Satisfiability Checking for Timed Systems

Scott Cotton
Verimag
Scott.Cotton@imag.fr
The Satisfiability Problem.

Given a formula $\varphi$, determine whether there exists an assignment to the variables in $\varphi$ which makes $\varphi$ true.

Examples:

- $(p \lor q) \land \neg p$
  \[ \{p \mapsto \text{false}, q \mapsto \text{true}\} \]

- $(p \lor q) \land \neg p \land \neg q$
  No assignment exists.

- $(x - y \leq 0 \lor y - x \leq -1) \land (y - x \leq -1)$
  \[ \{x \mapsto 1, y \mapsto 0\} \]

- $(y - x \leq -1) \land (x - y \leq -1)$
  No assignment exists.
The Other Satisfiability Problem.

Mick Jagger: “I can’t get no satisfaction”
Uses of SAT.

- Finding false paths in circuits (all Boolean variables).
- Circuit equivalence checking. (all Boolean variables).
- Bounded Model Checking of timed and hybrid automata (mixed variables).
- Finding optimal schedules (mixed variables).
SAT for Timed Systems.

- SAT can be used for answering yes/no questions about timed systems.
- SAT can provide witnesses in the form of states satisfying some constraints.
- Timed SAT problems generally consist of
  - Mixed variables.
  - Difference constraints $x - y \leq c$.
- Is SAT practical for timed systems?
Road Map.

- How SAT solving works.
- Implementing a SAT Solver for timed systems.
- Experiments.
Road Map.

- **How SAT solving works.**
  - Propositional Satisfiability:
    * CNF
    * Search Trees
    * Unit Propogation
    * Learning
  - Satisfiability Modulo Theories:
    * Lazy, Eager, and Direct methods.
    * The DPLL[T] direct method.
- Implementing a SAT Solver for timed systems.
- Experiments.
Road Map.

- How SAT solving works.
- **Implementing a SAT Solver for timed systems.**
  - Difference constraints and constraint graphs.
  - Stack threaded constraint graphs.
  - Incremental negative cycle detection.
  - Round robin constraint propagation.
- Experiments.
Introduction.

Road Map.

• How SAT solving works.

• Implementing a SAT Solver for timed systems

• Experiments.
  – Circuit timing analysis
  – Optimal schedules for hard job shop problems.
Boolean SAT Solving

- Translate to conjunctive normal form.
- Search the space of variable assignments.
- Augment the search with lookahead.
- While searching, learn clauses which are implied by the input problem.
Conjunctive Normal Form (CNF)

A formula is in CNF if it is in the form

\[ \bigwedge_{i} c_{i} \]

where each \( c_{i} \) is a clause in the form

\[ \bigvee_{j} l_{j} \]

and each \( l_{i} \) is a literal i.e. in the form \( x \) or \( \neg x \) for some variable \( x \).

A clause with no disjuncts denotes false (\( \bot \)) and is referred to as the empty clause.
Translation to CNF

We can translate a formula into CNF in linear time using extra variables:

- \( \text{CNF}(\varphi \land \psi) \rightarrow \text{CNF}(\varphi) \land \text{CNF}(\psi) \)
- \( \text{CNF}(\neg(\varphi \land \psi)) \rightarrow \text{CNF}(\neg\varphi \lor \neg\psi) \)
- \( \text{CNF}(\neg(\varphi \lor \psi)) \rightarrow \text{CNF}(\neg\varphi \land \neg\psi) \)
- \( \text{CNF}(\varphi \lor \psi) \rightarrow \text{CNF}^{x}(\varphi) \land \text{CNF}^{-x}(\psi) \) when \( \varphi \lor \psi \) is not a clause and \( x \) is a new variable.
- \( \text{CNF}^{x}(\varphi) \rightarrow \bigwedge_{i}(c_i \lor x) \) where \( \bigwedge_{i} c_i = \text{CNF}(\varphi) \)
Search for a Satisfying Assignment

A simple search algorithm: (simplified DPLL)

Repeat:

1. pick a variable and a truth value.
2. simplify the CNF formula.
3. if there are no more clauses to solve, quit.
4. if there is an empty clause (false), backtrack.
Search, Simplification

If we pick $x$ to be true, we would update the clauses containing the variable $x$ as follows:

1. $x \lor y \lor z$ becomes solved
2. $\neg x \lor y \lor z$ becomes $y \lor z$
3. $\neg x \lor y$ becomes $y$
4. $\neg x$ becomes $\bot$

Note that in case 3, there is no longer any need to guess the truth value of $y$.

