Reactive Synthesis

Lecture 6

Swen Jacobs and Martin Zimmermann
(Saarland University)

Plan for Today

- Basic Games
- Algorithms & Data Structures

- Advanced Games
- Temporal Logic Synthesis

Plan for Today

- Basic Games
- Algorithms & Data Structures
  - Algorithms based on CNF representation
- **Project: Second Phase**
- Advanced Games
- Temporal Logic Synthesis

Recap

To solve a safety game in symbolic representation, we can compute the (Player 1) attractor of the unsafe states.

The controllable predecessor of Player 1 becomes

$$\mathrm{CPre}_1(F) = \exists X_u \forall X_c \exists L'.\ F(L') \wedge T(L, X_u, X_c, L')$$

and the attractor is defined as (with $F = \mathit{Unsafe}$)

- $\mathrm{Attr}_1^0(F) = F$,
- $\mathrm{Attr}_1^{n+1}(F) = \mathrm{Attr}_1^n(F) \vee \mathrm{CPre}_1(\mathrm{Attr}_1^n(F))$, and
- $\mathrm{Attr}_1(F) = \bigvee_{n \in \mathbb{N}} \mathrm{Attr}_1^n(F)$.

ROBDDs support (efficiently) all operations needed for attractor computation

Recap

How hard is it to apply operations and stay in the given representation? Is this representation usually compact?

Representation	$\forall, \wedge$	$\exists, \vee$	equivalence	compact?
arbitrary formula	easy	easy	**hard**	often
DNF	**hard**	easy	easy	sometimes
CNF	easy	medium	medium	sometimes
BDD	medium	medium	easy	often

Disjunctive Normal Form (DNF): a disjunction of conjunctions of literals, i.e., $\bigvee(L_1 \wedge \ldots \wedge L_n)$

Conjunctive Normal Form (CNF): a conjunction of disjunctions of literals, i.e., $\bigwedge(L_1 \vee \ldots \vee L_n)$

BDD (Binary Decision Diagram): a graph-based representation of quantifier-free boolean formulas

QBF and SAT Solving

Conjunctive Normal Form (CNF)

A **literal** is a boolean variable or its negation.

A **clause** is a disjunction of literals, a **cube** is a conjunction of literals.

A propositional formula is in **conjunctive normal form** if it is a conjunction of clauses.

Conjunctive Normal Form (CNF)

A **literal** is a boolean variable or its negation.

A **clause** is a disjunction of literals, a **cube** is a conjunction of literals.

A propositional formula is in **conjunctive normal form** if it is a conjunction of clauses.

A quantified boolean formula (QBF) is in **prenex conjunctive normal form (PCNF)** if it consists of a quantifier prefix, followed by a propositional (quantifier-free) formula in CNF.

Variable Assignments, Satisfiability

A **variable assignment** is a mapping σ of boolean variables to truth values.

A **satisfying assignment** for a formula F is a variable assignment such that $\sigma(F) = 1$. Since σ can be seen as a substitution $[x_1 \leftarrow b_1, ..., x_n \leftarrow b_n]$, we also write $F\sigma$ for $\sigma(F)$.

Variable Assignments, Satisfiability

A **variable assignment** is a mapping σ of boolean variables to truth values.

A **satisfying assignment** for a formula F is a variable assignment such that $\sigma(F) = 1$. Since σ can be seen as a substitution $[x_1 \leftarrow b_1, \ldots, x_n \leftarrow b_n]$, we also write $F\sigma$ for $\sigma(F)$.

A formula F is **satisfiable** if there exists a satisfying assignment for F. Otherwise it is **unsatisfiable**.

Variable Assignments, Satisfiability

A **variable assignment** is a mapping σ of boolean variables to truth values.

A **satisfying assignment** for a formula F is a variable assignment such that $\sigma(F) = 1$. Since σ can be seen as a substitution $[x_1 \leftarrow b_1, ..., x_n \leftarrow b_n]$, we also write $F\sigma$ for $\sigma(F)$.

A formula F is **satisfiable** if there exists a satisfying assignment for F. Otherwise it is **unsatisfiable**.

Two formulas F_1, F_2 are **equivalent** if $F_1\sigma = F_2\sigma$ for every assignment σ.

Variable Assignments, Satisfiability

A **variable assignment** is a mapping σ of boolean variables to truth values.

