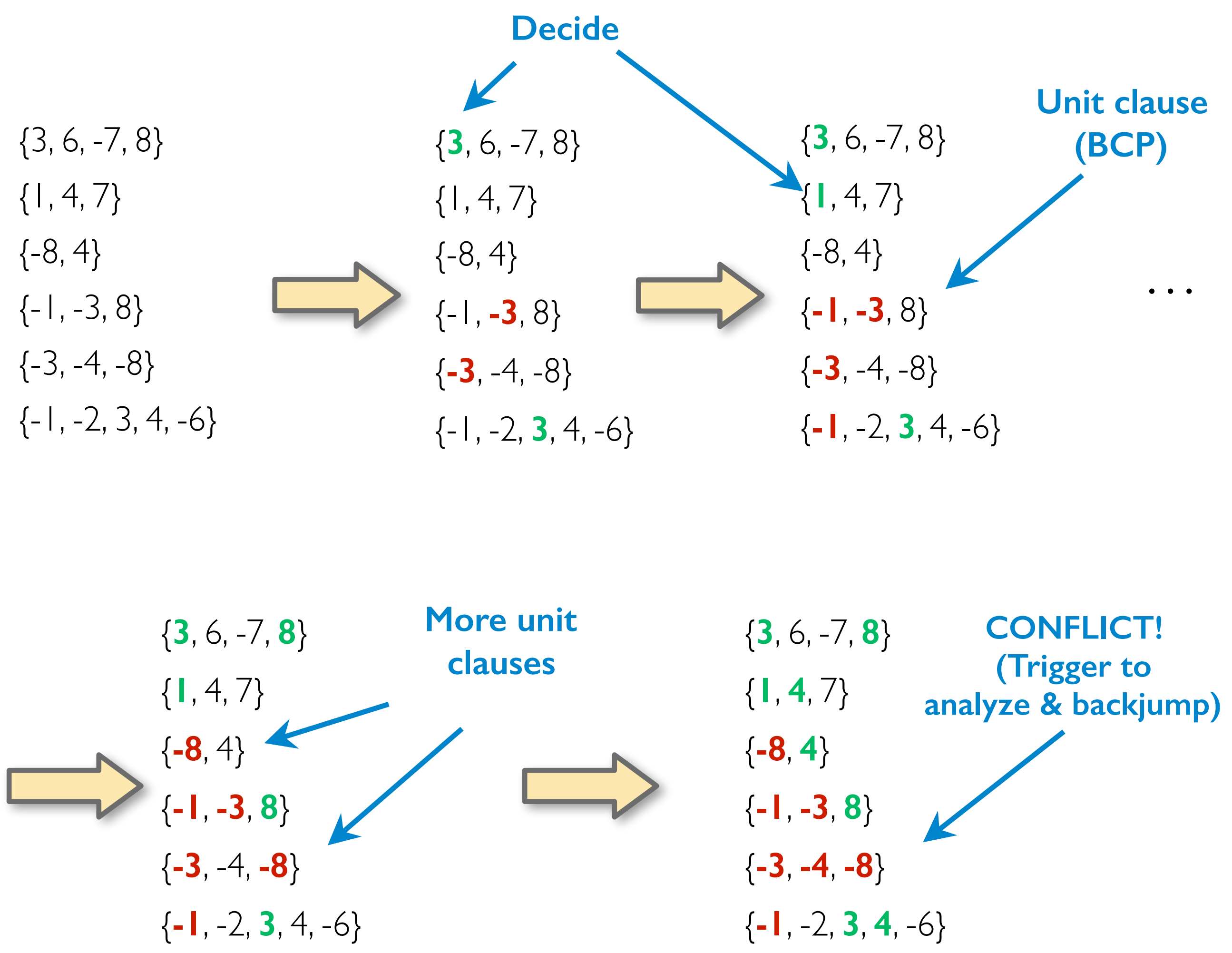


- Propagate:
  - Propagate inferences due to unit clauses
  - Most time in solving goes into this
- Detect Conflict?
  - Conflict: partial assignment is not satisfying
- Decide:
  - Choose a variable & assign some value (decision)
  - Each decision is a decision level
  - Imposes dynamic variable order
  - Decision Level (DL): variable ⇒ natural number
- Conflict analysis and clause learning:
  - Compute assignments that lead to conflict (analysis)
  - Construct conflict clause blocks the non-satisfying & a large set of other 'no-good' assignments (learning)
  - Marques-Silva & Sakallah (1996)
- Conflict-driven BackJump:
  - Undo the decision(s) that caused no-good assignment
  - Assign 'decision variables' different values
  - Go back several decision levels
  - Backjump: Marques-Silva, Sakallah (1999)
  - Backtrack: Davis, Putnam, Loveland, Logemann (1962)

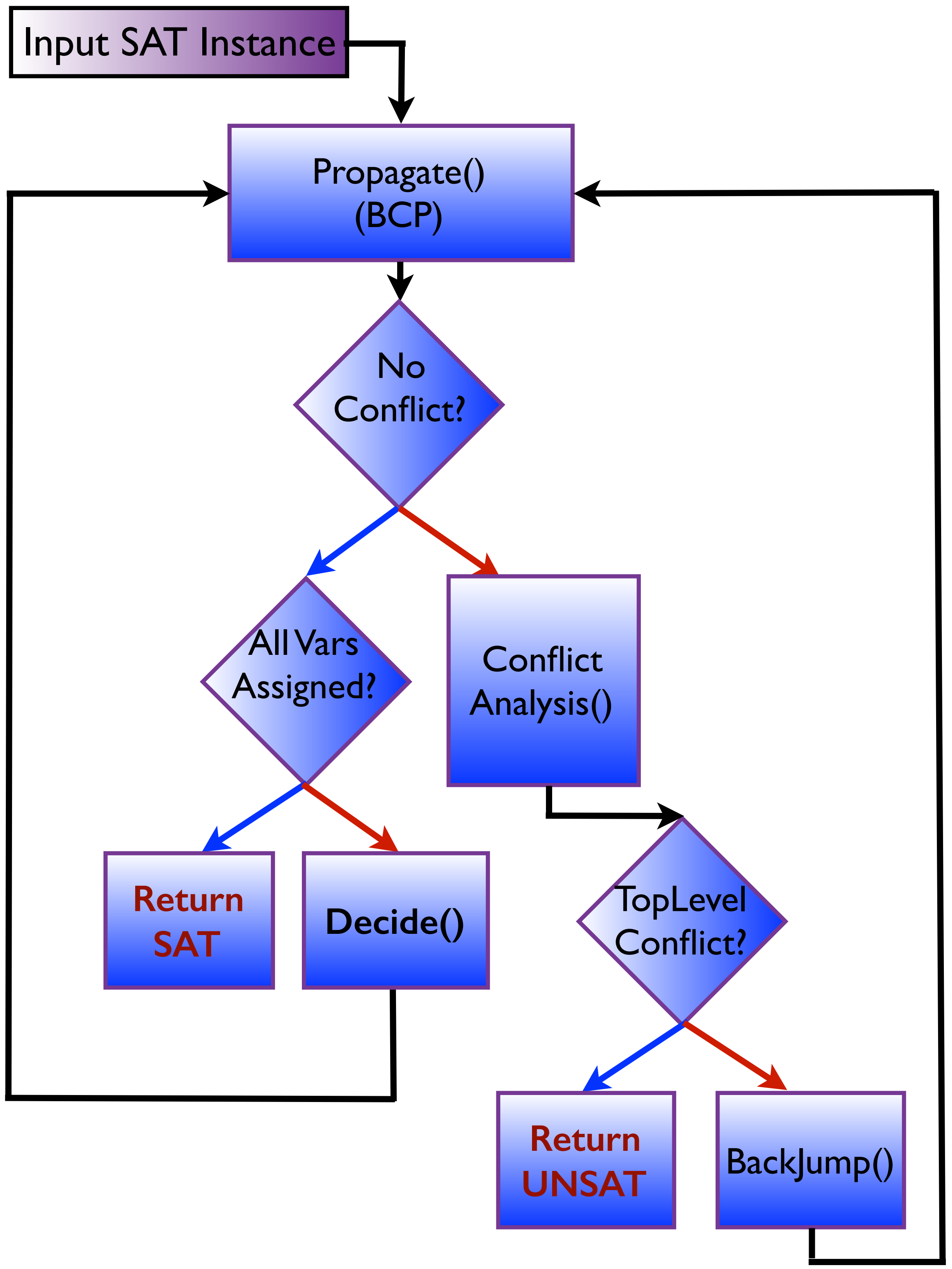
Modern CDCL SAT Solver Architecture

Propagate(), Decide(), Analyze/Learn(), BackJump()





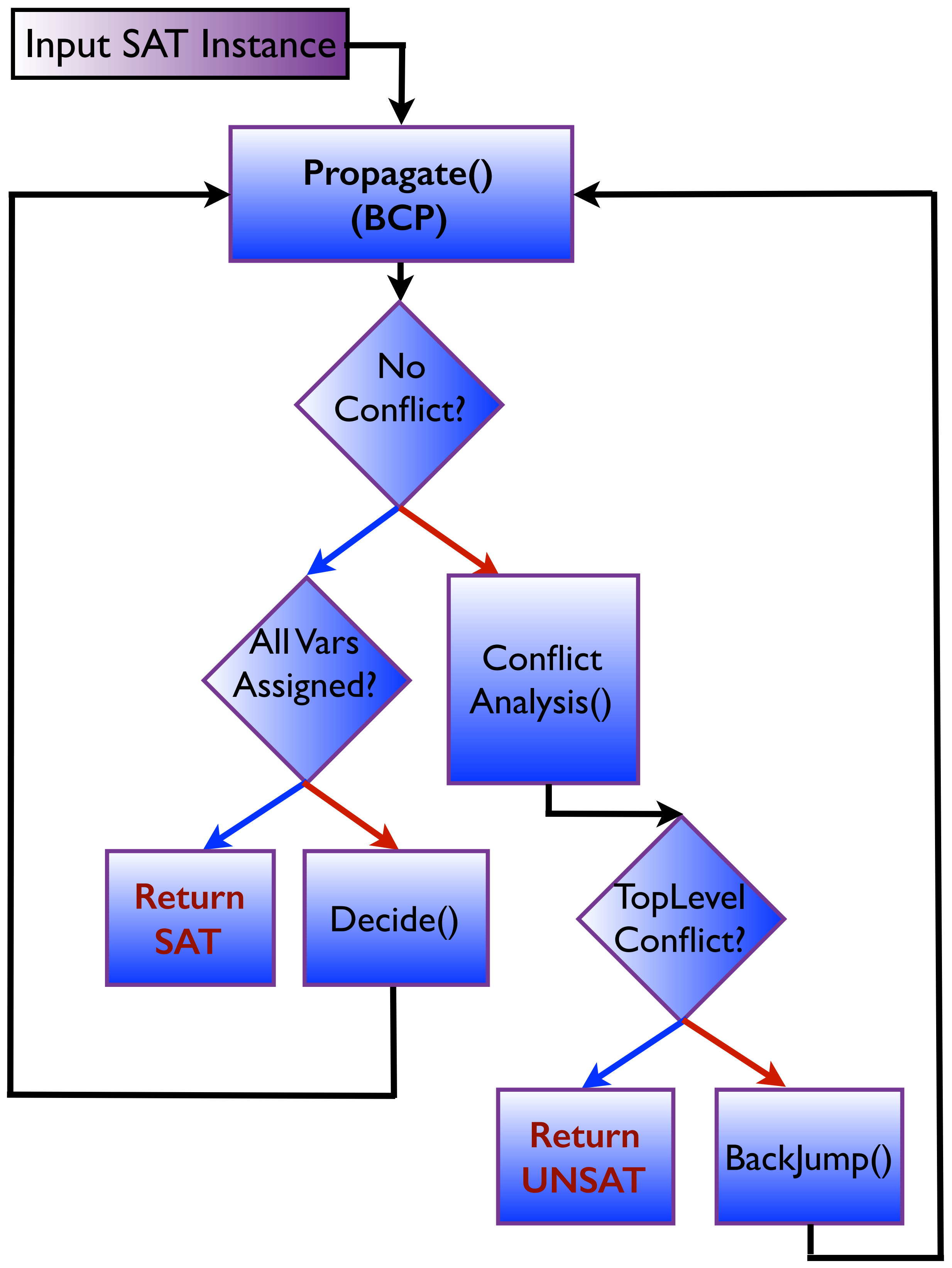
Modern CDCL SAT Solver Architecture
Decide() Details: VSIDS Heuristic