Terminology: A clause with one literal is called a unit clause.
Boolean SAT: Search.

Boolean Constraint Propogation

A better search procedure would be: (DPLL)

Repeat:

1. pick a variable and a truth value.
2. simplify the CNF formula.
3. if there is a unit clause with $v$ or $\neg v$, pick the appropriate truth value for $v$, goto 2.
4. if there are no more clauses to solve, quit.
5. if there is an empty clause (false), backtrack.
Boolean Constraint Propagation

No BCP

BCP

v1
v2
v3
v4
v5
v6
v7
v8

v1
v2
v3
v4
v5
v6
v7
v8
Efficient Boolean Constraint Propagation:
Two Literal Watching

A Clause:

After assigning not l2:
Learning and Backtracking

Backtracking is like learning \( \neg (v_1 \land \neg v_2 \land v_3 \land \neg v_4) \)
which is equivalent to the clause \( \neg v_1 \lor v_2 \lor \neg v_3 \lor v_4 \)


Learning: An Example

\[
\neg x_1 \lor \neg x_2 \lor \neg x_3 \quad \neg x_8 \lor \neg x_5 \lor x_3 \quad \neg x_7 \lor \neg x_8 \lor x_5 \\
\neg x_4 \lor \neg x_6 \lor x_1 \quad \neg x_6 \lor \neg x_7 \lor x_4 \quad \neg x_9 \lor \neg x_{10} \lor x_6 \\
\neg x_4 \lor \neg x_5 \lor x_2 \quad \neg x_{10} \lor \neg x_{11} \lor x_8 \quad \neg x_9 \lor \neg x_{11} \lor x_7
\]

x_{10}
Learning: An Example

\[ \neg x_1 \lor \neg x_2 \lor \neg x_3 \quad \neg x_8 \lor \neg x_5 \lor x_3 \quad \neg x_7 \lor \neg x_8 \lor x_5 \]

\[ \neg x_4 \lor \neg x_6 \lor x_1 \quad \neg x_6 \lor \neg x_7 \lor x_4 \quad \neg x_9 \lor \neg x_{10} \lor x_6 \]

\[ \neg x_4 \lor \neg x_5 \lor x_2 \quad \neg x_{10} \lor \neg x_{11} \lor x_8 \quad \neg x_9 \lor \neg x_{11} \lor x_7 \]
Learning: An Example

\begin{align*}
\neg x_1 & \lor \neg x_2 \lor \neg x_3 \\
\neg x_4 & \lor \neg x_5 \lor x_1 \\
\neg x_4 & \lor \neg x_5 \lor x_2 \\
\neg x_8 & \lor \neg x_5 \lor x_3 \\
\neg x_8 & \lor \neg x_7 \lor x_4 \\
\neg x_9 & \lor \neg x_{10} \lor x_6 \\
\neg x_{10} & \lor \neg x_{11} \lor x_8 \\
\neg x_{10} & \lor \neg x_{11} \lor x_7
\end{align*}
Learning: An Example

\[ \neg x_1 \lor \neg x_2 \lor \neg x_3 \quad \neg x_8 \lor \neg x_5 \lor x_3 \quad \neg x_7 \lor \neg x_8 \lor x_5 \]
\[ \neg x_4 \lor \neg x_6 \lor x_1 \quad \neg x_6 \lor \neg x_7 \lor x_4 \quad \neg x_9 \lor \neg x_{10} \lor x_6 \]
\[ \neg x_4 \lor \neg x_5 \lor x_2 \quad \neg x_{10} \lor \neg x_{11} \lor x_8 \quad \neg x_9 \lor \neg x_{11} \lor x_7 \]
Learning: An Example