A **satisfying assignment** for a formula F is a variable assignment such that $\sigma(F) = 1$. Since σ can be seen as a substitution $[x_1 \leftarrow b_1, ..., x_n \leftarrow b_n]$, we also write $F\sigma$ for $\sigma(F)$.

A formula F is **satisfiable** if there exists a satisfying assignment for F. Otherwise it is **unsatisfiable**.

Two formulas F_1, F_2 are **equivalent** if $F_1\sigma = F_2\sigma$ for every assignment σ.

Two formulas F_1, F_2 are **equisatisfiable** if they are either both satisfiable or both unsatisfiable.

Tseitin Transformation

Every propositional formula can be rewritten to an **equisatisfiable** formula in CNF, with a linear increase in length and introduction of a linear number of auxiliary variables. (Tseitin Transformation)

Tseitin Transformation

Every propositional formula can be rewritten to an **equisatisfiable** formula in CNF, with a linear increase in length and introduction of a linear number of auxiliary variables. (Tseitin Transformation)

We can also rewrite QBF formulas to equisatisfiable formulas in PCNF by using transformation to prenex form and the Tseitin transformation.

Minterms

A variable assignment can be represented by a cube: variables that appear negated are set to 0, unnegated ones to 1.

For a set of boolean variables X, an **X-minterm** is a cube that contains all variables from X either negated or unnegated, but not both.

Minterms

A variable assignment can be represented by a cube: variables that appear negated are set to 0, unnegated ones to 1.

For a set of boolean variables X, an **X-minterm** is a cube that contains all variables from X either negated or unnegated, but not both.

Minterms represent **complete** assignments of truth values to the boolean variables in X.

Satisfiability Checking (SAT Solving)

SAT solving is the process of determining whether a given propositional formula has a satisfying assignment.

Idea: Incrementally build satisfying assignment for F, distinguishing between **propagated literals** and **choice literals**.

Satisfiability Checking (SAT Solving)

SAT solving is the process of determining whether a given propositional formula has a satisfying assignment.

Idea: Incrementally build satisfying assignment for F, distinguishing between **propagated literals** and **choice literals**.

Basic algorithm:

- **Unit propagation**: a clause in which no literal has been assigned value 1, and in which only one literal has no assigned truth value is called a **unit clause**.
  If such a literal exists, add it as a propagated literal to σ.

Satisfiability Checking (SAT Solving)

SAT solving is the process of determining whether a given propositional formula has a satisfying assignment.

Idea: Incrementally build satisfying assignment for F, distinguishing between **propagated literals** and **choice literals**.

Basic algorithm:

- **Unit propagation**: a clause in which no literal has been assigned value 1, and in which only one literal has no assigned truth value is called a **unit clause**.
  If such a literal exists, add it as a propagated literal to σ.
- **Choices:** if no unit propagation is possible, then choose a truth value for one of the remaining variables. Add this as a choice literal to σ.

Satisfiability Checking (SAT Solving)

SAT solving is the process of determining whether a given propositional formula has a satisfying assignment.

Idea: Incrementally build satisfying assignment for F, distinguishing between **propagated literals** and **choice literals**.

Basic algorithm:

- **Unit propagation**: a clause in which no literal has been assigned value 1, and in which only one literal has no assigned truth value is called a **unit clause**.
  If such a literal exists, add it as a propagated literal to σ.
- **Choices:** if no unit propagation is possible, then choose a truth value for one of the remaining variables. Add this as a choice literal to σ.
- **Backtracking:** if we reach a complete variable assignment that is not satisfying, remove all literals that have been added since the last choice, and add the negation of this choice literal as a propagation literal.