- Decide() or Branching():
  - Choose a variable & assign some value (decision)
  - Imposes dynamic variable order (Malik et al. 2001)
- How to choose a variable:
  - VSIDS heuristics
  - Each variable has an activity
  - Activity is bumped additively, if variable occurs in conflict clause
  - Activity of all variables is decayed by multiplying by const < 1
  - Next decision variable is the variable with highest activity
  - Over time, truly important variables get high activity
  - This is pure magic, and seems to work for many problems

Modern CDCL SAT Solver Architecture

Propagate() Details: Two-watched Literal Scheme



Decide

Unit clause (BCP)

{3, 6, -7, 8}	{3, 6, -7, 8}	{3, 6, -7, 8}
{1, 4, 7}	{1, 4, 7}	{1, 4, 7}
{-8, 4}	{-8, 4}	{-8, 4}
{-1, -3, 8}	{-1, -3, 8}	{-1, -3, 8}
{-3, -4, -8}	{-3, -4, -8}	{-3, -4, -8}
{-1, -2, 3, 4, -6}	{-1, -2, 3, 4, -6}	{-1, -2, 3, 4, -6}

. . .

Watched Literal	Watcher List
-1	{-1, -3,
-3	{-1, -3,
...	...

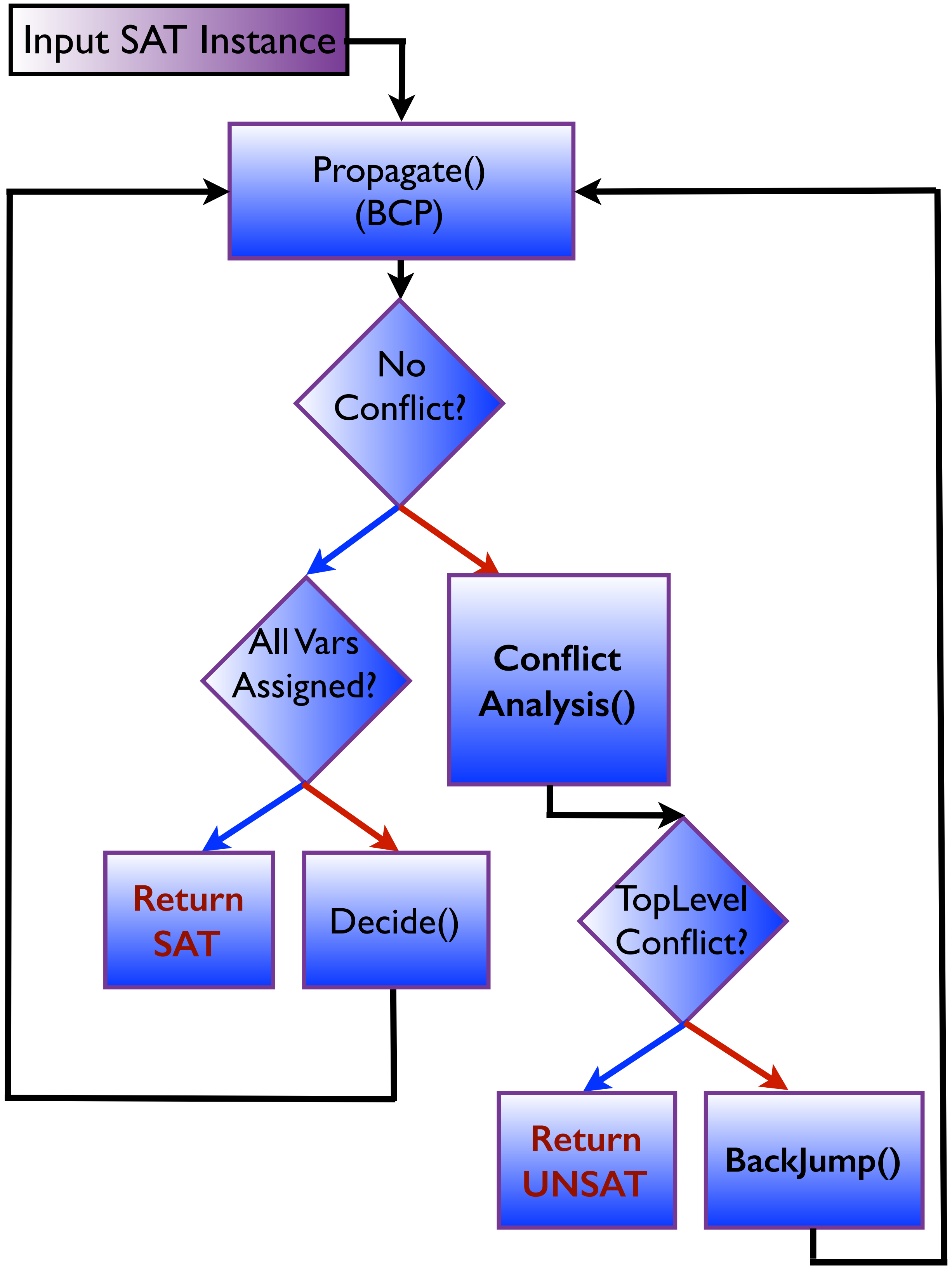
Watched Literal	Watcher List
-1	{-1, -3,
-3	...
8	{-1, -3,
...	...

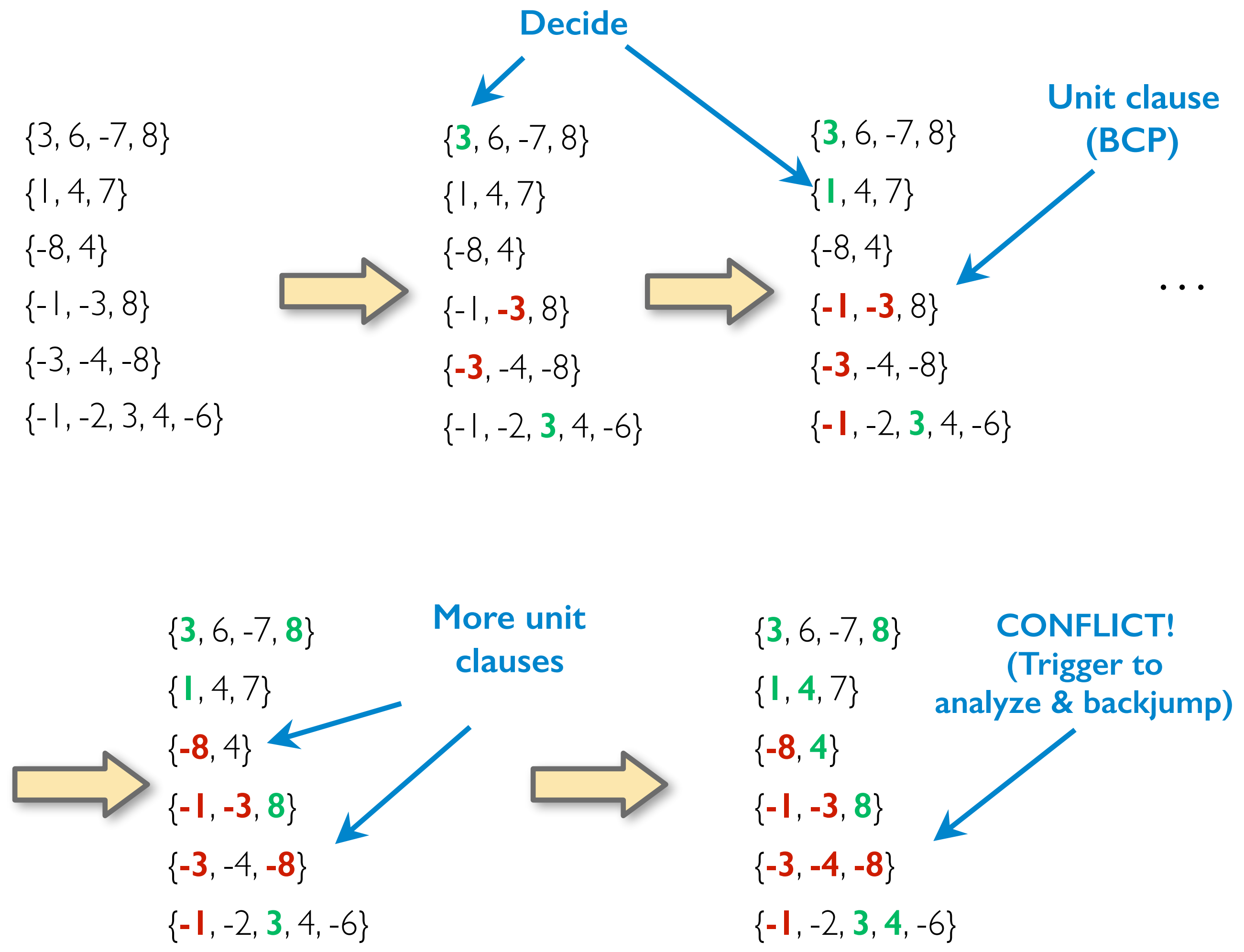
Watched Literal	Watcher List
-1	...
-3	...
8	{-1, -3,
...	...