\[ \neg x_1 \lor \neg x_2 \lor \neg x_3 \quad \neg x_8 \lor \neg x_5 \lor x_3 \quad \neg x_7 \lor \neg x_8 \lor x_5 \]

\[ \neg x_4 \lor \neg x_6 \lor x_1 \quad \neg x_6 \lor \neg x_7 \lor x_4 \quad \neg x_9 \lor \neg x_{10} \lor x_6 \]

\[ \neg x_4 \lor \neg x_5 \lor x_2 \quad \neg x_{10} \lor \neg x_{11} \lor x_8 \quad \neg x_9 \lor \neg x_{11} \lor x_7 \]
Learning: An Example

\[
\neg x_1 \lor \neg x_2 \lor \neg x_3 \quad \neg x_8 \lor \neg x_5 \lor x_3 \quad \neg x_7 \lor \neg x_8 \lor x_5 \\
\neg x_4 \lor \neg x_6 \lor x_1 \quad \neg x_6 \lor \neg x_7 \lor x_4 \quad \neg x_9 \lor \neg x_{10} \lor x_6 \\
\neg x_4 \lor \neg x_5 \lor x_2 \quad \neg x_{10} \lor \neg x_{11} \lor x_8 \quad \neg x_9 \lor \neg x_{11} \lor x_7
\]

Conflict!
Boolean SAT: Learning

Learning: An Example

\[
\neg x_1 \lor \neg x_2 \lor \neg x_3 \quad \neg x_8 \lor \neg x_5 \lor x_3 \quad \neg x_7 \lor \neg x_8 \lor x_5 \\
\neg x_4 \lor \neg x_6 \lor x_1 \quad \neg x_6 \lor \neg x_7 \lor x_4 \quad \neg x_9 \lor \neg x_{10} \lor x_6 \\
\neg x_4 \lor \neg x_5 \lor x_2 \quad \neg x_{10} \lor \neg x_{11} \lor x_8 \quad \neg x_9 \lor \neg x_{11} \lor x_7
\]

Conflict!
Boolean SAT: Conclusion

- Efficient translation to CNF with extra variables.
- Search alternates between decisions and deductions.
- Learning prunes the search space.
- Optimizations make this NP-hard problem feasible for many large problems.
Satisfiability Modulo Theories

Methods for satisfiability checking of formula with mixed variables:

- **Eager methods** (Strichman, CAV 2002).
  Translation to Boolean SAT.

- **Lazy methods** (MathSAT).
  Uses *model enumeration* available in some Boolean SAT solvers, checking each model.

- **Direct methods** (Tinelli, Ganzinger, CAV 2004)
  Interpretation of atoms in a theory is directly interleaved with the Boolean SAT solving process.
Mixed Variable SAT: DPLL[T].

**DPLL[T]**

- Instead of purely Boolean formulas, we solve Boolean combinations of atoms \( P(\bar{x}) \) with arbitrary variables \( \bar{x} \).
- The atoms in a formula fall within a given *theory*.
- The DPLL SAT procedure is parameterized on a solver for an *arbitrary theory*.
- Most of the structure of modern Boolean SAT solver is maintained.
- An interface to be implemented by a *theory interpreter* is specified.
DPLL[T]: Theory Interpreters

- A theory specifies the language in which atoms may appear. Example: for timed systems the theory of *difference constraints* allows $P(x) \equiv x - y \leq c$.

- The theory interpreter is asked whether a conjunction of atoms is satisfiable.

- The theory interpreter may tell the solver that some atoms are implied.

- The theory interpreter can be asked to give an explanation of why an atom is implied.

- During backtracking, the theory interpreter is told to uninterpret those atoms whose truth value will not be known after the backtracking is complete.
DPLL[T]: Parameterized DPLL

A parameterized DPLL Boolean solver acts just like an unparameterized one except it must in addition:

- Ask the theory interpreter to interpret every atom (or its negation) whenever its truth assignment is made.

- Check whether the result of the interpretation is feasible. Example: after interpreting $x - y \leq 0 \land y - x \leq -1$, the result is not feasible.

