EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 2/11: Satisfiability Modulo Theories, Part I

Scribe: Daniel Bundala *Editor: Sanjit A. Seshia*

Satisfiability modulo theories (SMT) is the study of the satisfiability of logical formulas (typically in first-order logic) with respect to (combinations of) background theories. The roots of SMT go back to work on *automatic theorem proving* and *decision procedures* for such logics performed in the 1970s and 80s. The defining characteristic of SMT solvers is their mode of operation: a satisfiability-preserving, explicit or implicit encoding of the original formula into a Boolean satisfiability (SAT) formula.

In this lecture, we provide a brief introduction to the area of SMT solving. The focus is on providing illustrative examples and background on first-order logic and background theories. Subsequent lectures will cover techniques used in SMT solvers today. A more detailed survey of SMT can be found in a recent book chapter [1].

1 Example

Suppose we want to decide whether the output of the following two functions is always the same.

```
int f(int y) {
  int x, z;
  z = y;
  y = x;
  x = z;
  return x * x;
}


int g(int x) {
  return x * x;
}
```

Consider some symbolic value of y. The formula $(z = y \wedge y_1 = x \wedge x_1 = z \wedge r_1 = x_1 * x_1)$ encodes the updates to variables in function f — the "relational" semantics of f. Here x_1 and y_1 denote the new values of variables x and y after they are updated. Specifically, given

y, it asserts that r_1 equals the value of $f(y)$. Similarly, the formula $(r_2 = y * y)$ asserts that $r_2 = g(y)$. Hence, consider the formula:

$$\varphi = (z = y \wedge y_1 = x \wedge x_1 = z \wedge r_1 = x_1 * x_1) \wedge (r_2 = y * y) \wedge (r_1 \neq r_2)$$

By construction, the formula φ is satisfiable if and only if there is a value of y such that $r_1 \neq r_2$. That is, $f(y) \neq g(y)$ for some y. In other words, the formula is satisfiable if and only if the two functions are not equivalent.

Formula φ is an example of a formula in the theory of bit-vector arithmetic.

1.1 Comparison with SAT

Some of the differences between SAT and SMT are following:

	SAT	SMT
Input	Formula in propositional logic (usually in CNF)	Formula in first-order logic with background theories
Variables	Boolean $\{\text{true}, \text{false}\}$	Various types: Boolean, $\mathbb{Z}, \mathbb{N}, \mathbb{R}$, finite-precision arithmetic, data structures: arrays, memories, lists, trees, etc.
Solving Strategy	Conflict-driven clause learning (CDCL/DPLL/DLL)	Encoding to SAT (lazy/eager)

2 First-order logic

To introduce background theories, we first have to introduce first-order logic. *First-order logic (FOL)* is a formal notation for mathematical reasoning and knowledge representation. A formal logic consists of two parts:

- Syntax: that specifies how we write formulas
- Semantics: that specifies the meaning of formula

For example, for the Boolean (propositional) logic, syntax specifies which Boolean formulas are well formed. The semantics of a well-formed formula is a Boolean function (e.g. truth tables) that give the truth value for each assignment to variables. E.g., the semantics of formula $x_1 \wedge x_2$ is $(\text{true}, \text{true}) \to \text{true}, (\text{true}, \text{false}) \to \text{false}, (\text{false}, \text{true}) \to \text{false}, (\text{false}, \text{false}) \to \text{false}$.

In practice, propositional logic is too brittle and inefficient to formalize anything more complicated. E.g., to formalize the sentence "All penguins are birds" in propositional logic, we can introduce a variable p_i for each penguin which is true if and only if the with penguin is a bird. Then we can say

$$p_1 \wedge p_2 \wedge p_3 \wedge \ldots \wedge p_N$$

where N is the total number of penguins in the world. However, such formula is too big and does not work for a general N.

In first-order formula we can express the same statement more concisely by

$$\forall x \big(P(x) \implies B(x) \big)$$

where $P(x)$ is the *predicate* asserting "x is a penguin" and $B(x)$ is the predicate asserting "x is a bird".

2.1 Syntax

The syntax of first-order logic consists of

- Variables: e.g., x, etc
- Logical symbols: e.g., $\wedge, \vee, \implies$ etc.
- Quantifiers: $\exists, \forall$
- Uninterpreted functions and predicates e.g. P, B
- Equality: $=$

Formulas are built from these symbols in the natural way. The first three classes of symbols are called *logical symbols* and the last two are called *nonlogical symbols*.

We usually reserve the word *variable* to refer only to bound variables that is variables that are under the scope of a quantifier. E.g., x in $\forall x$ or y in $\exists y$. Otherwise, we use the term *symbolic constant* or a *free variable* e.g., y in $\forall x \,.\, x = y$.

2.2 Semantics

Uninterpreted functions need to satisfy only a single condition. Namely:

"For every uninterpreted function f and for each x and y it holds that
$x = y \implies f(x) = f(y)$."

We can think of predicates as a special class of uninterpreted functions where the the codomain (range) of the function is $\{\text{true}, \text{false}\}$.

Let $\mathcal{P}$ be the set of all predicates and $\mathcal{F}$ be the set of all function symbols. A *model* for $(\mathcal{P}, \mathcal{F})$ consists of

- *Universe* $\mathcal{U}$: a nonempty set of concrete values
- for all symbolic constants σ, a model assigns a concrete value $\sigma_{\mathcal{U}} \in \mathcal{U}$
- for each function $f \in \mathcal{F}$ of arity n assigns a concrete function $f_{\mathcal{U}} : \mathcal{U}^n \to \mathcal{U}$
- for each predicate $P \in \mathcal{P}$ of arity n assign a concrete function $P_{\mathcal{U}} : \mathcal{U}^n \to \{\text{true}, \text{false}\}$

Note that a model fixes a universe and given the meaning to functions and predicates, the meaning of all other formulas can be deduced in the natural way.

3 Background Theories

A *background theory* is a set of nonlogical symbols and the semantics associated with the symbols. We call this the *signature* of the theory.

An example of a background theory is the *theory of integer linear arithmetic*. The *domain* or the universe of the theory is the set of integers $\mathbb{Z}$ and it has the following signature $\langle 0, 1, +, -, \leq \rangle$ where the semantics of the symbols and operators is the usual arithmetic over integers. An example of a formula in the theory is $\forall x \forall y \big[x > y \implies \exists z \,.\, (y + z = x)\big]$

Integer linear arithmetic is also known as *Presburger arithmetic*, after the logician who axiomatized it. (Some treatments of Presburger arithmetic define it over the domain of natural numbers $\mathbb{N}$, but this is not a significant difference.)

It turns out, that Presburger Arithmetic can be formalized using the following axioms:

- $\forall x \,.\, x + 1 > 0$
- $\forall x, y \,.\, (x + 1 = y + 1 \implies x = y)$
- $\forall x \,.\, (x = x + 0)$
- $\forall x, y \,.\, (x + y) + 1 = x + (y + 1)$
- for any formula φ of Presburger arithmetic: $[\varphi(0) \wedge (\forall x \,.\, \varphi(x) \implies \varphi(x + 1))] \implies \forall x \,.\, \varphi(x)$

Note that the first four axioms are in first-order logic, however, the last axiom is not as it quantifies over all formula (hence, it is a *second-order* formula).

Other examples of commonly used theories are:

- Free Theory: Consists only of uninterpreted functions and equality – hence also abbreviated as "EUF"
- Difference Logic over $\mathbb{Z}, \mathbb{R}, \ldots$. The only predicates that are allowed are always of the form $x - y \leq c$ where x, y are variables and c is a constant.
- Linear arithmetic over $\mathbb{Z}, \mathbb{R}, \ldots$. This generalizes the above and allows predicates of the form $\sum_{i=1}^{n} a_i * x_i \leq b_i$ where a_i, b are constants and x_i are variables.
- Finite-precision bit-vector arithmetic: is interpreted over integers of fixed size, e.g., int32, int64,... Note that each such number is a sequence of bits and allows combinations of arithmetic and Boolean operators: $x + y == z$ & 0x0FFFFFF.
- Theory of arrays
- Theory of lists

The SMT-LIB effort [2] seeks to define a common format for specifying SMT formulas in various (combinations of) background theories, and provides complete definitions of supported theories as well as a large repository of benchmarks.

References

[1] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[2] Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://smt-lib.org/`.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 2/18: Satisfiability Modulo Theories, Part II

Scribe: Yu-Yun Dai *Editor: Sarah Chasins*

In the last lecture, we used a *Satisfiability modulo theories (SMT)* solver to determine whether two functions were equivalent. When we attempt this task with a SAT solver, time grows exponentially with the number of bits. If instead we use a SMT solver with uninterpreted functions, it takes a fraction of a second.

SMT-LIB [2] defines a common format for SMT problems. It offers standardized descriptions of background theories and promotes standardized input and output languages for use by many different SMT solvers. In class, we used the SMT solver **Z3** [1], which uses the SMT-LIB standard.

In this lecture, we cover several background theories, including (1) the theory of Equality and Uninterpreted Functions (EUF), (2) the theory of Difference Logic (DL) and (3) the theory of Arrays. We also cover decision procedures for selected theories, and how the choice of encoding affects results. The next lecture will cover combining theories.

1 Defining Theories

To define a theory, one must define the notion of a literal.

- *Literal* or *Atomic Predicate*: Formula involving no Boolean operators — i.e. only function and predicate symbols in the theory. The simplest syntactic formulas one could write in the theory.

- *Formula*: A Boolean combination of literals.

For example, consider the following formula in EUF: $(x = y) \vee (f(x) = g(x, y))$. In this formula, $(x = y)$ and $(f(x) = g(x, y))$ are literals, while $(x = y) \vee (f(x) = g(x, y))$ is a formula.