Satisfiability Checking: Example

$$\text{Is} \quad F = \begin{array}{l} x_1 \vee x_2 \vee x_3 \\ \bar{x}_1 \vee \bar{x}_2 \\ \bar{x}_3 \\ x_4 \vee \bar{x}_2 \\ \bar{x}_1 \vee \bar{x}_4 \\ \bar{x}_1 \vee x_3 \end{array} \quad \text{satisfiable?}$$

Satisfiability Checking: Example

$$\text{Is} \quad F = \begin{array}{l} x_1 \vee x_2 \vee x_3 \\ \bar{x_1} \vee \bar{x_2} \\ \bar{x_3} \\ x_4 \vee \bar{x_2} \\ \bar{x_1} \vee \bar{x_4} \\ \bar{x_1} \vee x_3 \end{array} \quad \text{satisfiable?}$$

$\bar{x_3}$

Satisfiability Checking: Example

Is

$$F = \begin{array}{l} x_1 \lor x_2 \lor x_3 \\ \bar{x_1} \lor \bar{x_2} \\ \bar{x_3} \\ x_4 \lor \bar{x_2} \\ \bar{x_1} \lor \bar{x_4} \\ \bar{x_1} \lor x_3 \end{array}$$

satisfiable?

$\bar{x_3} x_1^c$

Satisfiability Checking: Example

$$F = \begin{array}{l} x_1 \vee x_2 \vee x_3 \\ \bar{x_1} \vee \bar{x_2} \\ \bar{x_3} \\ x_4 \vee \bar{x_2} \\ \bar{x_1} \vee \bar{x_4} \\ \bar{x_1} \vee x_3 \end{array}$$

Is F satisfiable?

$\bar{x_3} x_1^c \bar{x_2}$

Satisfiability Checking: Example

Is
$$F = \begin{array}{l} x_1 \vee x_2 \vee x_3 \\ \bar{x_1} \vee \bar{x_2} \\ \bar{x_3} \\ x_4 \vee \bar{x_2} \\ \bar{x_1} \vee \bar{x_4} \\ \bar{x_1} \vee x_3 \end{array}$$
satisfiable?

$\bar{x_3} x_1^c \bar{x_2} \bar{x_4}$ (does not satisfy last clause, so backtrack)

Satisfiability Checking: Example

Is $F = \begin{array}{l} x_1 \vee x_2 \vee x_3 \\ \bar{x_1} \vee \bar{x_2} \\ \bar{x_3} \\ x_4 \vee \bar{x_2} \\ \bar{x_1} \vee \bar{x_4} \\ \bar{x_1} \vee x_3 \end{array}$ satisfiable?

$\bar{x_3} x_1^c \bar{x_2} \bar{x_4}$ (does not satisfy last clause, so backtrack)

$\bar{x_3} \bar{x_1}$

Satisfiability Checking: Example

Is $F = \begin{array}{l} x_1 \vee x_2 \vee x_3 \\ \bar{x_1} \vee \bar{x_2} \\ \bar{x_3} \\ x_4 \vee \bar{x_2} \\ \bar{x_1} \vee \bar{x_4} \\ \bar{x_1} \vee x_3 \end{array}$ satisfiable?

$\bar{x_3} x_1^c \bar{x_2} \bar{x_4}$ (does not satisfy last clause, so backtrack)

$\bar{x_3} \bar{x_1} x_2$

Satisfiability Checking: Example

Is $F = \begin{array}{l} x_1 \vee x_2 \vee x_3 \\ \bar{x_1} \vee \bar{x_2} \\ \bar{x_3} \\ x_4 \vee \bar{x_2} \\ \bar{x_1} \vee \bar{x_4} \\ \bar{x_1} \vee x_3 \end{array}$ satisfiable?

$\bar{x_3} x_1^c \bar{x_2} \bar{x_4}$ (does not satisfy last clause, so backtrack)

$\bar{x_3} \bar{x_1} x_2 x_4$ (satisfying assignment)

SAT and QBF Solving

SAT solvers can solve huge problems, big improvements in last 15 years:

Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout

CPU Time (in seconds): 0, 200, 400, 600, 800, 1000, 1200

Number of problems solved: 0, 20, 40, 60, 80, 100, 120, 140, 160, 180

- Limmat (2002)
- Zchaff (2002)
- Berkmin (2002)
- Forklift (2003)
- Siege (2003)
- Zchaff (2004)
- SatELite (2005)
- Minisat 2 (2006)
- Picosat (2007)
- Rsat (2007)
- Minisat 2.1 (2008)
- Precosat (2009)
- Glucose (2009)
- Clasp (2009)
- Cryptominisat (2010)
- Lingeling (2010)
- Minisat 2.2 (2010)
- Glucose 2 (2011)
- Glueminisat (2011)
- Contrasat (2011)

QBF solving is a much younger research field, solvers are not as mature.