The constraint propagates 8

Modern CDCL SAT Solver Architecture

Propagate(), Decide(), Analyze/Learn(), BackJump()

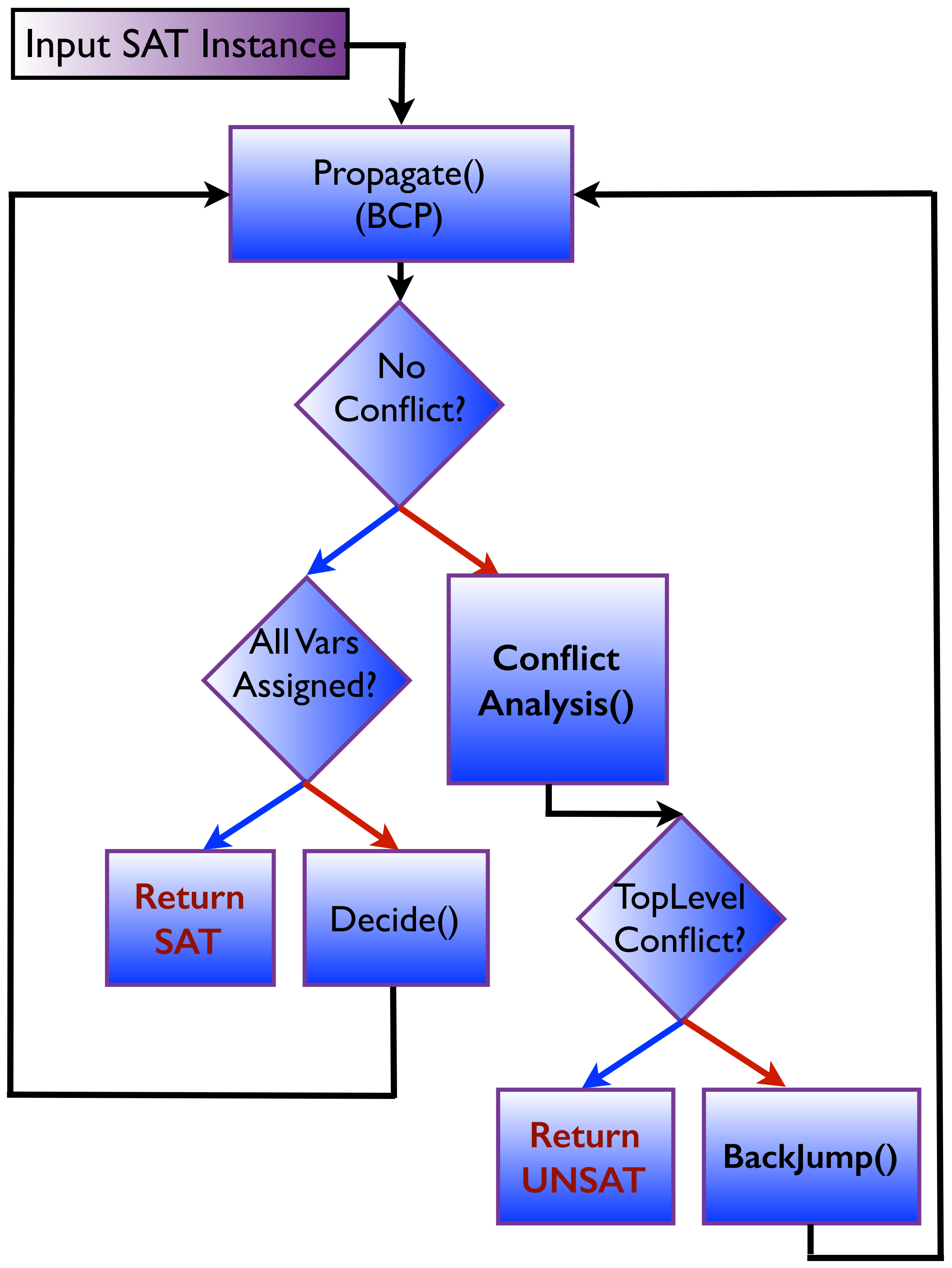




Basic Backtracking Search

- Flip the last decision 1
- Try setting 1 to False
- Highly inefficient
- No learning from mistakes

Modern CDCL SAT Solver Architecture
Conflict Analysis/Learn() Details



Some Definitions

- Decision Level (DL)
  - Map from Boolean variables in input to natural numbers
  - All unit clauses in input & resultant propagations get DL = 0
  - Every decision var gets a DL in increasing order >= 1
  - All propagations due to decision var at DL=x get the DL=x
- Conflict Graph (CG) or Implication Graph
  - Directed Graph that records decisions & propagations
  - Vertices: literals, Edge: unit clauses
- Conflict Clause (CC)
  - Clause returned by Conflict Analysis(), added to conflict DB
  - Implied by the input formula
  - A cut in the CG
  - Prunes the search
- Assignment Trail (AT)
  - A stack of partial assignment to literals, with DL info

Modern CDCL SAT Solver Architecture
Conflict Analysis/Learn() Details: Implication Graph

Current Assignment Trail: $\{X_9 = 0@1, X_{10} = 0@3, X_{11} = 0@3, X_{12} = 1@2, X_{13} = 1@2, \ldots\}$

Current decision: $\{X_1 = 1@6\}$

Clause DB

$W_1 = (\neg X_1 + X_2)$

$W_2 = (\neg X_1 + X_3 + X_9)$

$W_3 = (\neg X_2 + \neg X_3 + X_4)$

$W_4 = (\neg X_4 + X_5 + X_{10})$

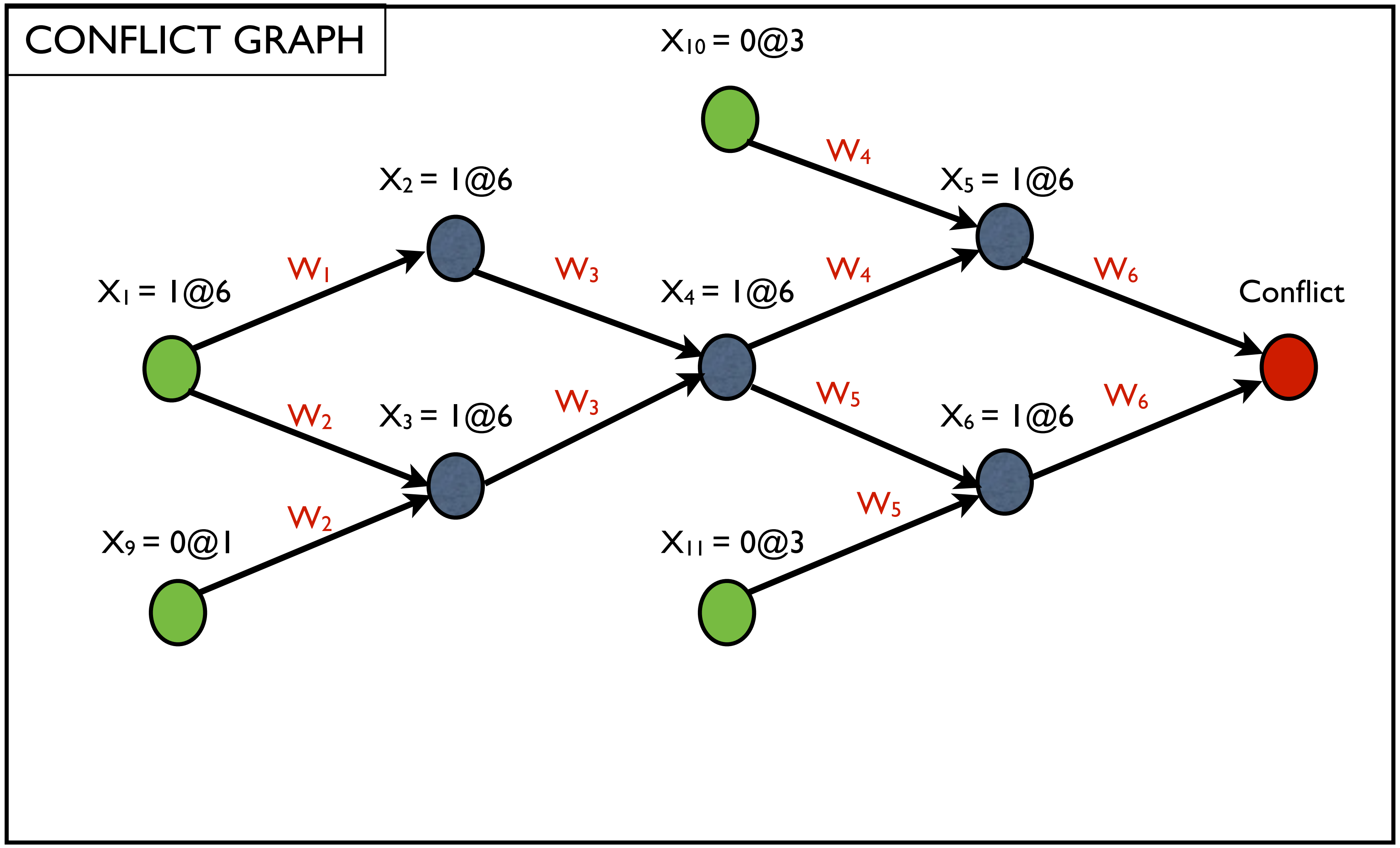
$W_5 = (\neg X_4 + X_6 + X_{11})$

$W_6 = (\neg X_5 + \neg X_6)$

$W_7 = (X_1 + X_7 + \neg X_{12})$

$W_8 = (X_1 + X_8)$

$W_9 = (\neg X_7 + \neg X_8 + \neg X_{13})$



Modern CDCL SAT Solver Architecture
Conflict Analysis/Learn() Details: Conflict Clause

Current Assignment Trail: $\{X_9 = 0@1, X_{10} = 0@3, X_{11} = 0@3, X_{12} = 1@2, X_{13} = 1@2, \ldots\}$

Current Decision: $\{X_1 = 1@6\}$

Simplest strategy is to traverse the conflict graph backwards until decision variables: conflict clause includes only decision variables $(\neg X_1 + X_9 + X_{10} + X_{11})$