In each of the following sections, we will first define a theory of interest (T), and then describe the *decision procedure* for T. A decision procedure is an algorithm to decide the satisfiability of a conjunction of literals in T.

2 Theory of Equality and Uninterpreted Functions

2.1 Definition

- Terms: Either a symbolic constant (a 0-arity function) (Ex: x, y) or a function application (Ex: $f(x)$, $g(x, y)$). There is no real distinction, since the symbolic constant is just a 0-arity function, but it is convenient to have both terms.
- Literals: Equalities over terms (Ex: $x = g(x, y)$).

Although we only discussed the axiom of congruence in class, here we list all axioms of EUF. Reflexivity, symmetry, and transitivity pertain to equality. Congruence pertains to function applications.

- Reflexivity: $\forall x.(x = x)$
- Symmetry: $\forall x, y.(x = y) \rightarrow (y = x)$
- Transitivity: $\forall x, y, z.(x = y \wedge y = z) \rightarrow (x = z)$
- Congruence: Let f be an uninterpreted function symbol of arity k. Then $(x_1 = y_1 \wedge x_2 = y_2 \wedge ... \wedge x_k = y_k) \rightarrow f(x_1, x_2, ..., x_k) = f(y_1, y_2, ..., y_k)$

2.2 Decision Procedure

The satisfiability problem for a conjunction of literals in EUF is decidable in polynomial time using an algorithm called *congruence closure*, an algorithm built around repeatedly applying the congruence rule. To start, one builds the expression graph for the formula. We first ignore all disequalities in the formula, but use all the equalities to identify equivalence classes. We apply the congruence rule repeatedly until we reach a fixpoint and can no longer find more equalities. Once we reach the fixpoint, we check whether the disequalities in the formula are consistent with our equivalence class information. If yes, the formula is satisfiable. If no, the formula is unsatisfiable.

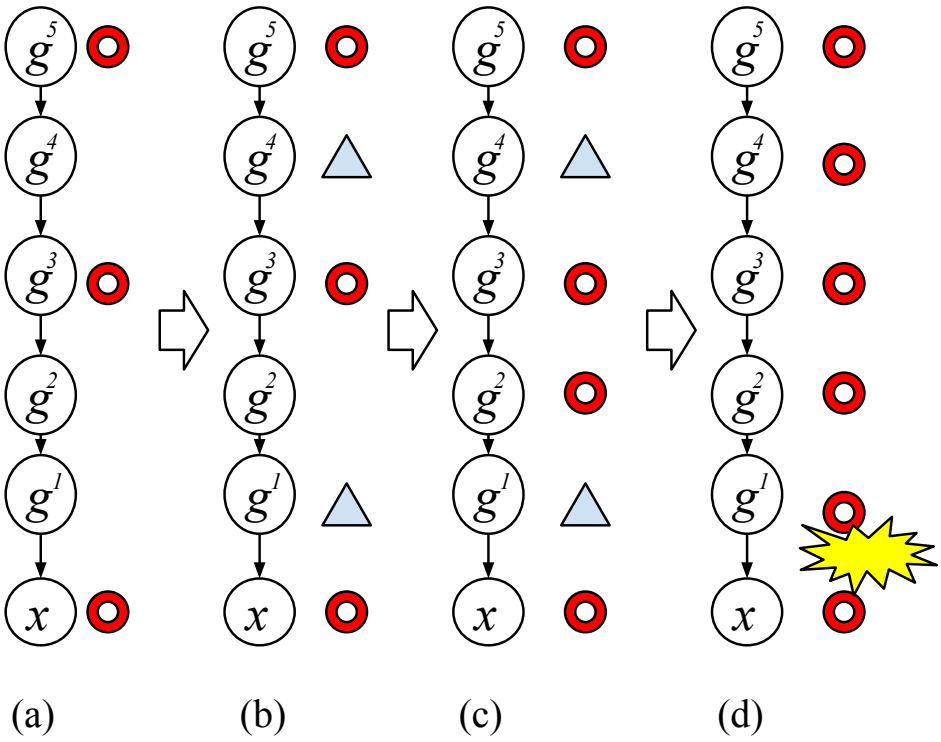
We illustrate this approach more concretely with an example. We use the following formula:

$$(g(g(g((x)))) = x) \wedge (g(g(g(g(g(x))))) = x) \wedge (g(x) \neq x)$$

For easy readability, we will use this notation:

$$(g^3(x) = x) \wedge (g^5(x) = x) \wedge (g(x) \neq x)$$

Figure 2.2 shows the expression graph for this formula, and provides a visualization of how the algorithm proceeds.



As shown in Figure 2.2(a), we start by incorporating the information in our formula, indicating with the node labels that x, $g^3(x)$ and $g^5(x)$ are part of a single equivalence class. Since g^3 and x are equivalent, by the congruence rule $g(g^3(x))$ and $g(x)$ must also be equivalent. Thus in (b), we indicate that $g^4(x)$ and $g^1(x)$ are equivalent. In (c), we apply the same axiom to $g^1(x)$ and $g^4(x)$; find that $g^2(x)$ and $g^5(x)$ are equivalent. Finally, we apply the congruence rule to x and $g^2(x)$ and reveal that both of the equivalence classes we have been maintaining so far are equivalent. Thus all nodes belong to the same equivalence class. The fact that x and $g(x)$ belong to the same equality class violates the inequality $(g(x) \neq x)$ in our formula, so the original formula is unsatisfiable.

2.3 Example

Recall the example of EUF in the last lecture, in which function f used a temporary variable to swap the values of x and y. In the following variation on f, we use bitwise XOR to perform the same swap.

```
int f(int y) {
  int x;
  x = x ^ y;
```

```
  y = x ^ y;
  x = x ^ y;
  return x * x;
}


int g(int x) {
  return x * x;
}
```

We can check whether f and g are equivalent with the following formula in EUF:

$$(x_1 = f(x, y)) \wedge (y_1 = f(x_1, y)) \wedge (x_2 = f(x_1, y_1)) \wedge (r_1 = g(x_2)) \wedge (r_2 = g(y)) \wedge (r_1 \neq r_2),$$

This formula is intended to encode the same problem that we encoded with EUF when we used a temporary variable. With that encoding, the SMT solver concluded that f and g were equivalent. However, with this encoding the SMT solver can find an assignment that satisfies this formula, indicating that the functions are not equivalent. This occurs because the encoding abstracts away too much information about XOR. It gives the SMT solver freedom to assign a meaning to XOR that is inconsistent with its actual meaning, but which allows f and g to diverge. In this case, using an uninterpreted function to represent XOR abstracts away too much information.

To address this problem, we could return to the EUF encoding that relied on equality to establish the swap semantics, or we could use an interpreted function for XOR.

3 Theory of Difference Logic

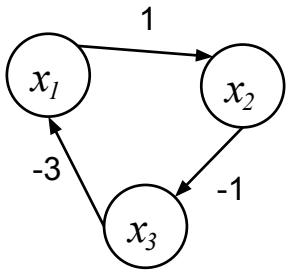
Difference logic (DL) is a fragment of linear arithmetic logic. One can use Rational Difference Logic (RDL) or Integer Difference Logic (IDL).

3.1 Definition

- Literals: $x_i - x_j \leq C_{ij}$, where x_i and x_j are symbolic constants, and C_{ij} is a concrete constant (an integer or a rational).

Note that DL can also express constraints on a single variable, such as $x_i \leq C_i$, by using the literal $x_i - x_0 \leq C_i$, where x_0 stands for our 0 item.

All axioms of arithmetic apply, but the literals are restricted.



3.2 Decision Procedure

To solve a DL formula, we use a *constraint graph*. To build our constraint graph, we add a node to represent each symbolic constant x_i in our formula. For each literal $x_i - x_j \leq C_{ij}$ we add an edge from x_i to x_j, with weight C_{ij}. Thus, nodes correspond to symbols and edges correspond to constraints.

Theorem: A DL formula is satisfiable if and only if there is no negative-weight cycle in its constraint graph.

Proof Sketch:

- Soundness: If there is a negative-weight cycle, then the constraints are unsatisfiable.
  - Consider what constraints a negative-weight cycle would imply. Take the cycle in Figure 3.2 as an example. The corresponding atoms are: $(x_1 - x_2 \leq 1) \wedge (x_2 - x_3 \leq -1) \wedge (x_3 - x_1 \leq -3)$. If we add the three atoms together, we must satisfy $0 \leq -3$, which is impossible.
- Completeness: If there is no negative-weight cycle, then the constraints are satisfiable.
  - We can compute the shortest paths between any x_i and x_0 using a shortest path algorithm; the length of the shortest path from x_0 to x_i gives the satisfying assignment for x_i.

We can detect the presence of negative weight cycles with any shortest path algorithm that handles negative weights — for instance, the Bellman-Ford algorithm.

4 Theory of Arrays

Memories show up in hardware and software alike. Further, many of data structures can be represented with arrays: lists, trees, graphs, complicated objects. So it is worth being able to represent arrays.

4.1 Definition

The theory of Arrays uses two interpreted functions, *select* (alternatively known as *read*) and *store* (alternatively known as *write*). It uses three abstract domains: *Addr*, *Data*, and *Array*.

- *select*: $select(Array * Addr) = Data$. *select* maps an array and an address to data.
- *store*: $store(Array * Addr * Data) = Array$. *store* maps an array, an address, and data to another array.

The theory of Arrays proposed by McCarthy in the 1960s uses the following "read-over-write" axioms:

- $(store(A, i, d), i) = d$.
- $(store(A, i, d), j) = select(A, j)$, when $i \neq j$

For some uses, it may be helpful to add the additional (optional) extensionality axiom:

- $[\forall i, select(A, i) = select(B, i)] \rightarrow A = B$

For applications in which having the same contents does not imply that two arrays are the same object, it is crucial to avoid using the extensionality axiom.