Reactive Synthesis based on QBF Solving

Notation

A call to a SAT solver to check satisfiability of a propositional formula F will be denoted $SAT(F)$.

$SAT(F)$ returns a truth value that corresponds to satisfiability of the formula. If the formula is satisfiable, it also returns a satisfying minterm.

Notation

A call to a SAT solver to check satisfiability of a propositional formula F will be denoted $SAT(F)$.

$SAT(F)$ returns a truth value that corresponds to satisfiability of the formula. If the formula is satisfiable, it also returns a satisfying minterm.

A call to a QBF solver on a QBF F will be denoted $QBFSAT(F)$, and has the same return type.

QBF-based Attractor Computation

Input: Symbolic Safety Game $G = (L, X_u, X_c, T, Unsafe)$
Output: Winning region of player 0 or "unrealizable"
QBFAttract(G)

1. $F(L) := \neg Unsafe(L)$
2. while $F(L)$ changes do
3. $\quad F(L) := F(L) \wedge (\forall X_u \exists X_c \exists L'. T(L, X_u, X_c, L') \wedge F(L'))$
4. $\quad$ if $QBFSAT(\mathrm{init}(L) \wedge \neg F(L))$ then return "unrealizable"
5. return $F(L)$

QBF-based Attractor Computation

Input: Symbolic Safety Game $G = (L, X_u, X_c, T, \mathit{Unsafe})$
Output: Winning region of player 0 or "unrealizable"
QBFAttract(G)

1. $F(L) := \neg \mathit{Unsafe}(L)$
2. while $F(L)$ changes do
3. $\quad F(L) := F(L) \wedge (\forall X_u \exists X_c \exists L'. T(L, X_u, X_c, L') \wedge F(L'))$
4. $\quad$ if $\mathit{QBFSAT}(\mathrm{init}(L) \wedge \neg F(L))$ then return "unrealizable"
5. return $F(L)$

Problems:

- "$F(L)$ changes" is check for equivalence between two formulas in PCNF, but representation is not canonical
- for the QBFSAT call, need to convert $\neg F$ into PCNF
- size and number of quantified variables in F grows with number of iterations

QBF-based Attractor Computation

Input: Symbolic Safety Game $G = (L, X_u, X_c, T, Unsafe)$
Output: Winning region of player 0 or "unrealizable"
QBFAttract(G)

1. $F(L) := \neg Unsafe(L)$
2. while $F(L)$ changes do
3. $\quad F(L) := F(L) \wedge (\forall X_u \exists X_c \exists L'. T(L, X_u, X_c, L') \wedge F(L'))$
4. $\quad$ if $QBFSAT(\mathrm{init}(L) \wedge \neg F(L))$ then return "unrealizable"
5. return $F(L)$

Problems:

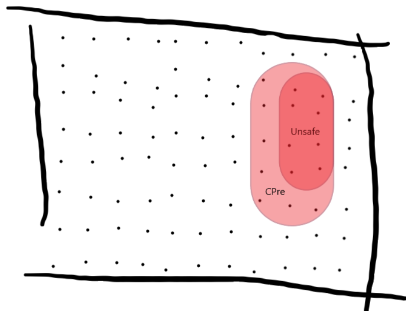
- "$F(L)$ changes" is check for equivalence between two formulas in PCNF, but representation is not canonical
- for the QBFSAT call, need to convert $\neg F$ into PCNF
- size and number of quantified variables in F grows with number of iterations

Can we compute attractor in a way that avoids adding new variables?

Alternative QBF-based Algorithm: Idea

Idea: identify states that can be added to the attractor as (partial) assignments to L

Instead of:

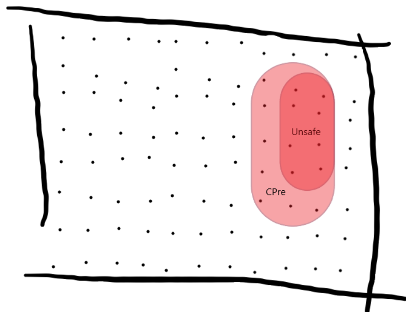


computing **all** forced predecessors as a QBF

Alternative QBF-based Algorithm: Idea

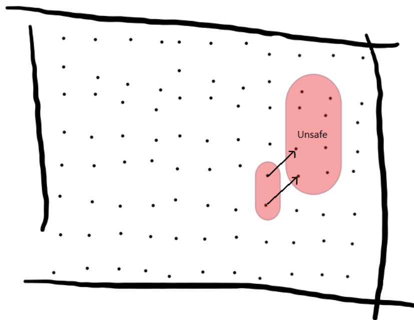
Idea: identify states that can be added to the attractor as (partial) assignments to L