- If the result is feasible, ask the theory interpreter if it knows of any implied atoms.

- If the result is not feasible, ask the theory interpreter to give an explanation of why.

- Tell the theory interpreter to backtrack.
Mixed Variable SAT: DPLL[T].

DPLL[T] Search Tree

\[ P_1(\bar{x}) \]
\[ P_2(\bar{x}) \]
\[ P_3(\bar{x}) \]
\[ P_4(\bar{x}) \]
\[ P_5(\bar{x}) \]
\[ P_6(\bar{x}) \]
\[ P_7(\bar{x}) \]
\[ P_8(\bar{x}) \]
Mixed Variable SAT: DPLL[T]. 32

DPLL[T]: Conclusion

- DPLL[T] is an enriched version of DPLL.
- Every node in the search tree is associated with a state of the theory interpreter.
- Theory implications are possible, but not necessary.
- Theory implications can be incomplete.
Timed SAT: Difference Constraints

- A difference constraint is an inequality of the form $x - y \leq c$ for variables $x$ and $y$ and some constant $c$.
- Difference constraints are characterized by shortest paths in constraint graphs.

Example:

$$(x - y \leq -1) \land (x - z \leq 0) \land (w - x \leq 0) \land (z - w \leq -1)$$
Timed SAT: Difference Constraint Interpreter

- Maintain a constraint graph $G$ and a potential function $\pi$ under every truth assignment.
- A potential function $\pi$ must satisfy:

$$\pi(x) + W_{xy} \geq \pi(y)$$

for every difference constraint $x - y \leq W_{xy}$.
- Given a potential function $\pi$, the function $-\pi(x)$ is a model of the difference constraints in the constraint graph.
- A potential function exists if and only if the constraint graph has no negative cycles.
Timed SAT: Interpreter Operations

We associate a constraint graph $G$ and a potential $\pi$ with every node in the search tree. Therefore we will use the following operations frequently (possibly exponentially many times):

- **Incremental extension** of the constraint graphs for every truth assignment.
- **Batch reduction** of the constraint graphs for every backtrack.
- **Implication** of uninterpreted constraints from interpreted constraints.
- **Explanation** of an implication.
Timed SAT: Interpreter Representation

- A stack $S$ of potential functions $\pi : V(\phi) \rightarrow \mathbb{Z}$, which grows with the depth of the search tree.

- Each difference constraint constitutes a weighted edge $(v_1, v_2, W_{v_1v_2}) \in V(\phi) \times V(\phi) \times \mathbb{Z}$.

- Represent the edges in a doubly linked adjacency list associated with the positive variable.

- A stack parallel to $S$ of pointers to the head of the adjacency list for each variable.

- Each constraint graph is threaded through the stack.
Timed SAT: Implementation.

Timed SAT: Interpreter Representation

\[
\begin{align*}
\pi_1 & \leq v_1 - v_2 \\ 
\pi_2 & \leq v_2 - v_3 \\ 
\pi_3 & \leq v_3 - v_1
\end{align*}
\]
Timed SAT: Stack Threaded Constraint Graphs

- Efficient incremental extension (next).
- Batch reduction (backtracking):
  - Simply pop the stack of potential functions while removing edges.
  - Backtracking which unassigns $n$ atoms’ truth values takes $O(n)$ time.
- Sharing of information between graphs $\implies$ compact representation.
Timed SAT: Incremental Negative Cycle Detection

Given a constraint graph $G$, a potential function $\pi$, and a difference constraint $x - y \leq c$:

- If $\pi(x) + c \geq \pi(y)$, then we add the edge $(x, y, c)$ to the graph at the top level.