4.2 Example

We can use the theory of Arrays to produce another encoding of the function equivalence problem from last lecture.

```
int f(int y) {
  int x[2];
  x[0] = y;
  y = x[1];
  x[1] = x[0];
  return x[1] * x[1];
}

int g(int x) {
  return x * x;
}
```

The corresponding SMT formula is

$$\varphi = (x_1 = store(x, 0, y)) \wedge (y_1 = select(x_1, 1)) \wedge (x_2 = store(x_1, 1, select(x_1, 0)))$$
$$\wedge (r_1 = sq(select(x_2, 1))) \wedge (r_2 = sq(y)) \wedge (r_1 \neq r_2).$$

However, using the theory of Arrays for this problem would be unnecessary. In this case, we could simply use two different variables for $x[0]$ and $x[1]$ and solve with EUF.

5 Choices of Encoding

Here we explore differences in encoding within a single theory. We start with the sample program below:

```
int a[10];
int fun3(int i){
  int j;
  for( j = 0; j < 10; j++) a[j] = j;
  assert(a[i] <=5 )
}
```

We will use SMT solving to check whether the assertion can be violated.

We can encode the problem as follows:

$\varphi_1 = (a_1 = store(a, 0, 0)) \wedge (a_2 = store(a_1, 1, 1)) \wedge ... \wedge (a_{10} = store(a_9, 9, 9)) \wedge (select(a_{10}, i) > 5)$

This way of describing the array introduces many arrays, a new one for each *store*. Alternatively, we could use the following formula:

$$\varphi_2 = (select(a, 0) = 0) \wedge (select(a, 1) = 1) \wedge ... \wedge (select(a, 9) = 9) \wedge (select(a, i) > 5)$$

This variation only refers to one array, and does not use any *store* operations. Both the formula and the output are more compact and readable.

The choice of encoding matters a great deal. A user must pick both a background theory (or set of background theories) and, within the selected theory, how any given problem should be encoded. The success of the solver depends on both of these encoding choices.

References

[1] Microsft Research. Z3: An Efficient Theorem Prover `http://z3.codeplex.com/`.

[2] Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://smt-lib.org/`.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 2/23: Satisfiability Modulo Theories, Part III

Scribe: Daniel Fremont *Editor: Yu-Yun Dai*

This lecture covers three main topics. First, we conclude our treatment of arrays with an alternative representation, called *parallel-update* arrays, to the standard *select-store* approach. Next we address the problem of combining theories: how to solve SMT formulae expressed by a combination of theories. Finally, we classify modern SMT solvers into *eager* and *lazy* methods, and give an overview of how lazy SMT solvers work (eager solvers will be discussed in the next lecture).

1 Parallel-Update Arrays

A *parallel-update* array is a non-standard representation of an array. In practical problems, parallel-update arrays are quite useful. Recall that the standard theory of arrays without extensionality [1] consists of the axiom:

$$\text{select}(\text{store}(a, i, d), j) = \begin{cases} d & i = j \\ \text{select}(a, j) & i \neq j. \end{cases} \tag{1}$$

We can interpret an array a as a function, $A : \mathit{Address} \to \mathit{Data}$, such that $A(i)$ is the data at address i. This then leads to the following interpretations of select and store:

$$\begin{aligned} \text{select}(a, i) &\leftrightarrow A(i) \\ \text{store}(a, i, d) &\leftrightarrow \lambda j.\, \text{ITE}(i = j, d, A(j)) \end{aligned}$$

Here we use the standard λ notation for functions. A lambda expression, $\lambda x.t$, is a function which takes x as a single input and substitutes it into the expression t.

This way of writing arrays only uses uninterpreted functions (A) and lambda expressions, so the select-store axiom is not required. Also, the expression $\lambda i.i$ can be used to represent an array which stores the value i in the i^{th} location, while a representation with universal quantification, $\forall i.\, \text{select}(a, i) = i$, would be required in the standard notation.

Here is a more elaborate example that comes up frequently in practice. Given an array A, we want to write a value $d(i)$ in all locations i which satisfy some condition $\psi(i)$. The resulting array is easily expressed in lambda notation: $\lambda j.\, \text{ITE}(\psi(j), d(j), A(j))$. Notice that the expression allows potentially changing multiple locations in A; this is where the name "parallel-update arrays" comes from.

Note: There was a question in class about why the *store* operator is required to model a program with arrays. Here is an example. Suppose we are trying to model the following program:

```
a[i] = 0;
a[j] = 42;
```

To model the program without arrays, since the variable `a` is assigned, we need to make two copies of it, to represent its states before and after the assignment. However, the following constraints are insufficient to encode the functionality of this program:

$$\text{select}(a_1, i) = 0 \wedge \text{select}(a_2, j) = 42.$$

Suppose we want to check this property: if `i` $\neq$ `j`, the program terminates with `a[i]` $= 0$. This property is clearly true. To verify it with the above constraint, we need to show that the following formula is always true:

$$(i \neq j \wedge \text{select}(a_1, i) = 0 \wedge \text{select}(a_2, j) = 42) \rightarrow \text{select}(a_2, i) = 0,$$

However, here is a counterexample: consider a situation where $i = 0$, $j = 1$, and a_1 and a_2 are arrays with the values 0 and 42 in all locations, respectively.

The problem is, the relationship between a_1 and a_2 is not modeled (and incidentally, the relationship between a_1 and the initial state of the array, a_0, is also missed). To do that, we need to use the store operation. The modified constraints are

$$a_1 = \text{store}(a_0, i, 0) \wedge a_2 = \text{store}(a_1, j, 42),$$

and then the formula for verifying the property becomes

$$(i \neq j \wedge a_1 = \text{store}(a_0, i, 0) \wedge a_2 = \text{store}(a_1, j, 42)) \rightarrow \text{select}(a_2, i) = 0.$$

The theory of arrays (axiom (1)) ensures this formula is true.

2 Combination of Theories

Formulae combining symbols from multiple theories are more expressive and applicable to real problems, so we are interested in checking satisfiability with respect to a combination of theories. For example, the following formula uses a combination of LIA (linear integer arithmetic) and EUF (equality and uninterpreted functions):

$$\phi = \ x \geq 1 \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2).$$

We cannot just use a congruence closure solver, for example, to test whether ϕ is satisfiable, since the solver cannot handle the relations, $\geq$ and $\leq$, or the constants, 1 and 2. On the other hand, an LIA solver cannot deal with the uninterpreted function f.

Obviously implementing solvers for all possible combinations of theories is impractical. We expect a modular framework, which puts solvers of individual theories together to get one that works for all of them simultaneously. The *Nelson-Oppen combination framework* [2] provides a general way of doing this among those theories satisfying certain conditions.

Recall that a formula ϕ over a theory T is satisfiable (T-satisfiable when the theory needs to be made explicit) iff there is a model (an interpretation of the function and predicate symbols) of T, such that ϕ evaluates to true. The notation, $m \models \phi$, represents that ϕ evaluates to true in m. Also, ϕ is satisfiable iff $\exists m.m \models \phi$.

Example: The formula $x \geq y \wedge y \geq 42$ in LIA is satisfiable, since it is true in the model $\{x = 42, y = 42\}$. The formula $x \geq y \wedge y \geq x \wedge x \neq y$ is unsatisfiable, since there does not exist any combination of $x, y \in \mathbb{Z}$ to satisfy it.

Definition 2.1 *A formula ϕ over the theory T is* valid *(T-valid), if it evaluates to true in all models of T: $\forall m.m \models \phi$.*

Example: Although $x \geq y \wedge y \geq 42$ is satisfiable as seen above, it is not valid: for example, it is false in the model $\{x = 45, y = 40\}$. The formula $x \geq 0 \rightarrow (x + x) \geq x$ is valid, since it is true for all $x \in \mathbb{Z}$.

Theorem 2.1 *A formula ϕ is T-valid iff $\neg\phi$ is not T-satisfiable.*

Proof: ϕ is true in a model iff $\neg\phi$ is false in that model. So ϕ is T-valid $\iff$ ϕ is true in every model of T $\iff$ $\neg\phi$ is false in every model of T $\iff$ $\neg\phi$ is not true in any model of T $\iff$ $\neg\phi$ is not T-satisfiable. ∎

Definition 2.2 *An* arrangement *of a set of variables S is a consistent set of equalities and disequalities between elements of S such that every pair of variables is made either equal or unequal.*

Example: If $S = \{x, y, z\}$, then $\{x = y, y = z, z = x\}$ is an arrangement of S. But $\{x = y, y = z, z \neq x\}$ is not an arrangement since it is inconsistent, and $\{x = y\}$ is not an arrangement since it doesn't specify whether or not $y = z$.

The conditions that must be satisfied to apply the Nelson-Oppen method are as follows:

1. Only quantifier-free formulae are considered.

   The Nelson-Oppen procedure cannot determine the satisfiability of a formula with quantifiers ($\exists$ and $\forall$). Note that if a formula has only existential quantifiers, like $\exists x \phi(x)$, where ϕ is quantifier-free, it is acceptable, because $\exists x \phi(x)$ is satisfiable iff $\phi(x)$ is.

2. The theory signatures must have no shared symbols other than equality.

   Nelson-Oppen deals with theories which are mutually independent except for the equality relation, so the combination of IDL (integer difference logic) and LIA violates this condition. The minus operator in IDL and the minus operator in LIA are the same, and likewise for the relations like $\leq$.

3. Each theory T must be *stably infinite*: every quantifier-free T-satisfiable formula must be true in some infinite model of T.