Instead of:



all forced predecessors as a QBF

Do:



some of the forced predecessors as a cube over L

Algorithm QBFWin

Input: Symbolic Safety Game $G = (L, X_u, X_c, T, \mathit{Unsafe})$
Output: Winning region of player 0 or "unrealizable"
QBFWin(G)

1. if $\mathit{SAT}(\mathrm{init}(L) \wedge \mathit{Unsafe}(L))$ then return "unrealizable"
2. $F(L) := \neg \mathit{Unsafe}(L)$

Algorithm QBFWin

Input: Symbolic Safety Game $G = (L, X_u, X_c, T, Unsafe)$
Output: Winning region of player 0 or "unrealizable"
QBFWin(G)

1. if $SAT(\mathrm{init}(L) \wedge Unsafe(L))$ then return "unrealizable"
2. $F(L) := \neg Unsafe(L)$
3. while $sat = true$ in $(sat, s) :=$ $QBFSAT(\exists X_u \forall X_c \exists L'.\ F(L) \wedge T(L, X_u, X_c, L') \wedge \neg F(L')$ do

Algorithm QBFWin

Input: Symbolic Safety Game $G = (L, X_u, X_c, T, Unsafe)$
Output: Winning region of player 0 or "unrealizable"
QBFWin(G)

1. if $SAT(\mathrm{init}(L) \wedge Unsafe(L))$ then return "unrealizable"
2. $F(L) := \neg Unsafe(L)$
3. while $sat = true$ in $(sat, s) :=$ $QBFSAT(\exists X_u \forall X_c \exists L'.\ F(L) \wedge T(L, X_u, X_c, L') \wedge \neg F(L'))$ do
4.   if $SAT(s \wedge \mathrm{init}(L))$ then return "unrealizable"
5. $F(L) := F(L) \wedge \neg s$
6. return $F(L)$

Algorithm QBFWin

Input: Symbolic Safety Game $G = (L, X_u, X_c, T, \mathit{Unsafe})$
Output: Winning region of player 0 or "unrealizable"
QBFWin(G)

1. if $\mathit{SAT}(\mathrm{init}(L) \wedge \mathit{Unsafe}(L))$ then return "unrealizable"
2. $F(L) := \neg \mathit{Unsafe}(L)$
3. while $\mathit{sat} = \mathit{true}$ in $(\mathit{sat}, s) :=$ $\mathit{QBFSAT}(\exists X_u \forall X_c \exists L'.\ F(L) \wedge T(L, X_u, X_c, L') \wedge \neg F(L')$ do
4. if $\mathit{SAT}(s \wedge \mathrm{init}(L))$ then return "unrealizable"
5. $F(L) := F(L) \wedge \neg s$
6. return $F(L)$

Problem: only one state s is removed per iteration

Generalization of Satisfying Assignments

Idea: find partial assignments that satisfy a QBF formula by dropping literals from a satisfying minterm

Generalization of Satisfying Assignments

Idea: find partial assignments that satisfy a QBF formula by dropping literals from a satisfying minterm

If minterm $\mathbf{x}$ satisfies QBF formula F, then for all $lit \in \mathbf{x}$ do:

1. let $\mathbf{x}' := \mathbf{x} \setminus \{lit\}$
2. if $QBFSAT(\mathbf{x}' \wedge \neg F) = unsat$, then let $\mathbf{x} := \mathbf{x}'$

If the $QBFSAT$ call returns unsatisfiable, then **all** completions of $\mathbf{x}'$ satisfy F.

The algorithms use a more general form of this idea, where only part of the formula is negated (not negated is the part that defines the pre-states and the transition relation, negated is whether or not Player 1 can force the game into F in the next step).