- If $\pi(x) + c < \pi(y)$, then we compute a new potential function $\pi'$ for the extended graph $G' = G + (x, y, c)$.
Timed SAT: Incremental Negative Cycle Detection

Given a constraint graph $G$, a potential function $\pi$, and a difference constraint $x - y \leq c$ such that $\pi(x) + c < \pi(y)$:

- We perform a variant of Dijkstra’s single source shortest path algorithm, starting from $y$.
- The potential function $\pi$ will monotonically decrease to form $\pi'$.
- We estimate “distances” $\Delta : V(\varphi) \rightarrow \mathbb{Z}$ with $\Delta(v) \mapsto \pi(v) - \pi'(v)$.
- If there is a negative cycle, there will be an update to the variable $x$ and $\pi'(x) < \pi(x)$. 
Timed SAT: Incremental Negative Cycle Detection

Given a constraint graph $G$, a potential function $\pi$, and a difference constraint $x - y \leq c$ such that $\pi(x) + c < \pi(y)$:

Initialization:

- $\pi'(y) = \pi(x) + c$
- $\Delta(y) = \pi(y) - \pi'(y)$
- $\Delta(v) = \infty$ if $v \neq y$
- Create a priority queue of all variables/vertices with priority $\Delta$. 
Timed SAT: Incremental Negative Cycle Detection

Given a constraint graph $G$, a potential function $\pi$, and a difference constraint $x - y \leq c$ such that $\pi(x) + c < \pi(y)$:

Main Loop:

- Remove the variable/vertex $v$ from the queue which minimizes $\Delta$.
- Let $\pi'(v) = \pi(v) - \Delta(v)$
- Scan all outgoing edges $(v, w, c)$ for variables $w$ such that $\pi'(v) + c < \pi(w) - \Delta(w)$.
- For all such edges, requeue $w$ with priority $\pi(w) - (\pi'(v) + c)$. 

Timed SAT: Incremental Negative Cycle Detection

Conclusions:

• The algorithm has time complexity $O(|V| \log |V| + |E|)$, with best case run time of $O(|V|)$.

• Alternative methods:
  – Bellman-Ford $O(|V||E|)$ with best case constant run time.
  – Incremental Floyd-Warshall (DBM) is $O(|V|^2)$ with $O(|V|^2)$ best case run time.

• A simple variation removes the $O(|V|)$ initialization overhead, making the best case run time constant.

• Dramatic improvement for sparse graphs!
Timed SAT: Round Robin Constraint Propogation

- Transitively implied constraints.
  Example: $x - y \leq 2 \land y - z \leq 3$ implies $x - z \leq 5$.

- Semi lazy constraint propagation.

- No loss of completeness.

- Optional incomplete propagation, with an effective heuristic.
Timed SAT: Round Robin Constraint Propagation

Interleaved propagation:

- Boolean constraint propagation and transitivity constraint propagation are interleaved but independent.
- Every time an implied constraint is identified, the resulting Boolean constraint propagation can imply a constraint which extends the set of implied difference constraints.
- A piecewise method of finding implied constraints can reduce the overhead of constraint propagation.
Timed SAT: Round Robin Constraint Propagation

Method:

- Partition the set of uninterpreted difference constraints into $P_x, P_y, P_z, \ldots$ where
  \[
P_v = \{(x - y \leq c) \mid x = v\}
  \]

- Maintain the partitions in a list $C$, and a location $L(C)$ in the list.

- When looking for constraints implied by transitivity, scan the list in circular order starting from $L(C)$.

- When a partition is found which contains an implied constraint, update the location $L(C)$ to the next partition in the list and defer to Boolean constraint propagation.
Timed SAT: Round Robin Constraint Propagation

- Complete propagation will periodically require a scan of the entire list of partitions of uninterpreted constraints.
- Incomplete propagation can be performed by scanning only part of the list.
- Incomplete propagation can be made more effective by heuristically determining which partitions to include in the scan.
- Both complete and incomplete propagation can be optimized by using a quick test indicating possible existence of an implied constraint in a partition.
Timed SAT: Round Robin Constraint Propagation

Finding constraints in a partition $P_x$.