   This is true for many theories: in LIA, it follows immediately, since every model of LIA is infinite (there are infinitely many integers). Obviously this condition holds in EUF as well. However, it doesn't hold in the theory of bitvectors (BV). For example, the theory of 1-bit vectors has only finite models; the only models for the satisfiable formula $x = x$ are $\{x = 0\}$ and $\{x = 1\}$.

Suppose we have theories T_1 and T_2 satisfying the conditions above, and Sat_1 and Sat_2 are solvers for each of them respectively. The Nelson-Oppen procedure which determines if a formula ϕ is $(T_1 \cup T_2)$-satisfiable works as follows:

1. By introducing new variables, convert ϕ into an equisatisfiable formula $\phi_1 \wedge \phi_2$, where ϕ_1 only uses interpreted functions and relations in T_1, and ϕ_2 only uses those in T_2.

2. Compute S, the set of shared symbolic constants between ϕ_1 and ϕ_2 (by our assumptions, no other non-logical symbols are shared by these formulae).

3. For each arrangement Δ of S:

   (a) Run Sat_1 on $\phi_1 \wedge \Delta$.

   (b) Run Sat_2 on $\phi_2 \wedge \Delta$.

4. If both $\phi_1 \wedge \Delta$ and $\phi_2 \wedge \Delta$ are satisfiable for some Δ, then ϕ is $(T_1 \cup T_2)$-satisfiable. Otherwise, ϕ is not $(T_1 \cup T_2)$-satisfiable.

Example: Let's apply Nelson-Oppen to the formula ϕ from the beginning of this section. Among the four literals of ϕ, only $f(x) \neq f(1)$ and $f(x) \neq f(2)$ use non-logical symbols

from both LIA and EUF: the function f comes from EUF, while the symbols 1 and 2 come from LIA. By introducing new variables y and z, we can split ϕ into $\phi_{LIA} \wedge \phi_{EUF}$ as follows:

$$\phi_{LIA} = x \geq 1 \wedge x \leq 2 \wedge y = 1 \wedge z = 2$$
$$\phi_{EUF} = f(x) \neq f(y) \wedge f(x) \neq f(z).$$

Now ϕ_{LIA} is an ordinary LIA formula, and ϕ_{EUF} is an ordinary EUF formula, while $\phi_{LIA} \wedge \phi_{EUF}$ and ϕ are equisatisfiable.

Next we find the set of shared symbolic constants $S = \{x, y, z\}$, and go through all possible arrangements Δ:

1. $\{x = y, y = z, z = x\}$: inconsistent with ϕ_{LIA}, since $(1 \neq 2)$ under LIA, and so ϕ_{LIA} implies $(y \neq z)$
2. $\{x = y, y \neq z, z \neq x\}$: inconsistent with ϕ_{EUF}, since under EUF, $(x = y)$ implies $(f(x) = f(y))$, contradicting ϕ_{EUF}
3. $\{x \neq y, y = z, z \neq x\}$: inconsistent with ϕ_{LIA} as above
4. $\{x \neq y, y \neq z, z = x\}$: inconsistent with ϕ_{EUF}, since under EUF $(x = z)$ implies $(f(x) = f(z))$, contradicting ϕ_{EUF}
5. $\{x \neq y, y \neq z, z \neq x\}$: inconsistent with ϕ_{LIA}: to satisfy ϕ_{LIA}, the only possible values for x are 1 and 2 (x must be an integer), so either $(x = y)$ or $(x = z)$ is true.

Since for each Δ either $\phi_{LIA} \wedge \Delta$ or $\phi_{EUF} \wedge \Delta$ is unsatisfiable, Nelson-Oppen correctly concludes that ϕ is unsatisfiable.

As we have presented, this technique requires going through potentially exponentially-many arrangements Δ. Refinements of the Nelson-Oppen method have been developed to avoid this in some cases.

3 SMT Solving Strategies

We begin with a little history. In the early 90s, Greg Nelson wrote the theorem prover Simplify. It was a widely-used tool, but was weak at Boolean reasoning. When SAT solvers got dramatically faster around 2000, many groups tried to see if those solvers could be used to accelerate theorem proving. This was achieved by various people simultaneously in 2002, and SMT was born. The basic idea was to convert an SMT formula ϕ_{SMT} into a propositional formula ϕ_{SAT} that is *equisatisfiable*: ϕ_{SMT} is satisfiable (with respect to its background theory) iff ϕ_{SAT} is satisfiable.

SMT solvers can be categorized into two classes: eager and lazy. Eager solvers encode all the necessary properties of background theories into an SAT problem upfront; they produce one big formula, which is then passed to an SAT solver to check satisfiability. Lazy solvers, in contrast, use an iterative approach where the background theory reasoning (a "theory lemma") is encoded into SAT on demand. Depending on the theory, either an eager or a lazy approach can be more effective.

3.1 Lazy Solvers

The basic idea behind lazy SMT solvers is to treat the input formula as being propositional until forced to do otherwise. For example, consider an EUF formula $\phi_{SMT} = f(x) = f(y) \wedge f(x) \neq f(y)$. If we ignore the explicit meaning of the literal $f(x) = f(y)$ and regard it just as a proposition p, then the formula becomes $\phi_{SAT} = p \wedge \neg p$. The obtained propositional formula can be passed to an ordinary SAT solver. According to the SAT solver, ϕ_{SAT} is unsatisfiable, and therefore so is the original formula ϕ_{SMT}. In this case, determining the satisfiability of ϕ_{SMT} can be done without knowing the meaning of f.

However, this approach seldom gets results so easily as above. Considering an LIA formula, $\phi_{SMT} = x = 0 \wedge x = 42$, its Boolean abstraction $\phi_{SAT} = p \wedge q$ is satisfiable, but ϕ_{SMT} is clearly unsatisfiable. The reason is, different literals in ϕ_{SAT} can involve the same terms in ϕ_{SMT}, so that the literals cannot be treated as totally independent. Here the needed background theory knowledge, the *theory lemma*, indicates that p and q are inconsistent because $x = 0$ and $x = 42$ are in conflict with each other.

One way to represent the conflict of $x = 0$ and $x = 42$ is adding a Boolean constraint, $p \rightarrow \neg q$. Notice that if we add this new information to ϕ_{SAT}, such that ϕ_{SAT} becomes $p \wedge q \wedge (p \rightarrow \neg q)$, then ϕ_{SAT} is unsatisfiable.

Figure 1 shows the general procedure of lazy SMT solvers. First of all, the input formula ϕ_{SMT} is abstracted into a propositional formula ϕ_{SAT} by introducing extra Boolean variables and ignoring dependencies of literals. If ϕ_{SAT} is unsatisfiable, so is ϕ_{SMT}, and the procedure terminates. Otherwise, the SAT solver returns a satisfying assignment $\vec{x}_{SAT}$ which asserts literals in ϕ_{SMT}.

Then, we construct an SMT formula, ψ_{SMT}, which is the conjunction of the literals asserted true by $\vec{x}_{SAT}$, as well as the negations of those asserted false. This conjunction of theory literals is checked by the theory solver. If ψ_{SMT} is satisfiable, the model also satisfies ϕ_{SMT} and the procedure terminates.

Otherwise, ψ_{SMT} is unsatisfiable, which means $\vec{x}_{SAT}$ results in conflicts among theory literals. In this case, a theory lemma, ℓ, which accounts for the unsatisfiability, is extracted from the unsatisfiability proof and added back to ϕ_{SMT}.

After refining ϕ_{SMT}, we restart the above procedure from the first step. The theory lemmas

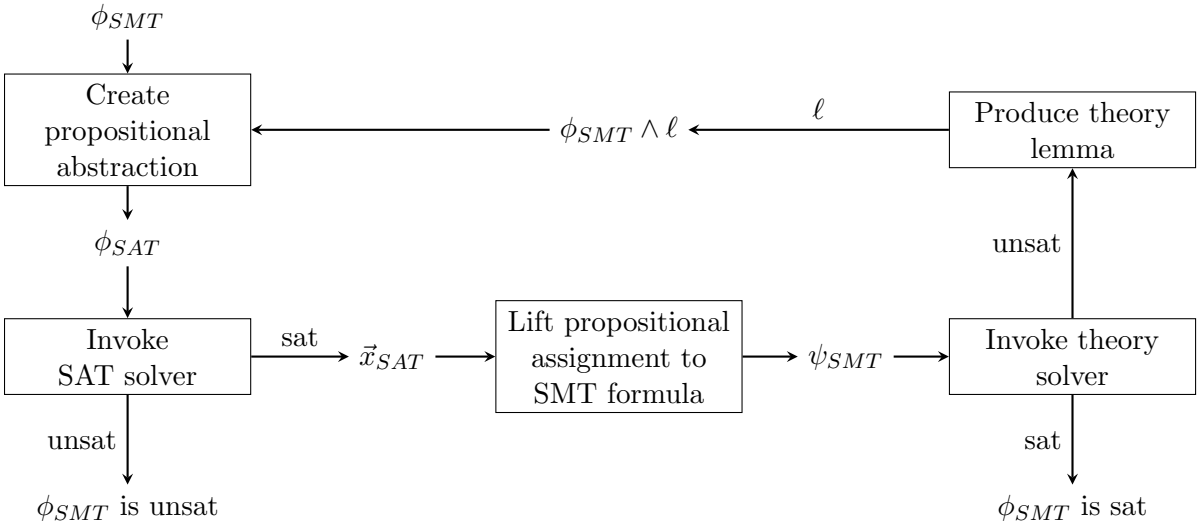


Figure 1: Basic structure of a lazy SMT solver.

we add in previous runs block out the failed assignments in the future runs, so eventually the procedure will determine the satisfiability of ϕ_{SMT}.