Algorithm QBFWin with Generalization

Input: Symbolic Safety Game $G = (L, X_u, X_c, T, Unsafe)$
Output: Winning region of player 0 or "unrealizable"
QBFWinGen(G)

1. if $SAT(\mathrm{init}(L) \wedge Unsafe(L))$ then return "unrealizable"
2. $F(L) := \neg Unsafe(L)$
3. while $sat = true$ in **// get one state in the Player 1 attractor**
   $(sat, s) := QBFSAT(\exists X_u \forall X_c \exists L'.\ F(L) \wedge T(L, X_u, X_c, L') \wedge \neg F(L'))$
   do
4. $\quad s_g := s$
5. $\quad$ for each literal lit in s do **// generalize s**
6. $\quad\quad s_t := s_g \setminus \{lit\}$
7. $\quad\quad$ if $\neg QBFSAT(\forall X_u \exists X_c \exists L'.\ s_t \wedge F(L) \wedge T(L, X_u, X_c, L') \wedge F(L'))$
   $\quad\quad$ then $s_g := s_t$
8. $\quad$ if $SAT(s_g \wedge \mathrm{init}(L))$ then return "unrealizable"
9. $\quad F(L) := F(L) \wedge \neg s_g$ **// remove set of states s_g**
10. return $F(L)$

Remark

1. *Generalization depends on the order in which literals are picked. Different generalizations exist for different orders.*
2. $\mathrm{init}, T,$ *Unsafe and* $\neg$*Unsafe are translated into CNF once. Then, F will be in CNF by construction. However, each round we also need to translate $\neg F$ into CNF.*
3. *For efficient computations, the CNF representation of F should be simplified regularly.*

Remark

1. *Generalization depends on the order in which literals are picked. Different generalizations exist for different orders.*
2. init, T, *Unsafe and* $\neg$*Unsafe are translated into CNF once. Then, F will be in CNF by construction. However, each round we also need to translate* $\neg F$ *into CNF.*
3. *For efficient computations, the CNF representation of F should be simplified regularly.*

Problem: even if QBFSAT calls remain relatively simple, the number of calls is very large, and solvers are not very efficient.

Remark

1. *Generalization depends on the order in which literals are picked. Different generalizations exist for different orders.*
2. $\mathrm{init}, T,$ *Unsafe and* $\neg$*Unsafe are translated into CNF once. Then, F will be in CNF by construction. However, each round we also need to translate $\neg F$ into CNF.*
3. *For efficient computations, the CNF representation of F should be simplified regularly.*

Problem: even if QBFSAT calls remain relatively simple, the number of calls is very large, and solvers are not very efficient.

Can we use a similar approach with SAT solvers instead of QBF?

Reactive Synthesis based on SAT Solving

Unsatisfiable Cores

For an unsatisfiable formula F in CNF, a **clause-level unsatisfiable core** is a subset of the clauses of F that is still unsatisfiable.

Unsatisfiable Cores

For an unsatisfiable formula F in CNF, a **clause-level unsatisfiable core** is a subset of the clauses of F that is still unsatisfiable.

For a cube $\mathbf{x}$ and a formula F in CNF such that $\mathbf{x} \wedge F$ is unsatisfiable, an **unsatisfiable core** of $\mathbf{x}$ with respect to F is a subset $\mathbf{x}' \subseteq \mathbf{x}$ such that $\mathbf{x}' \wedge F$ is still unsatisfiable.

Unsatisfiable Cores

For an unsatisfiable formula F in CNF, a **clause-level unsatisfiable core** is a subset of the clauses of F that is still unsatisfiable.

For a cube $\mathbf{x}$ and a formula F in CNF such that $\mathbf{x} \wedge F$ is unsatisfiable, an **unsatisfiable core** of $\mathbf{x}$ with respect to F is a subset $\mathbf{x}' \subseteq \mathbf{x}$ such that $\mathbf{x}' \wedge F$ is still unsatisfiable.

An unsatisfiable core $\mathbf{x}'$ is **minimal** if no proper subset $\mathbf{x}''$ of $\mathbf{x}'$ makes $\mathbf{x}'' \wedge F$ unsatisfiable.

A call to a SAT solver for an unsatisfiable core of $\mathbf{x}$ with respect to F will be denoted $\mathit{UNSATCORE}(\mathbf{x}, F)$.