Clause DB

$W_1 = (\neg X_1 + X_2)$

$W_2 = (\neg X_1 + X_3 + X_9)$

$W_3 = (\neg X_2 + \neg X_3 + X_4)$

$W_4 = (\neg X_4 + X_5 + X_{10})$

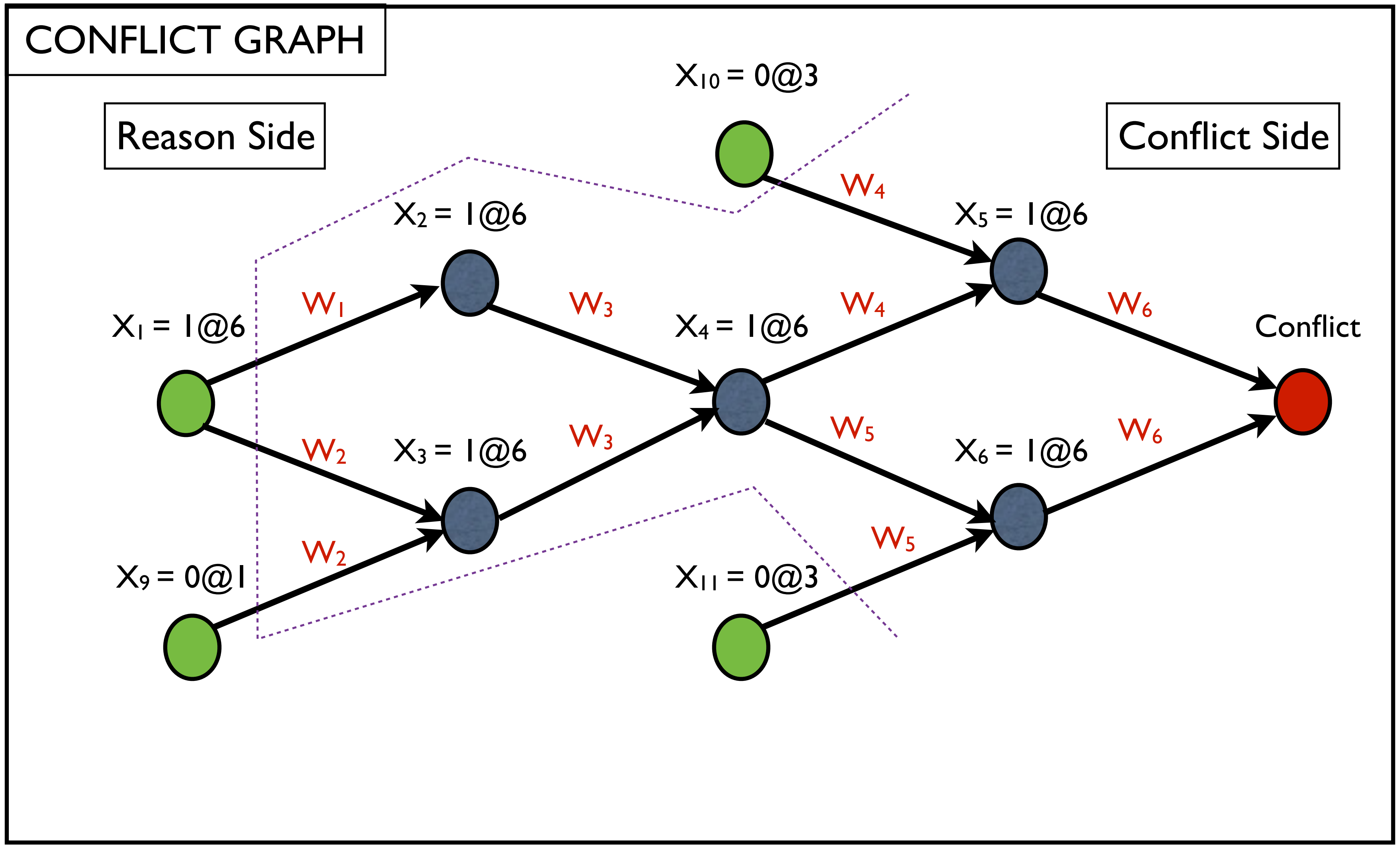
$W_5 = (\neg X_4 + X_6 + X_{11})$

$W_6 = (\neg X_5 + \neg X_6)$

$W_7 = (X_1 + X_7 + \neg X_{12})$

$W_8 = (X_1 + X_8)$

$W_9 = (\neg X_7 + \neg X_8 + \neg X_{13})$



Modern CDCL SAT Solver Architecture
Conflict Analysis/Learn() Details: Conflict Clause

Current Assignment Trail: {$X_9 = 0@1, X_{10} = 0@3, X_{11} = 0@3, X_{12} = 1@2, X_{13} = 1@2, ...$}

Current Decision: {$X_1 = 1@6$}

Another strategy is to use First Unique Implicant Point (UIP):
Traverse graph backwards in breadth-first, expand literals of conflict, stop at first UIP

Clause DB

$W_1 = (\neg X_1 + X_2)$

$W_2 = (\neg X_1 + X_3 + X_9)$

$W_3 = (\neg X_2 + \neg X_3 + X_4)$

$W_4 = (\neg X_4 + X_5 + X_{10})$

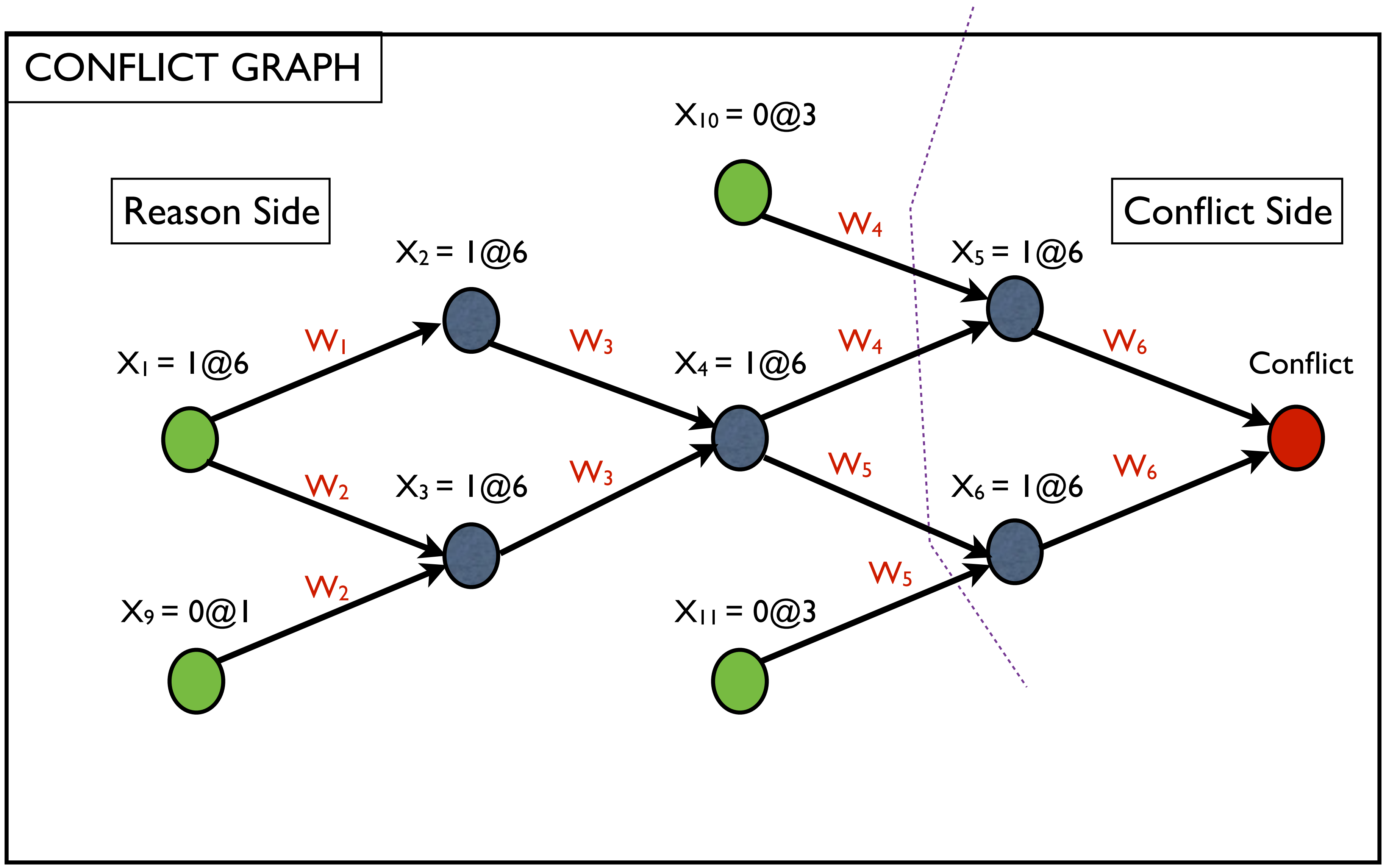
$W_5 = (\neg X_4 + X_6 + X_{11})$

$W_6 = (\neg X_5 + \neg X_6)$

$W_7 = (X_1 + X_7 + \neg X_{12})$

$W_8 = (X_1 + X_8)$

$W_9 = (\neg X_7 + \neg X_8 + \neg X_{13})$



Modern CDCL SAT Solver Architecture

Conflict Analysis/Learn() Details: BackTrack

Current Assignment Trail: {X_9 = 0@1, X_{10} = 0@3, X_{11} = 0@3, X_{12} = 1@2, X_{13} = 1@2, ...}

Current decision: {X_1 = 1@6}

Strategy: Closest decision level (DL) ≤ current DL for which conflict clause is unit. Undo {X_1 = 1@6}