Example: Consider the EUF formula

$$\phi_{SMT} = [g(a) = c] \wedge [f(g(a)) \neq f(c) \vee g(a) = d] \\ \wedge [c \neq d \vee g(a) \neq d] .$$

First, we introduce new Boolean variables, x_1, x_2, x_3, x_4, to represent theory terms in the SMT formula. Then the propositional abstraction of ϕ_{SMT} is

$$\phi_{SAT} = x_1 \wedge (x_2 \vee x_3) \wedge (x_4 \vee \neg x_3).$$

Solved by an SAT solver, ϕ_{SAT} gets a satisfying assignment, $\{x_1, x_2, x_3, x_4\}$. This corresponds to the following conjunction of SMT literals:

$$\psi_{SMT} = (g(a) = c) \wedge (f(g(a)) \neq f(c)) \wedge (g(a) = d) \wedge (c \neq d).$$

Then, the EUF solver returns that ψ_{SMT} is unsatisfiable: the first two literals violate the congruence axiom. Therefore, we learn a theory lemma, $\ell = \neg x_1 \vee \neg x_2$, stating that the first two literals are inconsistent. Then the refined propositional abstraction is

$$\phi_{SAT} = x_1 \wedge (x_2 \vee x_3) \wedge (x_4 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2).$$

This is still satisfiable, via the model $\{x_1, \neg x_2, x_3, x_4\}$. The corresponding theory formula is

$$\psi_{SMT} = (g(a) = c) \wedge (f(g(a)) = f(c)) \wedge (g(a) = d) \wedge (c \neq d).$$

The EUF solver still returns that ψ_{SMT} is unsatisfiable: the first and third literals imply $(c = d)$, contradicting the last literal. Another theory lemma, $\ell = \neg x_1 \vee \neg x_3 \vee \neg x_4$ is added to ϕ_{SAT}. The new abstraction is

$$\phi_{SAT} = x_1 \wedge (x_2 \vee x_3) \wedge (x_4 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4).$$

Finally, the SAT solver returns that this formula is unsatisfiable, so the original formula, ϕ_{SMT}, is also unsatisfiable.

References

[1] John McCarthy. Towards a mathematical science of computation. In *Proc. IFIP Congress*, pages 21–28, 1962.

[2] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 2/25: SMT Part III And Syntax Guided Synthesis

Scribe: Shromona Ghosh *Editor: Julie Newcomb*

In the last lecture, we have studied the basics of *Satisfiability modulo theories (SMT)*. We have studied the different background theories, and how to combine multiple background theories. The previous lecture introduced 2 approaches of solving *SMT* by converting into a SAT problem, the *Lazy* approach and *Eager* approach.

In this lecture, we solve an example problem with *Lazy* approach and *Eager* approach. We also show a framework which outlines how a general *SMT* problem is solved. We end the lecture with an introduction to *Syntax Guided Synthesis*.

1 Lazy Approach

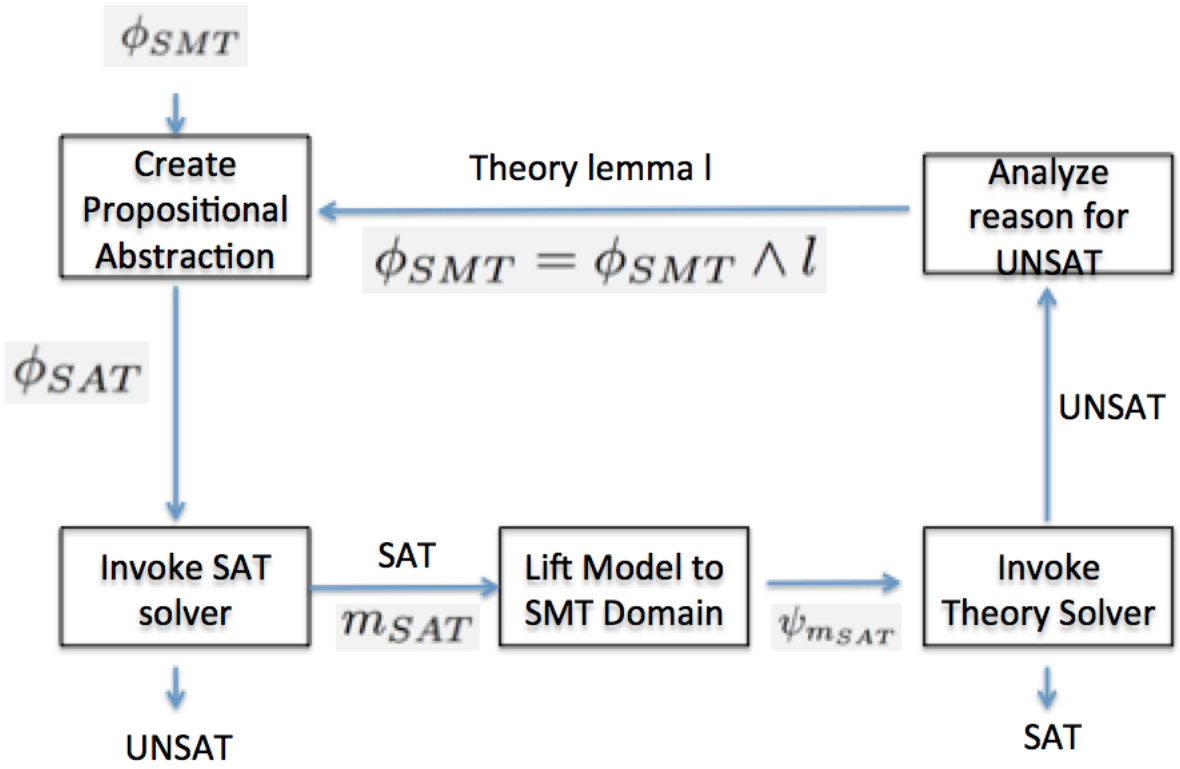
Figure 1 shows the overall framework of the *Lazy* approach. Here, the SMT problem ϕ_{SMT} is encoded into a SAT problem ϕ_{SAT} by the *Create propositional abstraction* block. This SAT problem is now solved invoking a SAT solver. If the SAT solver returns UNSAT, then we deem the original SMT also UNSAT. However, if the problem is SAT, we take the satisfying assignments m_{SAT} and lift it to the SMT domain to get $\psi_{m_{SAT}}$. We send this through a theory solver. If the solver returns SAT, the original SMT problem has a satisfying assignment. If UNSAT, we try to learn the reason for UNSAT. This can be seen as a process of theory learning. We can now introduce a theory lemma l in the form of a new constraint in the SMT problem.

The interesting part of this framework is the *Theory Learning* block. The theory solver can give us a proof of unsatisfiabilty. A new clause is added by backtracking to the last decision variable in the SAT solver and introducing this as a new constraint. This approach is naive and recently many new approaches have been studied.

Another approach to detecting clauses is called Early Conflict Detection, here the theory solver interacts with the SAT solver. The SMT solver does not wait for the SAT solver to make a complete assignment, and can insert new clauses as when it can detect a conflict. We can see this as the SMT and SAT solver working concurrently.

Another approach is known as theory propogation, here we use a theory engine to make decisions in the SAT solver. The theory solver makes inferences. Here the solver doesn't make a decision, it only makes inferences with respect to present assignments.

Figure 1: Lazy Approach



1.1 Example

Suppose we want to find out if the given set of equations is satisfiable,

$$(g(a) = c) \wedge ((f(g(a)) \neq f(c)) \vee (g(a) = d)) \wedge ((c \neq d) \vee (g(a) \neq d))$$

We first encode this problem as a SAT problem,

$$\mathbf{1} = (g(a) = c)$$

$$\bar{\mathbf{2}} = (f(g(a) \neq f(c))$$

$$\mathbf{3} = (g(a) = d)$$

$$\bar{\mathbf{4}} = (c \neq d)$$

$$\bar{\mathbf{3}} = (g(a) \neq d)$$

The translation here into SAT is done independent of the theory being used in SMT.

The above example has been solved in the slides by Clark Barrett and Sanjit Seshia *An Introduction to Satisfiability Modulo Theory* using all three approaches mentioned in the previous section.

The SAT problem is now

$$(\mathbf{1}) \wedge (\bar{\mathbf{2}} \vee \mathbf{3}) \wedge (\bar{\mathbf{4}} \vee \bar{\mathbf{3}})$$

1.1.1 Naive SAT solving

We first use unit propagation and set all unit clauses and it's effects on the other clauses. We then make a decision on the remaining variables and propagates it's effect on the other clauses. Once we can get a assignment for all the variables, the theory solver checks if this satisfies the overall SMT problem. Is it finds a conflict, it introduces a lemma i.e a conflict clause and sends it back to the SAT problem. This continues until the SMT is satisfiable or the SAT solver returns unsatisfiable.

In this example we see, there is a unit clause $\mathbf{1}$. On setting $\mathbf{1}$, there are no effects on the other clauses. The SAT solver now makes a decision, $\bar{\mathbf{2}}$, which satisfies the second clause. It makes another decision, $\bar{\mathbf{4}}$, which satisfies the third clause. The SAT solver sends these assignments to the theory learn. The thoery learner detects a conflict. In EUF, we have

$$x = y \implies f(x) = f(y)$$

which is violated by $\mathbf{1}$ and $\bar{\mathbf{2}}$. Thus, it now introduces a new clause negating the present assignment as a conflict clause. The new clause is

$$\bar{\mathbf{1}} \vee \mathbf{2} \vee \mathbf{4}$$

The SAT solver now backtracks to the last assigned variable i.e $\mathbf{4}$ and chooses the other assignment i.e setting $\mathbf{4}$ and continues.