Algorithm SATWin

(**Input, Output** as before)

SATWin(G)

1. if $SAT(\mathrm{init}(L) \wedge Unsafe(L))$ then return "unrealizable"
2. $F(L) := \neg Unsafe(L)$, $U(L, X_u) := 1$

Algorithm SATWin

(**Input, Output** as before)

SATWin(G)

1. if $SAT(\mathrm{init}(L) \wedge Unsafe(L))$ then return "unrealizable"
2. $F(L) := \neg Unsafe(L)$, $U(L, X_u) := 1$
3. while *true* do
4. **// get one (s,u) that may be in the Player 1 attractor**
   $(sat, (s, u, _, _)) := SAT(F(L) \wedge U(L, X_u) \wedge T(L, X_u, X_c, L') \wedge \neg F(L'))$

Algorithm SATWin

(**Input, Output** as before)

SATWin(G)

1. if $\mathit{SAT}(\mathrm{init}(L) \wedge \mathit{Unsafe}(L))$ then return "unrealizable"
2. $F(L) := \neg \mathit{Unsafe}(L)$, $U(L, X_u) := 1$
3. while *true* do
4. **// get one (s,u) that may be in the Player 1 attractor**
   $(\mathit{sat}, (s, u, _, _)) := \mathit{SAT}(F(L) \wedge U(L, X_u) \wedge T(L, X_u, X_c, L') \wedge \neg F(L'))$
5. if $\neg \mathit{sat}$ then return $F(L)$
6. else **// check if it is really in attractor**
7. $(\mathit{sat}, (_, _, c, _)) := \mathit{SAT}(F(L) \wedge s \wedge u \wedge T(L, X_u, X_c, L') \wedge F(L'))$

Algorithm SATWin

(**Input, Output** as before)

SATWin(G)

1. if $SAT(\mathrm{init}(L) \wedge Unsafe(L))$ then return "unrealizable"
2. $F(L) := \neg Unsafe(L)$, $U(L, X_u) := 1$
3. while *true* do
4. **// get one (s,u) that may be in the Player 1 attractor**
   $(sat, (s, u, _, _)) := SAT(F(L) \wedge U(L, X_u) \wedge T(L, X_u, X_c, L') \wedge \neg F(L'))$
5. if $\neg sat$ then return $F(L)$
6. else **// check if it is really in attractor**
7. $(sat, (_, _, c, _)) := SAT(F(L) \wedge s \wedge u \wedge T(L, X_u, X_c, L') \wedge F(L'))$
8. if $\neg sat$ then **// it is in attractor**
9. $s_g := UNSATCORE(s, F(L) \wedge u \wedge T(L, X_u, X_c, L') \wedge F(L))$
10. if $SAT(s_g \wedge init(L))$ then return "unrealizable"
11. $F(L) := F(L) \wedge \neg s_g$, $U(L, X_u) := 1$

Algorithm SATWin

(**Input, Output** as before)

SATWin(G)

1. if $SAT(\mathrm{init}(L) \wedge Unsafe(L))$ then return "unrealizable"
2. $F(L) := \neg Unsafe(L),\ U(L, X_u) := 1$
3. while *true* do
4. **// get one (s,u) that may be in the Player 1 attractor**
   $(sat, (s, u, _, _)) := SAT(F(L) \wedge U(L, X_u) \wedge T(L, X_u, X_c, L') \wedge \neg F(L'))$
5. if $\neg sat$ then return $F(L)$
6. else **// check if it is really in attractor**
7. $(sat, (_, _, c, _)) := SAT(F(L) \wedge s \wedge u \wedge T(L, X_u, X_c, L') \wedge F(L'))$
8. if $\neg sat$ then **// it is in attractor**
9. $s_g := UNSATCORE(s, F(L) \wedge u \wedge T(L, X_u, X_c, L') \wedge F(L))$
10. if $SAT(s_g \wedge init(L))$ then return "unrealizable"
11. $F(L) := F(L) \wedge \neg s_g,\ U(L, X_u) := 1$
12. else **// it is not; refine U**
13. $U := U \wedge \neg UNSATCORE(s \wedge u, c \wedge F(L) \wedge U(L, X_u) \wedge T(L, X_u, X_c, L') \wedge \neg F(L'))$