Clause DB

$W_1 = (\neg X_1 + X_2)$

$W_2 = (\neg X_1 + X_3 + X_9)$

$W_3 = (\neg X_2 + \neg X_3 + X_4)$

$W_4 = (\neg X_4 + X_5 + X_{10})$

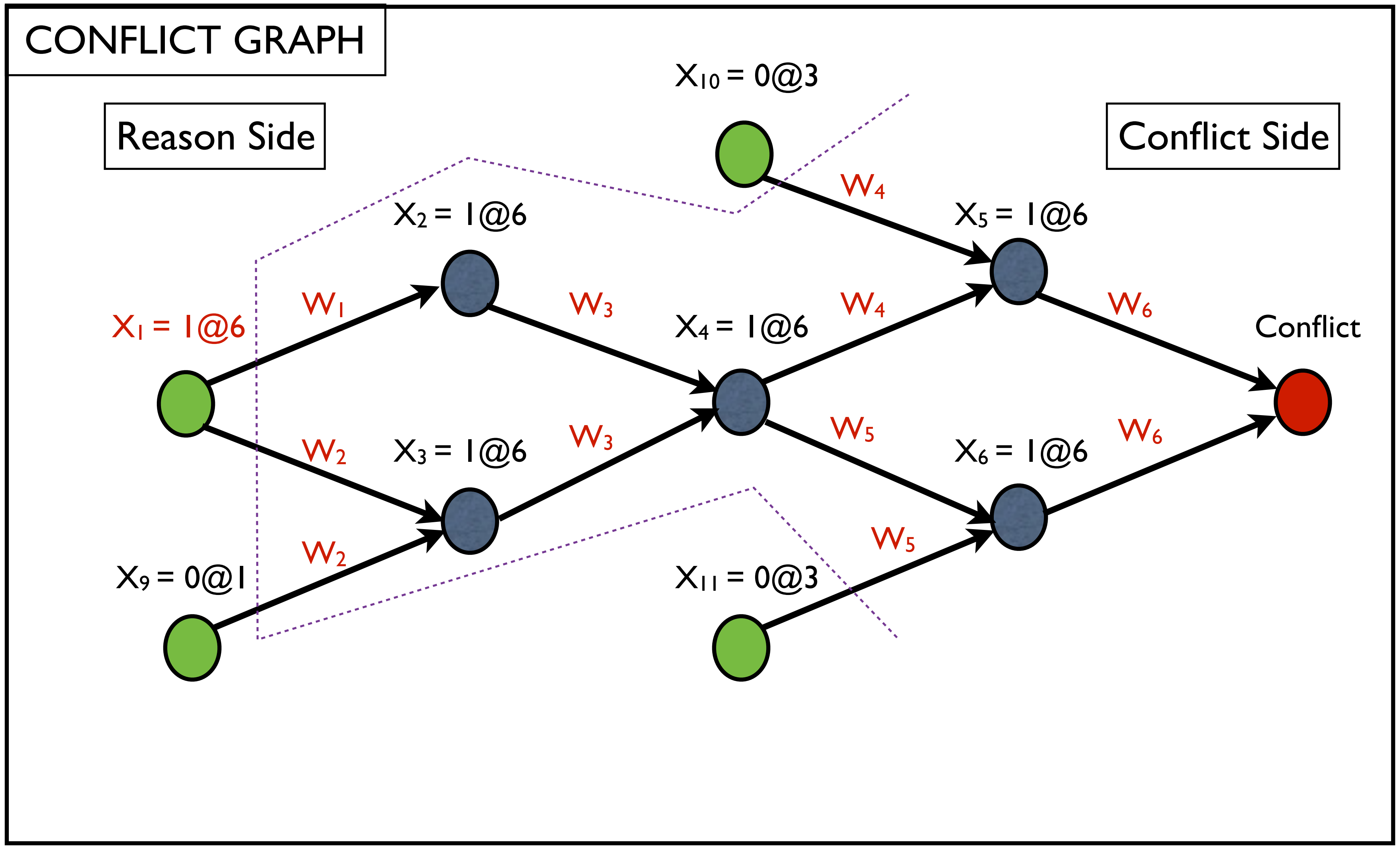
$W_5 = (\neg X_4 + X_6 + X_{11})$

$W_6 = (\neg X_5 + \neg X_6)$

$W_7 = (X_1 + X_7 + \neg X_{12})$

$W_8 = (X_1 + X_8)$

$W_9 = (\neg X_7 + \neg X_8 + \neg X_{13})$



Modern CDCL SAT Solver Architecture
Conflict Analysis/Learn() Details: BackJump

$\neg X_1$ was implied literal, leading to another conflict described below

Conflict clause: $(X_9 + X_{10} + X_{11} + \neg X_{12} + \neg X_{13})$

BackJump strategy: Closest decision level (DL) $\leq$ current DL for which conflict clause is unit. Undo $\{X_{10} = 0@3\}$