The step-by-step solving of the SMT problem is as shown-

$$
\begin{array}{rll}
\emptyset \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(UnitProp)} \\
\mathbf{1} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Decide)} \\
\mathbf{1}\bar{\mathbf{2}}^{\mathbf{d}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Decide)} \\
\mathbf{1}\bar{\mathbf{2}}^{\mathbf{d}}\bar{\mathbf{4}}^{\mathbf{d}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Theory Learn)} \\
\mathbf{1}\bar{\mathbf{2}}^{\mathbf{d}}\bar{\mathbf{4}}^{\mathbf{d}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \mathbf{4} & \implies \text{(Backjump)} \\
\mathbf{1}\bar{\mathbf{2}}^{\mathbf{d}}\mathbf{4} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \mathbf{4} & \implies \text{(UnitProp)} \\
\mathbf{1}\bar{\mathbf{2}}^{\mathbf{d}}\mathbf{4}\bar{\mathbf{3}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \mathbf{4} & \implies \text{(Theory Learn)} \\
\mathbf{1}\bar{\mathbf{2}}^{\mathbf{d}}\mathbf{4}\bar{\mathbf{3}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \mathbf{4}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \bar{\mathbf{4}} \vee \mathbf{3} & \implies \text{(Backjump)} \\
\mathbf{12} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \mathbf{4}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \bar{\mathbf{4}} \vee \mathbf{3} & \implies \text{(UnitProp)} \\
\mathbf{123}\bar{\mathbf{4}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \mathbf{4}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \bar{\mathbf{4}} \vee \mathbf{3} & \implies \text{(Theory Learn)} \\
\mathbf{123}\bar{\mathbf{4}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \mathbf{4}, \bar{\mathbf{1}} \vee \mathbf{2} \vee \bar{\mathbf{4}} \vee \mathbf{3}, \bar{\mathbf{1}} \vee \bar{\mathbf{2}} \vee \bar{\mathbf{3}} \vee \mathbf{4} & \implies \text{(Fail)} \\
fail &
\end{array}
$$

Here, the green clauses are those which are satisfied by the assignment while the red clauses are the conflict clauses inserted by the theory learner. Literals with subscript 'd' are decided by the SAT solver.

Instead of the theory learner negating the current assignment as a learned clause, it can return a minimised clause which contains only those terms which conflict. In this case, the conflict is

$$g(a) = c \implies f(g(a)) = f(c)$$

i.e $\mathbf{1} \implies \mathbf{2}$. Thus, the theory learner returns,

$$\bar{\mathbf{1}} \vee \mathbf{2}$$

The step-by-step solving of the SMT problem with minimized learned clauses is as shown-

$$\begin{array}{rll}
\emptyset \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(UnitProp)} \\
\mathbf{1} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Decide)} \\
\mathbf{1}\mathbf{2}^{\bar{\mathbf{d}}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Decide)} \\
\mathbf{1}\mathbf{2}^{\bar{\mathbf{d}}}\mathbf{4}^{\bar{\mathbf{d}}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Theory Learn)} \\
\mathbf{1}\mathbf{2}^{\bar{\mathbf{d}}}\mathbf{4}^{\bar{\mathbf{d}}} \| & \mathbf{1}, \{\bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} & \implies \text{(Backjump)} \\
\mathbf{1}\mathbf{2} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} & \implies \text{(UnitProp)} \\
\mathbf{1}\mathbf{2}\mathbf{3}\bar{\mathbf{4}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} & \implies \text{(Theory Learn)} \\
\mathbf{1}\mathbf{2}\mathbf{3}\bar{\mathbf{4}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2}, \bar{\mathbf{1}} \vee \bar{\mathbf{3}} \vee \mathbf{4} & \implies \text{(Fail)} \\
fail &&
\end{array}$$

Here, the green clauses are those which are satisfied by the assignment while the red clauses are the conflict clauses inserted by the theory learner. Literals with subscript 'd' are decided by the SAT solver.

1.1.2 Early Conflict Detection

Here the SAT solver interacts with the theory solver. As the SAT solver generates assignment, the theory solver can detect if there are an conflicts. Once it learns a conflict clause it sends it back to the SAT solver, which now considers this while solving.

In our example, when the SAT solver sets $\mathbf{1}$, and makes a decision to set $\bar{\mathbf{2}}$, the theory solver immediately detects the conflict and reports the learned clause

$$\bar{\mathbf{1}} \vee \mathbf{2}$$

back to the SAT solver.

The step-by-step solving of the SMT problem with early conflict detection is as shown-

$$
\begin{array}{rll}
\emptyset \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(UnitProp)} \\
\mathbf{1} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Decide)} \\
\mathbf{1}\bar{\mathbf{2}}^{\mathbf{d}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Theory Learn)} \\
\mathbf{1}\bar{\mathbf{2}}^{\mathbf{d}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} & \implies \text{(Backjump)} \\
\mathbf{12} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} & \implies \text{(UnitProp)} \\
\mathbf{123}\bar{\mathbf{4}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \mathbf{2} & \implies \text{(Theory Learn)} \\
\mathbf{123}\bar{\mathbf{4}} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}}, \bar{\mathbf{1}} \vee \bar{\mathbf{3}} \vee \mathbf{4} & \implies \text{(Fail)} \\
fail &
\end{array}
$$

Here, the green clauses are those which are satisfied by the assignment while the red clauses are the conflict clauses inserted by the theory learner. Literals with subscript 'd' are decided by the SAT solver.

1.1.3 Theory Propagation

Here we have theory engine within the SAT solver which makes inferences within the SAT solver based on the current assignment.

In our example, the SAT solver first sets **1** because of unit propagation. The theory engine sees that, $\mathbf{1} \implies \mathbf{2}$, and sets **2**. This causes the second clause to now become a unit clause, and the SAT solver sets **3**. The theory engine now infers that if,

$$g(a) = c \wedge g(a) = d \implies c = d$$

i.e $\mathbf{1} \wedge \mathbf{3} \implies \mathbf{4}$. This sets **4**, which causes the third clause to be satisfied and the overall SMT problem is UNSAT.

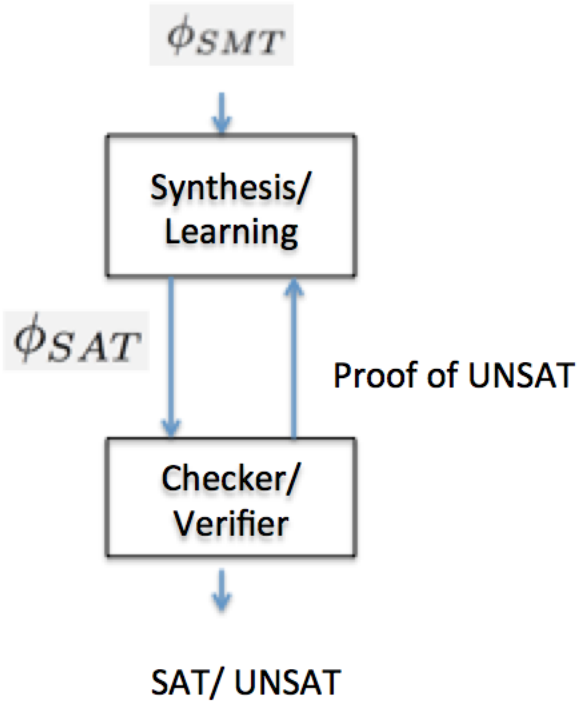
The step-by-step solving of the SMT problem with theory propagation is as shown-

$$
\begin{array}{rll}
\emptyset \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(UnitProp)} \\
\mathbf{1} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Theory Propagate)} \\
\mathbf{12} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(UnitProp)} \\
\mathbf{123} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Theory Propagate)} \\
\mathbf{1234} \| & \mathbf{1}, \bar{\mathbf{2}} \vee \mathbf{3}, \bar{\mathbf{4}} \vee \bar{\mathbf{3}} & \implies \text{(Fail)} \\
fail &
\end{array}
$$

Here, the green clauses are those which are satisfied by the assignment while the red clauses are those that are unsatisfied.

Figure 2 is a generalized abstraction of the lazy approach. This framework will be revisited later on in the course in the context of CEGIS and abstraction-refinement.

Figure 2: General Abstraction of Lazy Approach



2 Eager Approach

The key difference between Lazy and Eager approach is that, while converting ϕ_{SMT} to ϕ_{SAT} we also take into account the theories being used in SMT.

For example, suppose we have have a SMT problem with background theories EUF and LIA. We can encode the SMT problem into a SAT problem by encoding EUF first, and the LIA.

$$\phi_{EUF+LIA} \rightarrow \phi_{LIA} \rightarrow \phi_{BV} \rightarrow \phi_{SAT}$$

Each arrow signifies equi-satisfiability, i.e the left hand side is SAT if and only if the right hand side is SAT. We convert LIA to BV to introduce finite precision for all integers. This process is conceptually very similar to compilation or logic synthesis tools.

This can now be sent to a SAT solver, if the SAT solver returns SAT, then we do the reverse assignments to get the assignments to the original SMT problem.

$$m_{SAT} \rightarrow m_{BV} \rightarrow m_{LIA} \rightarrow m_{EUF+LIA}$$

2.1 Encoding

In the Eager approach, we need to encode the theories into the original SMT problem to get the SAT problem. There are two commonly used encoding schemes - Ackermann Encoding and Small Domain encoding.