Performance Comparison of BDD-based and QBF/SAT-based Algorithms

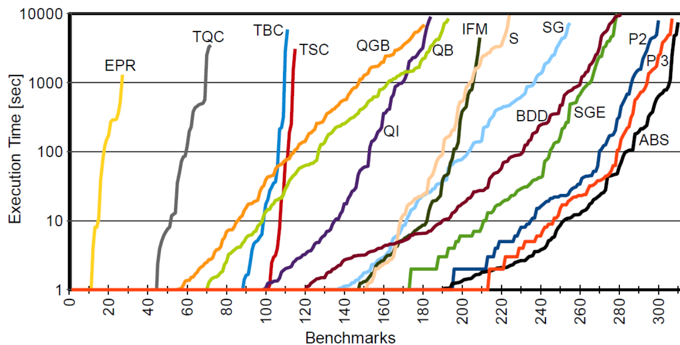


Figure 17: A cactus plot summarizing the execution times for computing a winning strategy with different methods and configurations.

BDD: basic BDD-based algorithm

ABS: state of the art BDD-based tool

Q*: QBF-based algorithms (with different optimizations)

S*: SAT-based algorithms (with different optimizations)

P*: Parallel execution of different QBF/SAT-based algorithms

Project: Phase II

Project Overview

Phase I:

We provided: a basic implementation (Python, CUDD)

You have:

- implemented new and improved existing procedures (based on lectures)
- provided an engineering notebook as documentation

Project Overview

Phase I:

We provided: a basic implementation (Python, CUDD)

You have:

- implemented new and improved existing procedures (based on lectures)
- provided an engineering notebook as documentation

Phase II:

We provide:

- a benchmarking framework
- additional suggestions for optimizations

You will:

- optimize for solving time and solution circuit size (based on suggestions and own ideas)
- choose optimizations for competition

Project: Important Dates

Check-point: December 12, 2017

- implement mandatory procedures and optimizations
- submit preliminary version and documentation

Project: Important Dates

Check-point: December 12, 2017

- implement mandatory procedures and optimizations
- submit preliminary version and documentation

Final submission: January 14, 2018

- implement additional optimizations
- choose optimizations for competition
- submit final version and documentation
  (for grading and competition)

Project: Important Dates

Check-point: December 12, 2017

- implement mandatory procedures and optimizations
- submit preliminary version and documentation

Final submission: January 14, 2018

- implement additional optimizations
- choose optimizations for competition
- submit final version and documentation
  (for grading and competition)

Competition results: January 30, 2018

- all tools run on a large set of challenging benchmarks
- we compare number of benchmarks solved and solution sizes
- competition results do not enter grading
  (but winners will get a prize)

Project Phase II: Suggestions for Optimization

Time:

- use **Hashtable** in `walk` function
- use **AndAbstract** instead of separate conjunction and quantification?

Project Phase II: Suggestions for Optimization

Time:

- use **Hashtable** in `walk` function
- use **AndAbstract** instead of separate conjunction and quantification?

Space:

- determine **initial variable order**?
- can BDDs that are not needed anymore be **de-allocated**?
- can we **avoid** building the transition relation (and using the primed variables) at all?
- what is the best dynamic **reordering strategy**?
- (when) does it make sense to **trigger reorderings directly**?

Project Phase II: Suggestions for Optimization

Size of Solution:

- use **reachability analysis** to further improve(?) the computed strategy
- strategy extraction: see Section 2.2, Fig.3 in Bloem et al., "Specify, Compile, Run: Hardware from PSL"
- does the order of variables in strategy extraction make a difference?

Project Phase II: Evaluation

We will provide additional benchmarks and a benchmarking framework.

You will:

- determine which of your optimizations improve performance
- choose a combination of optimizations that should run in the competition

Engineering Notebook

Keep recording your work on the program:

- Which part of the program did you work on?
- What was the goal?
- What was the result?

Record project meetings, including discussion items and action items that result from the meeting.

Record related work that influenced you, including books, research papers, webpages, source code, etc.
(be inspired, but do not copy code!)

Obtain a timestamp: Upload notebook to rcms at the end of the workday, **at least once a week**.

Format: simple textfile or pdf; **only add, never delete**.

Organization

No tutorial next week

Important dates:

December 19: no tutorial

December 20: lecture

January 2, 2018: no tutorial

from January 3, 2018: regular lecture and tutorial schedule

January 14, 2018: final tool submission

January 30, 2018 (tutorial slot): tool competition results

January 31, 2018 (lecture slot): final exam