CLAUSE DB

$W_1 = (\neg X_1 + X_2)$

$W_2 = (\neg X_1 + X_3 + X_9)$

$W_3 = (\neg X_2 + \neg X_3 + X_4)$

$W_4 = (\neg X_4 + X_5 + X_{10})$

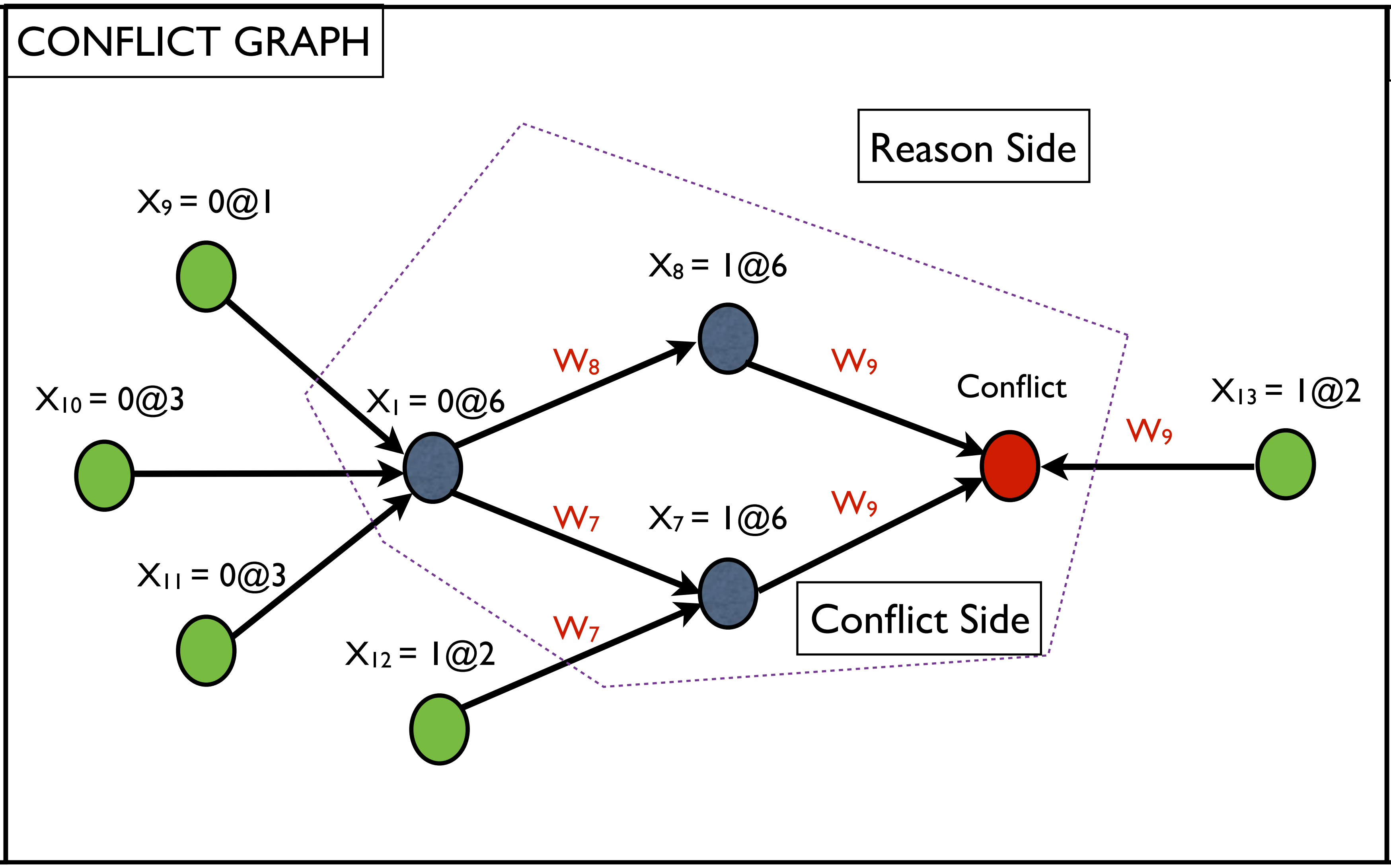
$W_5 = (\neg X_4 + X_6 + X_{11})$

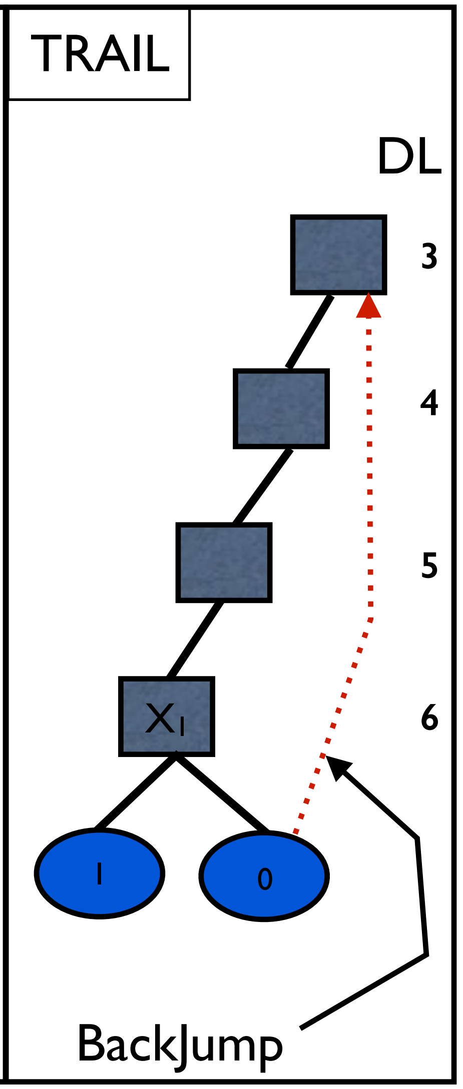
$W_6 = (\neg X_5 + \neg X_6)$

$W_7 = (X_1 + X_7 + \neg X_{12})$

$W_8 = (X_1 + X_8)$

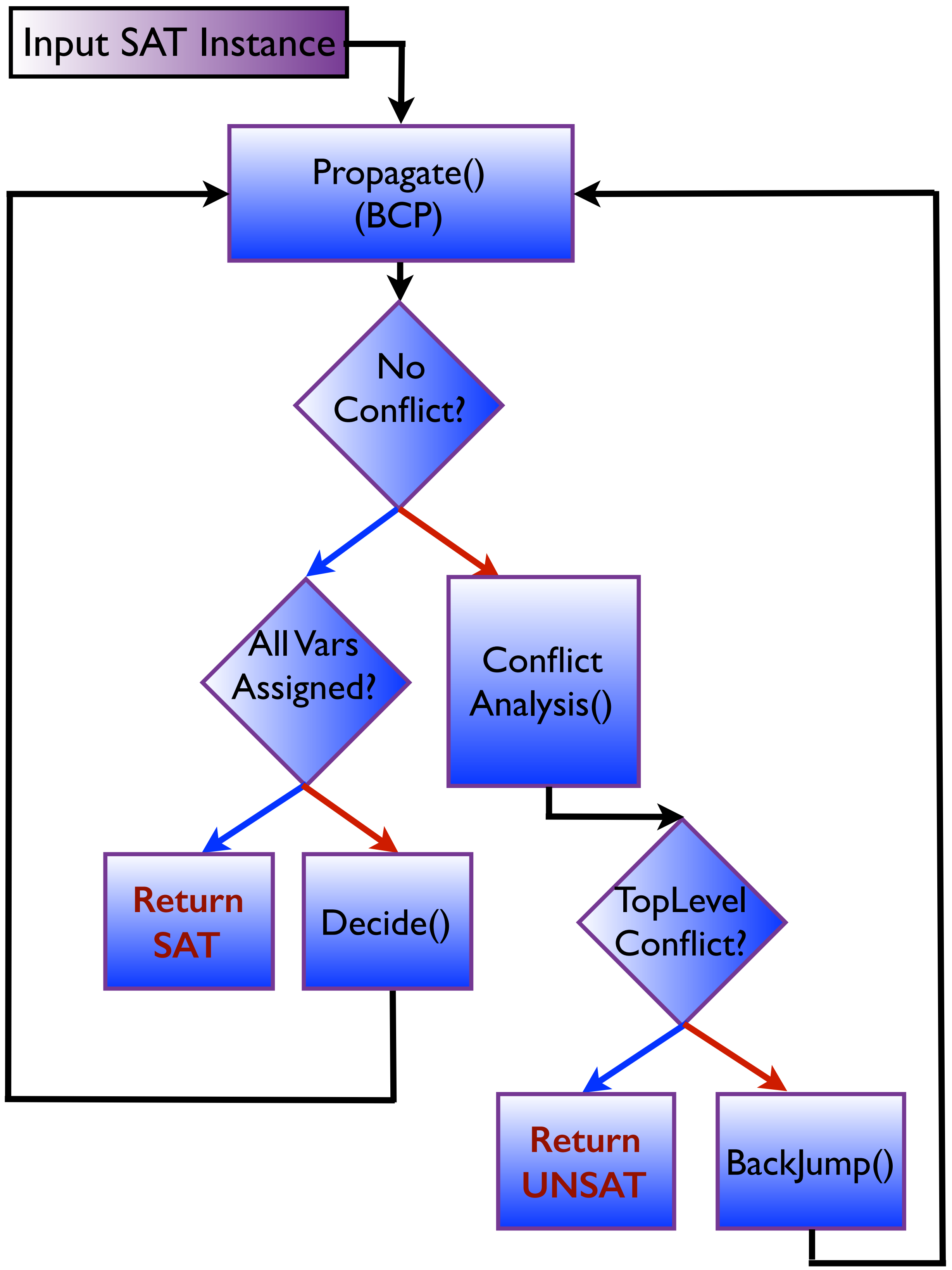
$W_9 = (\neg X_7 + \neg X_8 + \neg X_{13})$





Modern CDCL SAT Solver Architecture

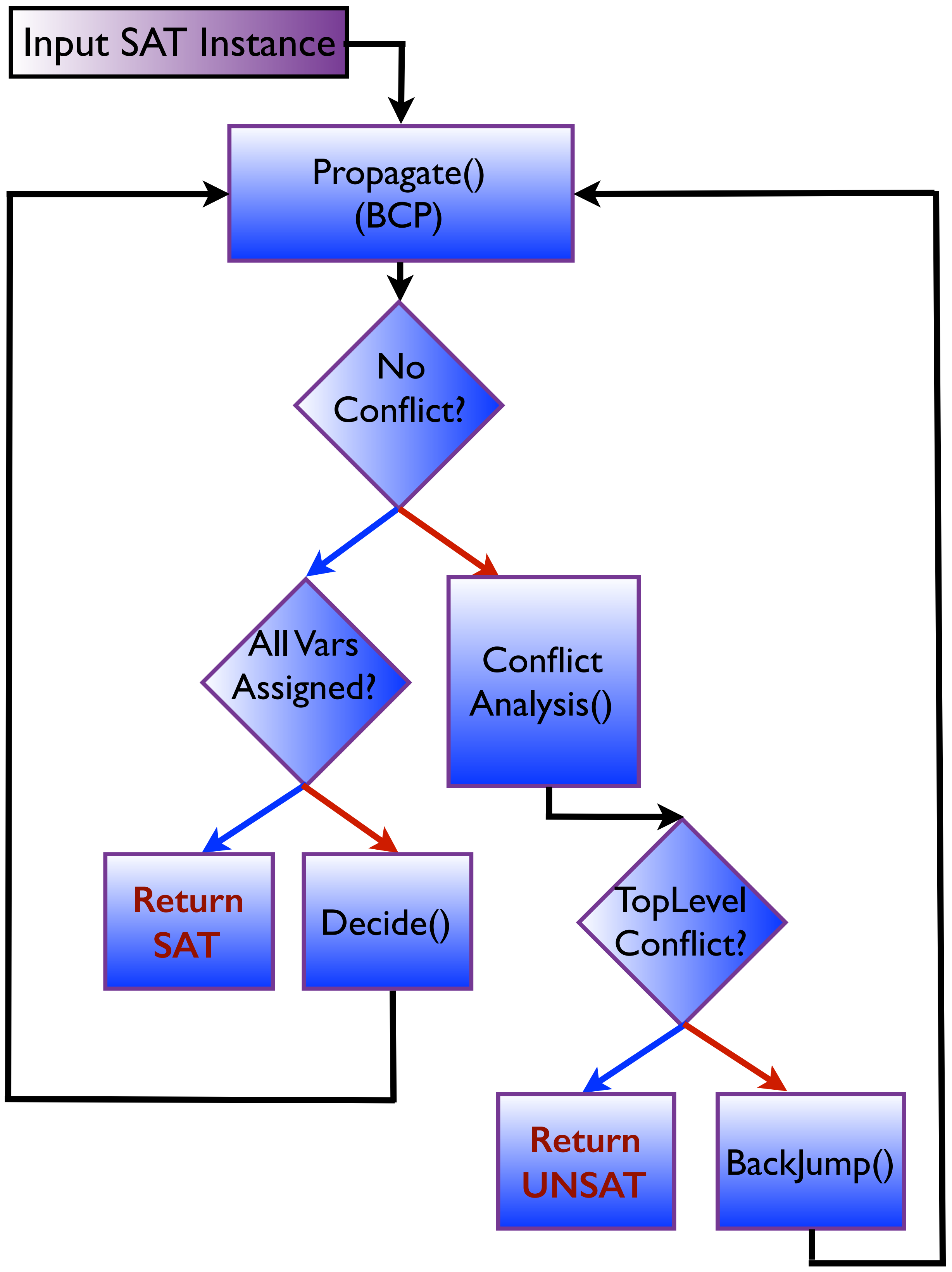
Restarts and Forget



- Restarts
  - Clear the Trail and start again
  - Start searching with a different variable order
  - Only Conflict Clause (CC) database is retained
- Forget: throw away less active learnt conflict clauses routinely
  - Routinely throw away very large CC
  - Logically CC are implied
  - Hence no loss in soundness/completeness
  - Time Savings: smaller DB means less work in propagation
  - Space savings

Modern CDCL SAT Solver Architecture

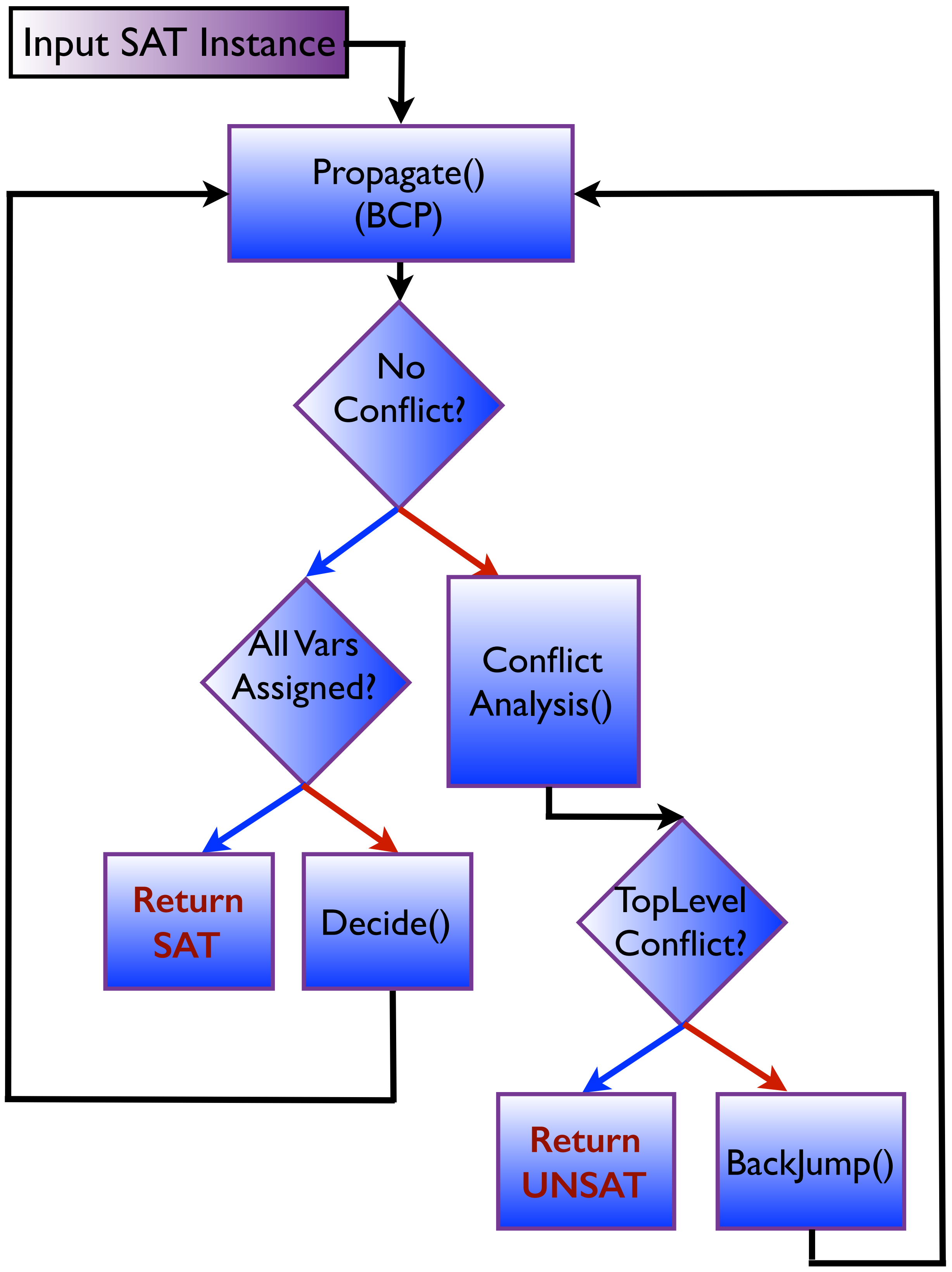
Why is SAT efficient?