2.1.1 Ackermann Encoding

In Ackerman encoding, which dates back to 1954, we eliminate the uninterpreted functions and all applications of uninterpreted functions. We show this encoding with the example from the previous section.

$$(g(a) = c) \wedge ((f(g(a) \neq f(c)) \vee (g(a) = d)) \wedge ((c \neq d) \vee (g(a) \neq d))$$

We identify all uninterpreted functions and replace them with a symbolic constant.

$$g(a) \rightarrow v_g$$

$$f(g(a)) \rightarrow v_f^1$$

$$f(c) \rightarrow v_f^2$$

ϕ_{SMT} now reduces to

$$(v_g = c) \wedge ((v_f^1 \neq v_f^2) \vee (v_g = d)) \wedge ((c \neq d) \vee (v_g \neq d))$$

However, replacing the uninterpreted functions with symbolic constants makes us lose the congruence property of uninterpreted functions which states,

$$a = b \implies f(a) = f(b)$$

We reintroduce this information by adding the following constraint,

$$v_g = c \implies v_f^1 = v_f^2$$

to get the final ϕ_{LIA} as,

$$(v_g = c) \wedge ((v_f^1 \neq v_f^2) \vee (v_g = d)) \wedge ((c \neq d) \vee (v_g \neq d)) \wedge (v_g = c \implies v_f^1 = v_f^2)$$

2.1.2 Small Domain Encoding

Theorem: Given SMT formula ϕ in theory T, ϕ is satisfiable iff there exists a satisfying solution to ϕ with all domains for function symbols having cardinality bounded by d_ϕ (a finite integer). Since the satisfying solutions are bounded, this implies that we can use an enumerative approach to finding a satisfying assignment. Making this form of SMT solving efficient often relies on taking advantage of special types of formulas. (Note however that not all theories admit small domain encodings.)

We illustrate the above theorem with the following example. Given a SMT formula,

$$x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_3 \neq x_1$$

where x_1, x_2 , $x_3 \in \mathbf{Z}$.

By the "folk theorem" for equality logic without uninterpreted functions, d_ϕ is the number of symbolic constants in ϕ; it is clear that if the formula above is UNSAT over a three-value domain, it will be UNSAT over all domains. From here, transformation into a SAT formula is trivial and a solution can potentially be found very quickly, without performing any solver iterations.

For a formula ϕ in difference logic theory, $d = n(b_{max}+1)$, where n is the number of variables in ϕ and b_{max} is the maximum over the absolute values of all difference constraints in ϕ. In the generalized 2SAT theory (linear constraints with at most 2 integer variables), $d = 2n(b_{max}+1)$. For the full linear integer arithmetic, where m is the number of inequalities, $d = m^m$. However, in software and hardware verification, constraints tend to be sparse and so smaller bounds can often be found.

There is as yet no theoretical basis for preferring the eager or the lazy approach to SMT solving. Empirically, particular benchmarks have been shown to be better suited to one or the other, however.

3 Syntax Guided Synthesis

Syntax Guided Synthesis is a program synthesis technique which allows the user to define the logical specifications in a syntactic template. Although introduced as a program synthesis technique, it is getting widespread acclaim in various other domains such as controller synthesis in hybrid systems. How to implement SyGus in practice is an open research problem, but the CEGIS approach is considered below.

We formally define the problem of *Syntax Guided Synthesis (SyGuS)* as:

Given: An SMT formula ϕ in theory T that references a vector of uninterpreted function symbols, $\vec{f}$ and a formal language vector $\vec{L}$ of expressions from T (this is usually expressed

as a context free grammar).

Find: Expression $\vec{e} \in \vec{L}$ such that

$$\phi[\vec{f} \leftarrow \vec{e}]$$

is T-valid.

Here, we can say that ϕ is a set of constraints which impose correctness on the program.

3.1 Example

$$\phi \triangleq f(x,y) = f(y,x) \wedge f(x,y) \geq x$$

with Theory T being LIA. We do not need to consider f an uninterpreted function, because we will use SyGuS to replace it.

Consider two languages: L_1 defined by grammar G_1 and L_2 defined by grammar G_2.

G_1:

$$\begin{aligned} Exp \ ::= \ & x \\ | \ & y \\ | \ & c \in \mathbf{Z} \\ | \ & Exp + Exp \end{aligned}$$

G_2:

$$\begin{aligned} Term \ ::= \ & x \\ | \ & y \\ | \ & (c \in \mathbf{Z} \\ | \ & ITE(Cond, Term, Term) \end{aligned}$$

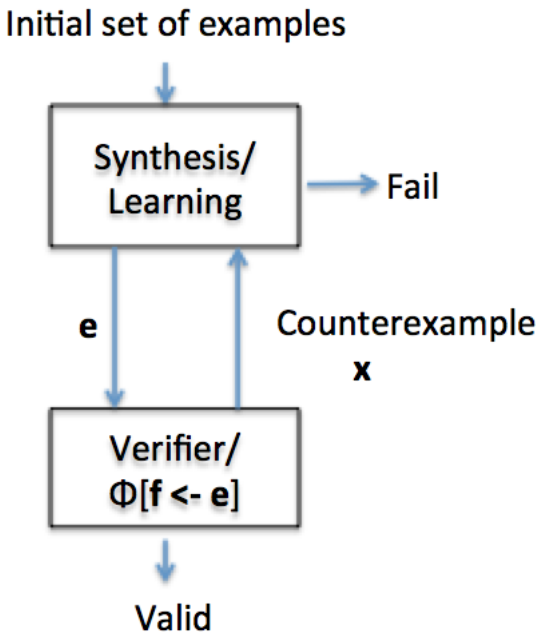
$$\begin{aligned} Cond \ ::= \ & Term \leq Term \\ | \ & Cond \wedge Cond \\ | \ & \bar{Cond} \\ | \ & (Cond) \end{aligned}$$

L_1, as defined in G_1, can produce any expression in the form $ax+by+c$, which will essentially rewrite ϕ into the form $ax + by + c = ay + bx + c \wedge ax + by + c \geq x$. Thus, G_1 fails to give an expression that satisfies the formula. However, G_2 produces a satisfying assigment for $f = ITE(x \geq y, x, y)$. As demonstrated here, the grammar acts as a syntax guidance.

3.2 Counter Example Guided Inductive Synthesis (CEGIS)

This is a specific class of SyGuS, where the counterexample returned by a synthesis tool acts as a learnt constraint. We show the framework in Figure 3.

Figure 3: CEGIS



A set of initial examples is given to the synthesizer. If the synthesizer is not able to produce a candidate program, it returns failure. If it is able to produce a candidate program, it is formally verified. If the verification returns a counterexample, it is added to the list of examples and sent to the synthesizer. This is repeated till the synthesizer can no longer synthesize candidate programs or verifier verifies a given candidate as valid.

4 Conclusion

Today SMT solvers are a mix of Lazy and Eager approaches. Eager techniques use some approaches used in Lazy approaches such as refinement and abstraction while many Lazy solvers have heruistics such as theory propogation common to Eager solvers. We encourage the reader to also read [2] for better understanding of how modern SMT solvers are built on top it.

SyGuS is a new approach program synthesis introduced in 2013. There is an online repository [3] of information about SyGuS. The original paper [1] concentrates on how CEGIS is used to solve the synthesis problem.

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[2] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[3] Syntax Guided Synthesis(SyGuS). Available at `http://www.sygus.org/`.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: Marta Kwiatkowska*

Lecture 3/2: Probabilistic Model Checking, Part I

Scribe: Dexter Scobee *Editor: Eric Kim*

Probabilistic model checking (PMC) is the application of model checking techniques to *probabilistic models* (not to be confused with probabilistic checking of deterministic models). These probabilistic models can arise in a variety of applications, from biological systems to wireless communication, and the details of the models can be derived from the high-level specifications of the system or extracted from probabilistic programs describing the systems.

In the first part of this lecture, we discuss some motivating examples for PMC and build up the basic framework for modeling these types of probabilistic, or *stochastic*, systems. The tools and techniques that are used to perform PMC on these probabilistic systems (including PRISM) are covered in the second part of the notes for this lecture. The accompanying slides for this lecture can be found on the EECS 219C bCourses page under "`Files -> Computer-Aided Verification -> Slides -> marta-ucb-pmc-feb15.pdf`."

1 Motivation

In order to begin model checking, it is necessary to first develop a model that captures the relevant behaviors of the system. In conventional model checking, the system is represented with a deterministic model, typically a *Finite-State Machine (FSM)*. In order to capture the probabilistic nature of the systems to which PMC is applied, it is necessary to use a probabilistic model such as a *Markov Chain*, described in more detail below.

Question: Why is it necessary to model probabilistic behavior?

Probabilistic behavior is evident in both man-made and naturally occuring areas. For instance, wireless communication standards intentionally use randomization in their protocols to improve performance. Biological processes often exhibit stochastic behavior as well, with molecular reactions occuring according to some probability distribution. Even in systems that may seem inherently deterministic, modeling or measuring these systems can introduce uncertainties that can be captured with probabilistic descriptions. Looking forward, there is potential for cross pollination of ideas and techniques from PMC and *Probabilistic Programming (PP)*.

Note: While a probabilistic model is not detrministic, it important to note that, in the domain of system modeling, *probabilistic* or *stochastic* systems are distinct from *non-*

deterministic systems. While probabilistic systems provide information about how they are likely to behave (usually in the form of probability distributions over possible state transitions), non-deterministic systems give no information about how the system will evolve when multiple transitions are possible.

2 Discrete-time Markov Chains

A *Discrete-time Markov Chain (DTMC)* is a set of states coupled with the probabilities of transitioning from one state to another. These discrete states represent possible configurations of the system being modeled, and the transitions between states occur in discrete time steps. For every state there is a probability distribution that describes the likelihood of transitioning from that state into each state (including the likelihood of remaining in the current state).