- Use a variant Johnson’s all pairs shortest path algorithm for shortest paths.
- The current potential potential function $\pi$ is used to reduce the single source shortest paths problem from vertex $x$ to Dijkstra’s algorithm.
- To process a single partition which passes the tests for possible existence of an implied constraint, the run time is $O(|V| \log |V| + |E|)$. 
Timed SAT: Round Robin Constraint Propagation

Heuristic for incomplete propagation:

- Maintain the uninterpreted difference constraints in a constraint graph.
- Whenever the potential function for a vertex $v$ decreases, add $\text{pre}(v) = \{P_x \mid (x - v \leq c) \text{ is an uninterpreted constraint}\}$ to the list of partitions.
- Whenever a partition is scanned, remove it from the list.
- Negative cycle detection + incomplete propagation of transitivity constraints takes $O(|V| \log |V| + |E|)$ time.
Timed SAT: Round Robin Constraint Propogation

Conclusions

- Semi-lazy propagation method.
- Can be made complete.
- Effective heuristic for incomplete propagation.
- Dramatic improvements for sparse graphs.
Experiments and Results

- Maximum stabilization time of a circuit.
- Classical job shop scheduling.
A timed model of circuit behavior (Maler):

- Each gate $g$ computes a boolean function $f_g$ of its inputs.

- A change in the input to a gate puts the gate in an excited state.

- An excited gate may stabilize within any time $t$ satisfying $l_g \leq t \leq u_g$.

- Only after stabilization will the new output of a gate be visible to other gates.

- An excited gate may regret within any time $t$ satisfying $0 \leq t \leq u_g$.

- The output of a gate that becomes excited and then regrets remains unchanged.
Timed Automaton for a single gate $y = f(x)$ with stabilization timing constraints $l \leq x \leq u$. 

\[
\begin{align*}
(1,0) & \quad \rightarrow \quad (0,0) \\
(0,1) & \quad \downarrow \quad x \leq u \quad \downarrow \quad x \geq l \land x \leq u \\
(1,1) & \quad \leftarrow \quad (0,0) \\
(0,0) & \quad \rightarrow \quad (1,1) \\
\end{align*}
\]
Overview of encoding bounded timed circuit behavior into DL via timed automata:

- Encode states for $n$ gates using $\log(4n)$ Boolean variables.
- Encode an initial state in which all gates are stable in a formula $I$.
- Encode a transition $\delta_i$ from step $i$ to step $i + 1$ as $\wedge_{g \in \text{gates}} \text{trans}(g, i)$
- The formula $\text{trans}(g, i)$ is a disjunction over the various transitions (timed, discrete, and idling) and introduces a new numeric variable for each (clock, step) pair.
- Encode a $k$-step unfolding of the timed automaton as $I \wedge \bigwedge_{i=1}^{k-1} \delta_i$
A 3-bit adder:

Given a constant input to the circuit following a stable initial state, can the circuit remain unstable after $k$ steps and $d$ time units?
<table>
<thead>
<tr>
<th>NBits</th>
<th>Unwinding (k)</th>
<th>Duration</th>
<th>Time (Secs)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>12</td>
<td>0.78</td>
<td>sat</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>13</td>
<td>0.54</td>
<td>unsat</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>18</td>
<td>7.09</td>
<td>sat</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>19</td>
<td>6.50</td>
<td>unsat</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>24</td>
<td>64.32</td>
<td>sat</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>25</td>
<td>39.90</td>
<td>unsat</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>30</td>
<td>305.62</td>
<td>sat</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>31</td>
<td>406.82</td>
<td>unsat</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>30</td>
<td>57.86</td>
<td>sat</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>31</td>
<td>61.47</td>
<td>unsat</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>30</td>
<td>128.68</td>
<td>sat</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>31</td>
<td>14.09</td>
<td>unsat</td>
</tr>
</tbody>
</table>
### Timed SAT: Experiments

<table>
<thead>
<tr>
<th>NBits</th>
<th>Unwinding ($k$)</th>
<th>Duration</th>
<th>Time (Secs)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>10</td>
<td>30</td>
<td>143.76</td>
<td>sat</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>31</td>
<td>119.82</td>
<td>unsat</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>30</td>
<td>124.81</td>
<td>sat</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>31</td>
<td>20.85</td>
<td>unsat</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>30</td>
<td>688.70</td>
<td>sat</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>31</td>
<td>211.67</td>
<td>unsat</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>36</td>
<td>929.10</td>
<td>sat</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>37</td>
<td>186.36</td>
<td>unsat</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>42</td>
<td>2063.75</td>
<td>sat</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>43</td>
<td>794.45</td>
<td>unsat</td>
</tr>
</tbody>
</table>
Job Shop Scheduling via Timed SAT

• Encode the schedulability of a job shop problem within a given makespan as a SAT problem with difference constraints.