- VSIDS branching heuristic and propagate (BCP)
- Conflict-Driven Clause-Learning (CDCL)
- Forget conflict clauses if DB goes too big
- BackJump
- Restarts
- All the above elements are needed for efficiency
- Deeper understanding lacking
- No predictive theory

Modern CDCL SAT Solver Architecture

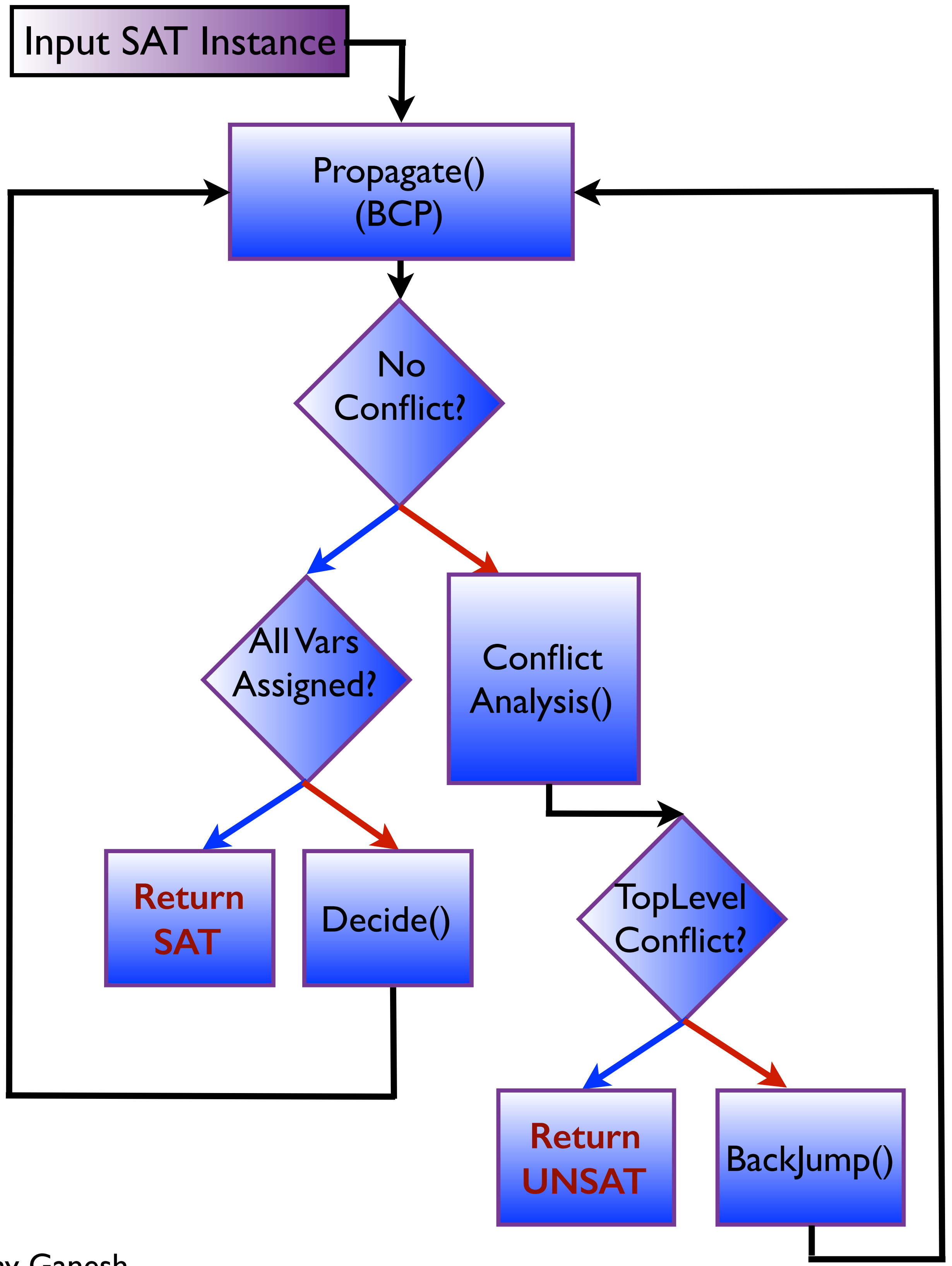
Propagate(), Decide(), Analyze/Learn(), BackJump()



- Conflict-Driven Clause-Learning (CDCL)
  (Marques-Silva & Sakallah 1996)
- Decide/branch and propagate (BCP)
  (Malik et al. 2001, Zabih & McAllester 1988)
- BackJump
  (McAllester 1980, Marques-Silva & Sakallah 1999)
- Restarts
  (Selman & Gomes 2001)
- Follows MiniSAT
  (Een & Sorensson 2003)

Modern CDCL SAT Solver Architecture

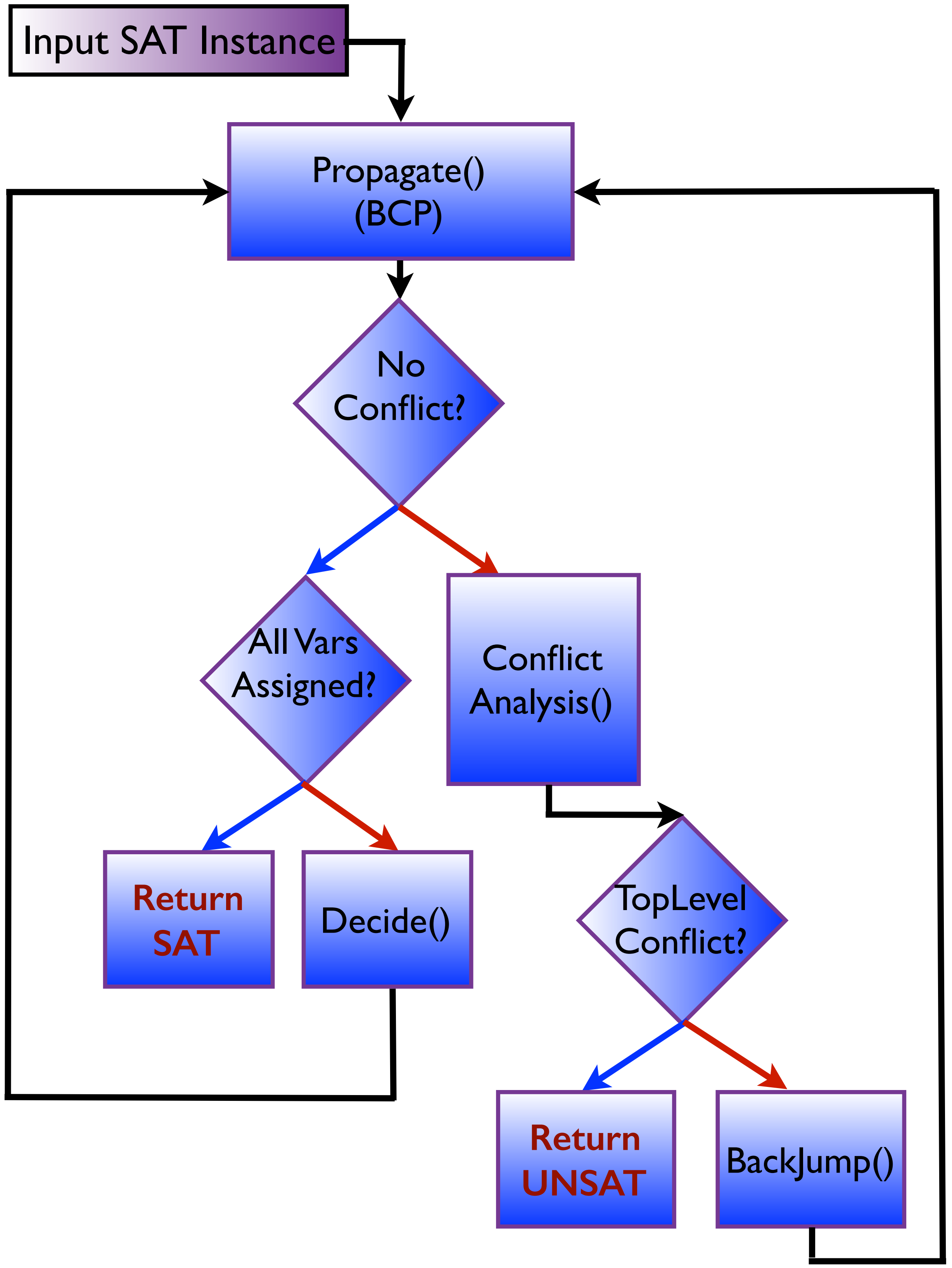
Soundness, Completeness & Termination



Soundness: A solver is said to be sound, if, for any input formula F, the solver terminates and produces a solution, then F is indeed SAT

Proof: (Easy) SAT is returned only when all vars have been assigned a value (True, False) by Decide or BCP, and solver checks the solution.