• Find the optimal makespan by a binary search on the makespan.

• Show the role of propagation of transitivity constraints.
  – No propagation of transitivity constraints.
  – Incomplete propagation.
  – Complete propagation.
## Job Shop Scheduling via Timed SAT

### FT06 runtimes:

<table>
<thead>
<tr>
<th></th>
<th>52</th>
<th>53</th>
<th>54</th>
<th>55</th>
<th>56</th>
<th>57</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>0.11</td>
<td>0.13</td>
<td>0.11</td>
<td>0.21</td>
<td>0.23</td>
<td>0.12</td>
<td>0.03</td>
</tr>
<tr>
<td>I</td>
<td>0.04</td>
<td>0.08</td>
<td>0.25</td>
<td>0.04</td>
<td>0.21</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>C</td>
<td>0.06</td>
<td>0.08</td>
<td>0.26</td>
<td>0.12</td>
<td>0.06</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>unsat</td>
<td>unsat</td>
<td>unsat</td>
<td>unsat</td>
<td>sat</td>
<td>sat</td>
<td>sat</td>
</tr>
</tbody>
</table>
Job Shop Scheduling via Timed SAT

ABZ5 (10 × 10) runtimes:

<table>
<thead>
<tr>
<th></th>
<th>1200</th>
<th>1232</th>
<th>1233</th>
<th>1234</th>
<th>1235</th>
<th>1250</th>
<th>1300</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>550</td>
<td>&gt;1000</td>
<td>&gt;1000</td>
<td>&gt;1000</td>
<td>510</td>
<td>421</td>
<td>229</td>
</tr>
<tr>
<td>I</td>
<td>80</td>
<td>300</td>
<td>378</td>
<td>120</td>
<td>110</td>
<td>154</td>
<td>21</td>
</tr>
<tr>
<td>C</td>
<td>562</td>
<td>&gt;1000</td>
<td>&gt;1000</td>
<td>&gt;1000</td>
<td>&gt;1000</td>
<td>335</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>unsat</td>
<td>unsat</td>
<td>unsat</td>
<td>sat</td>
<td>sat</td>
<td>sat</td>
<td>sat</td>
</tr>
</tbody>
</table>
**Job Shop Scheduling via Timed SAT**

Various runtimes at the optimum with incomplete propagation:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Optimum</th>
<th>Makespan</th>
<th>Run Time (Secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>abz5</td>
<td>1234</td>
<td>1234</td>
<td>120</td>
</tr>
<tr>
<td>ft10</td>
<td>930</td>
<td>930</td>
<td>216</td>
</tr>
<tr>
<td>la19</td>
<td>842</td>
<td>842</td>
<td>342</td>
</tr>
<tr>
<td>la02</td>
<td>655</td>
<td>655</td>
<td>22</td>
</tr>
<tr>
<td>la21</td>
<td>1046</td>
<td>1191</td>
<td>315</td>
</tr>
</tbody>
</table>
Conclusion.

- New algorithms and representations for timed SAT.
- Several orders of magnitude of speed improvement (despite redoing everything in Java).
- A parametric SAT solver may be extended to include linear constraints or other theories.
- Some algorithms (round robin propagation) can be extended to other theories.
Questions?

• How SAT solving works.
  – Propositional Satisfiability:
    * CNF, Search Trees, Unit Propogation, Learning
  – Satisfiability Modulo Theories:
    * Lazy, Eager, and Direct methods.
    * The DPLL[T] direct method.

• Implementing a SAT Solver for timed systems.
  – Difference constraints and constraint graphs.
  – Stack threaded constraint graphs.
  – Incremental negative cycle detection.
  – Round robin constraint propogation.

• Experiments.
  – Circuit stabilization, job-shop scheduling.
Thank you.