Modern CDCL SAT Solver Architecture
Soundness, Completeness & Termination

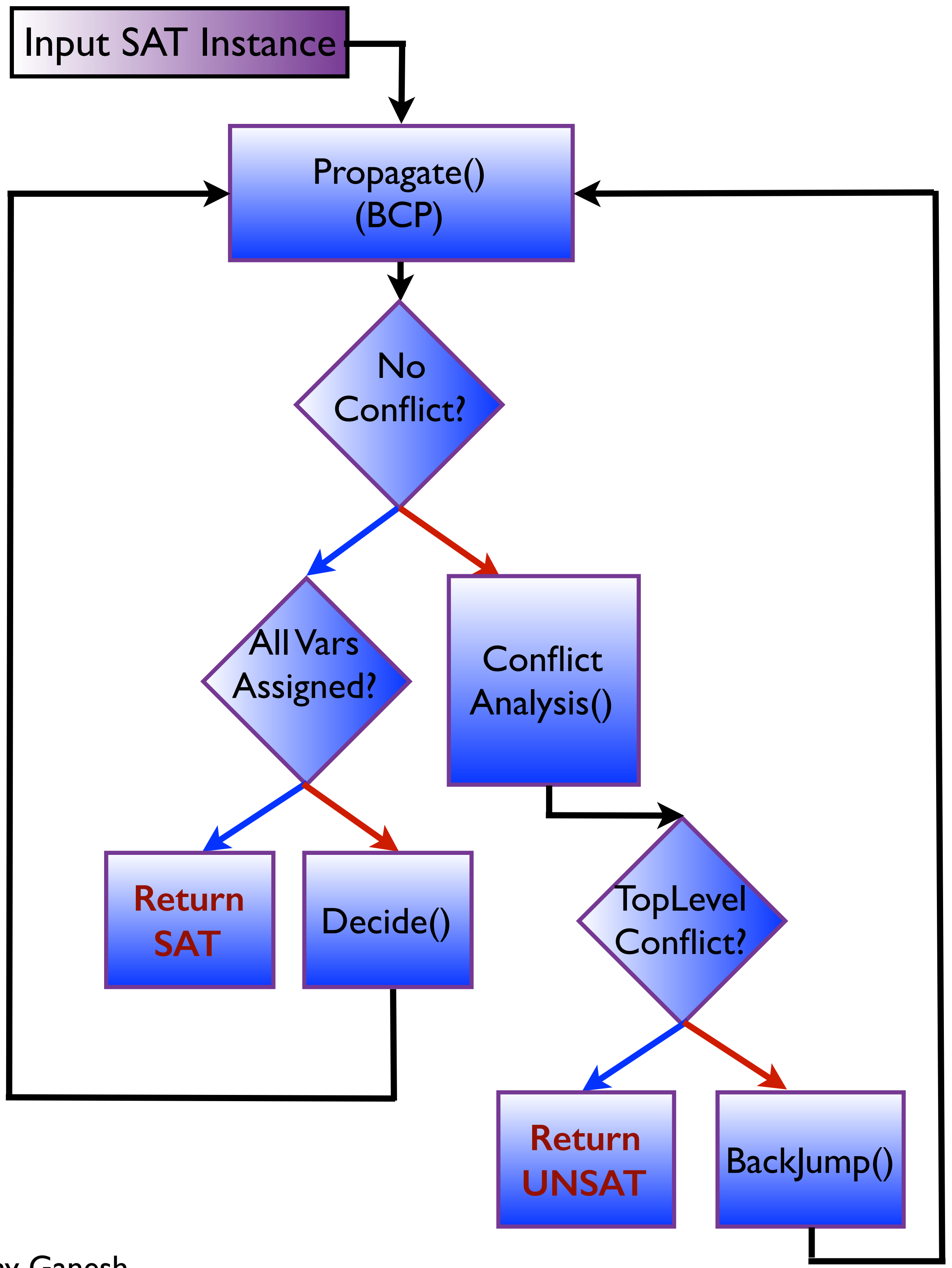


Completeness: A solver is said to be complete, if, for any input formula F that is SAT, the solver terminates and produces a solution (i.e., solver does not miss solutions)

Proof: (Harder)

- Backtracking + BCP + decide is complete (easy)
- Conflict clause is implied by input formula (easy)
- Only need to see backjumping does not skip assignments
    - Observe backjumping occurs only when conflict clause (CC) vars < decision level (DL) of conflicting var
    - Backjumping to max(DL of vars in CC)
    - Decision tree rooted at max(DL of vars in CC)+1 is guaranteed to not satisfy CC
    - Hence, backjumping will not skip assignments

Modern CDCL SAT Solver Architecture
Soundness, Completeness & Termination



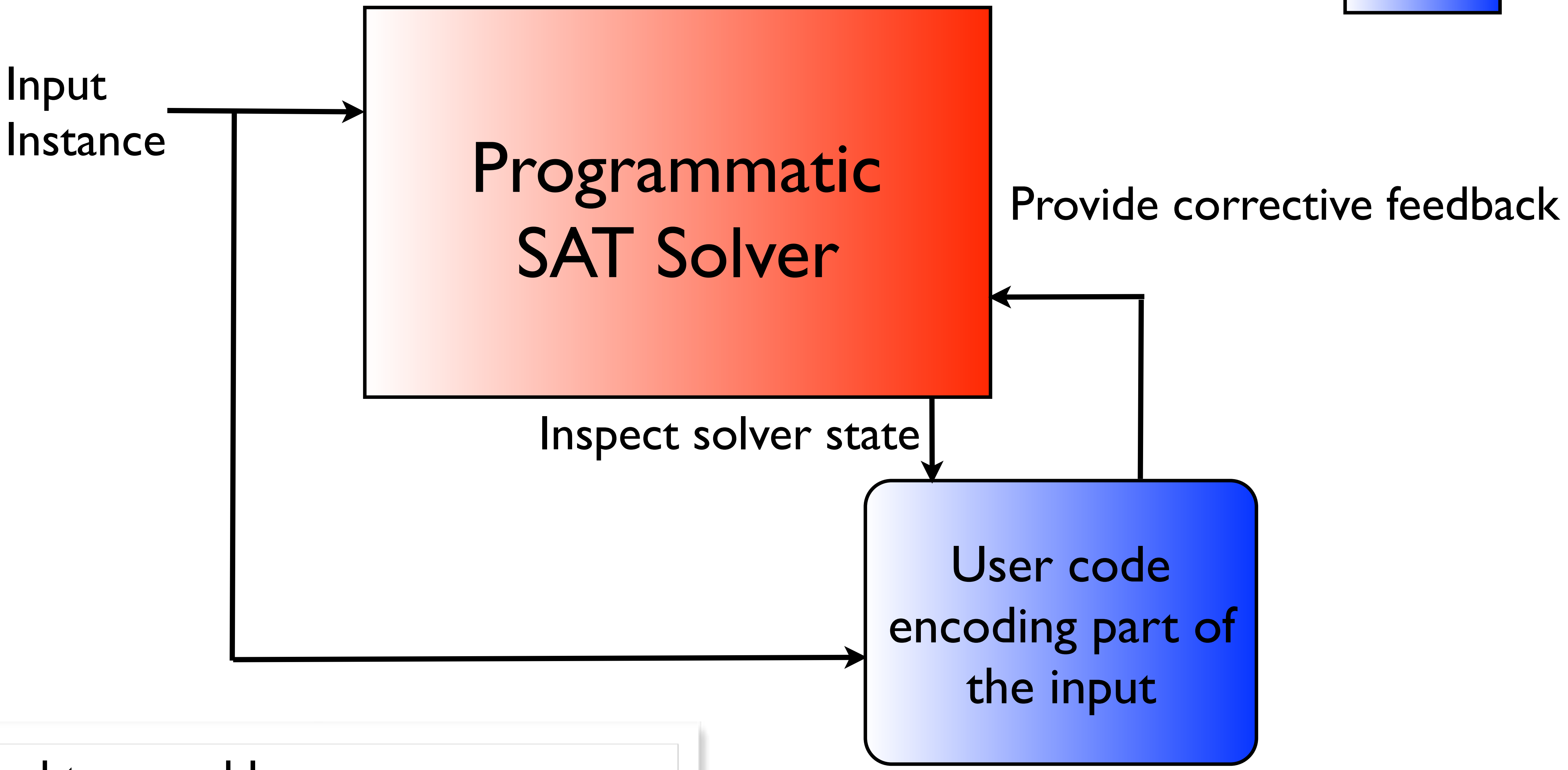
Termination: Some measure decreases every iteration

Proof Sketch:

- Loop guarantees either conflict clause (CC) added OR assign extended
- CC added. What stops CC addition looping forever?
  - Recall that CC is remembered
  - No CC duplication possible
  - CC blocks UNSAT assign exploration in decision tree. No duplicate UNSAT assign exploration possible
  - Size of decision tree explored decreases for each CC add

Problem: Solvers are blackboxes
Solution: Programmatic SAT

SAT 2012


Input Instance
Programmatic SAT Solver
Provide corrective feedback
Inspect solver state
User code encoding part of the input

Solved two problems:
- RNA folding problem
- Sub-graph isomorphism

Solvers and Software Engineering
Putting it All Together

1. SAT/SMT solvers are crucial for software engineering
2. SAT solvers key to SMT (Z3, CVC4, Yices, MathSAT, STP,...)
3. Huge impact in formal methods, program analysis and testing
4. Key ideas that make SAT efficient
    1. Conflict-driven clause learning
    2. VSIDS (or similar) variable selection heuristics
    3. Backjumping
    4. Restarts
5. Teacher-student analogy

One Slide History of Constraint Solving Methods

Before modern conception of logic (Before Boole and Frege)

• From Babylon to present day: Huge amount of work on methods to solve (find roots of) polynomials over reals, integers,...

• System of linear equations over the reals (Chinese methods, Cramer's method, Gauss elimination)

• These methods were typically not complete (e.g., worked for a special class of polynomials)

After modern conception of logic

•Systems of linear inequalities over the integers are solvable (Presburger, 1927)

•Peano arithmetic is undecidable (hence, not solvable) (Godel, 1931)

•First-order logic is undecidable (hence, not solvable) (Turing,1936. Church, 1937)

•A exponential-time algorithm for Boolean SAT problem (Davis, Putnam, Loveland, Loggeman in 1962)

•Systems of Diophantine equations are not solvable (Matiyasevich. 1970)

•Boolean SAT problem is NP-complete (Cook 1971)

•Many efficient, scalable SAT procedures since 1962 for a variety of mathematical theories

Modern CDCL SAT Solver Architecture

References & Important SAT Solvers

1. Marques-Silva, J.P. and K.A. Sakallah. *GRASP: A Search Algorithm for Propositional Satisfiability.* IEEE Transactions on Computers 48(5), 1999, 506-521.

2. Marques-Silva, J.P. and K.A. Sakallah. *GRASP: A Search Algorithm for Propositional Satisfiability*. Proceedings of ICCAD, 1996.

3. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. *CHAFF: Engineering an efficient SAT solver*. Proceedings of the Design Automation Conference (DAC), 2001, 530-535.

4. L. Zhang, C. F. Madigan, M. H. Moskewicz and S. Malik. *Efficient Conflict Driven Learning in a Boolean Satisfiability Solver*. Proceedings of ICCAD, 2001, 279-285.

5. 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/

6. M. Davis, G. Logemann, and D. Loveland. *A machine program for theorem proving.* Communications of the ACM. 1962.

7. zChaff SAT Solver by Lintao Zhang 2002.

8. GRASP SAT Solver by Joao Marques-Silva and Karem Sakallah 1999.

9. MiniSAT Solver by Niklas Een and Niklas Sorenson 2005 - present

10. SAT Live: http://www.satlive.org/, SAT Competition: http://www.satcompetition.org/

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