Mathematically, a DTMC can be represented as a 4-tuple $D = (S, s_{init}, P, L)$, where:

- S is a finite set of states (also known as the *state space*)
- $s_{init} \in S$ is the initial state
- $P : S \times S \to [0, 1]$ is the *transition probability matrix*
- $L : S \to 2^{AP}$ is the labeling function which assigns states a set of labels drawn from a set of atomic propositions AP.

In the matrix P, the entry $P(s, s')$ is the probability of transitioning from state s to state s'. This convention leads to the requirement that $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$. Furthermore, a DTMC is required to not contain any "deadlock" or "blocking" states, meaning that every state must have at least one possible transition, even if that transition is a *self loop* (a transition back to itself). Terminal states are accomodated in this framework by including such self loops with a transition probability of one. Figure 1 shows the example DTMC that was presented in class with a terminal state s_3.

Alternatively, we can view a DTMC as a family of random variables $\{X(k) \mid k = 0, 1, 2, ...\}$ where every $X(k)$ is an observation at a discrete time step. In this formulation, each random variable $X(k)$ corresponds to the state of the system at time k as described above. These variables (states) exhibit the *Markov Property*.

Definition 2.1 Markov Property - *Also known as* "memorylessness"*. Given the current state of the system, future states of the system are independent of the past. This can be*

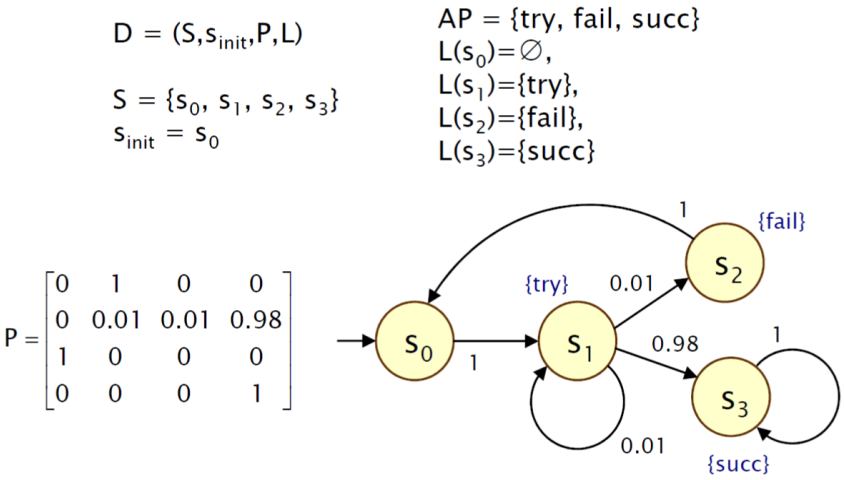


Figure 1: An example of a Discrete-time Markov Chain, taken from Marta Kwiatkowska's in-class presentation

alternatively understood as the next state depending only on the current state, or more formally:

$$Pr(X(k) = s_k \mid X(k-1) = s_{k-1}, \ldots, X(0) = s_0) = Pr(X(k) = s_k \mid X(k-1) = s_{k-1})$$

In working with DTMCs, there are additional assumptions that are used when working with or analyzing these models:

- *Time-Homogenous*: The transition probabilities are constant or independent of time. It is possible to have these probabilities be time-dependent, in which case the model is *time-inhomogenous*, but this change introduces additional complexity to model analysis.
- *Finite*: In most cases, the state space S is assumed to be finite but in general can be any countable set.
- *Initial States*: The concept of an intial state s_{init} can be also be extended to represent an *initial probability distribution* $s_{init} : S \to [0, 1]$.
- *Rational Transition Probabilities*: While transition probabilities $P(s, s')$ are real, they are assumed to be rationals in the algorithms discussed in this lecture (in practice this is not a restrictive assumption).

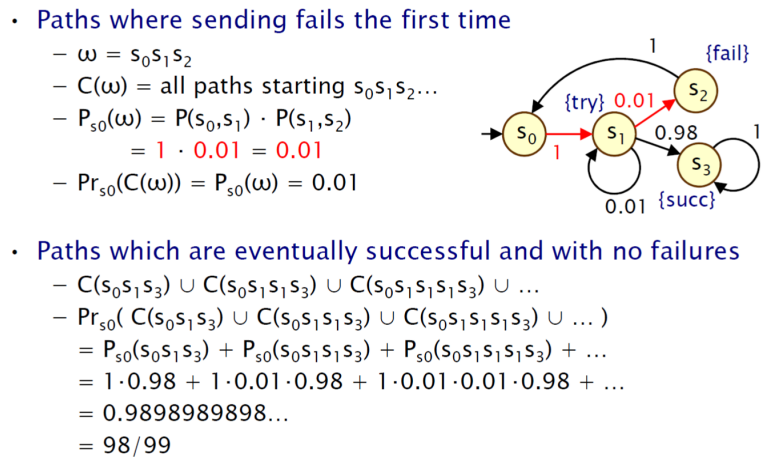


Figure 2: An example of a probability space calculation, taken from Marta Kwiatkowska's in-class presentation

3 Paths and Probability Space

In the context of DTMCs, we can consider a *path* through any given DTMC D. A path is a finite or infinite sequence of states that could possibly occur by allowing D to execute. In order for a path like $\{s_0, s_1, s_2, s_0, s_1, \ldots\}$ to be valid on a DTMC such as that in Figure 1, it must be the case that $P(s_i, s_j) > 0$ for all s_j immediately following s_i in the path. This requirement captures the condition that, for a sequence to be possible, there must be a non-zero probability of transitioning from a given state in that sequence to the next state in that sequence. Any valid path represents one possible behavior of the system being modeled by the DTMC.

In order to reason quantitatively about DTMC systems, we need to define a *probability space* over the possible paths of the system. For a given DTMC D, consider the sample space $\Omega = \text{Path}(s)$ which is the set of all infinite paths of D that begin with initial state s. Let ω be a finite path through D such that $\omega = \{s_0, s_1, \ldots, s_n\}$, then we can define the *cylinder set* $C(\omega) = \{\omega' \in \Omega \mid \omega \text{ is a prefix of } \omega'\}$. Note that $C(\omega)$ is a set of *infinite* paths ω' which have a common *finite* path ω as their prefixes.

Further define the *event set* Σ_Ω to be the *least σ-algebra* on Ω containing $C(\omega)$ for all finite paths ω starting with state s. We can then define a probability measure to reason about the likelihood of different outcomes. Let P_{s_0} for a finite path $\omega = \{s_0, \ldots, s_n\}$ be equal to 1 if ω has length one and otherwise $\mathrm{P}_{s_0}(\omega) = P(s_0, s_1) \cdot \ldots \cdot P(s_{n-1}, s_n)$. Now, define $\mathrm{Pr}_{s_0} : \Sigma_\Omega \to [0, 1]$ that maps from the set of events to a probability between zero and one. Let $\mathrm{Pr}_{s_0}(C(\omega)) = \mathrm{P}_{s_0}(\omega)$ for all finite paths ω. Figure 2 shows an example of a probability

space calculation given in Professor Kwiatkowska's lecture which uses the same example DTMC presented in Figure 1. For related information on probabilistic model checking, refer to [1].

4 To Be Continued ...

Part 2 of the notes for this lecture will cover *Probabilistic Computational Tree Logic (PCTL)*, the probabilistic model checking tool *PRISM*, and applications of Professor Kwiatkowska's reasearch, such as verification and synthesis of pace maker controllers.

References

[1] Andrea Bianco and Luca De Alfaro. Model checking of probabilistic and nondeterministic systems. In *Foundations of Software Technology and Theoretical Computer Science*, pages 499–513. Springer, 1995.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: Marta Kwiatkowska*

Lecture 3/2: Probabilistic Model Checking

Scribe: Ben Caulfield *Editor: Liang Gong*

This document summarizes the latter half of the guest lecture by Professor Kwiatkowska [1]. We will describe in detail the concept of *probabilistic computational tree logic (PCTL)*, probabilistic model checking on discrete-time Markov chain, as well as the advanced topics that were covered in the lecture.

1 Probabilistic Computational Tree Logic

Probabilistic Computational Tree Logic (PCTL) was first introduced by Hansson and Jonsson [2] as a way of representing *soft deadlines*, which guarantee only that certain events are likely to happen. It is an extension of Computational Tree Logic (CTL), which we have seen in class. PCTL can be used to represent complicated systems where failure is acceptable within certain probability, such as computer networks and telephone switching network [2].

One key change is that PCTL removes the existential quantifier (i.e., $\exists$) in traditional deterministic temporal logic, and add a probabilistic quantifier P. As an concrete example for illustration, the following PCTL specifies "With probability greater than 0.9, p will occur within 5 time-steps."

$$P_{\geq 0.9}\ [true\ U^{\leq 5}\ p]$$

1.1 PCTL Syntax

Similar to standard CTL, PCTL statements contain both path and state formulas.

State formulas describe properties of individual states in the corresponding model. They are denoted by propositional formulas and the probability that those formulas are correct:

$$\phi := true \mid a \mid \phi \wedge \phi \mid \neg\phi \mid P_{\sim p}[\psi]$$

To specify that for some paths some temporal logic hold, P operator always takes a path formula (ψ) as a parameter. $\sim$ means any comparison operator and $p \in [0, 1]$ is a probability.

Path formulas represent paths through the computational tree:

$$\psi := X\phi \mid \phi\ U^{\leq k}\ \phi \mid \phi\ U\ \phi$$

Here, X is the next operator, k is any natural number, U is the unbounded until operator, and $U^{\leq k}$ is the bounded until operator.

Keep in mind that a PCTL formula always gives a specification of states instead of paths (path formulas only occur inside the P operator).

1.2 More Syntax for PCTL

We can use the initial operators from the definition of PCTL to define more operators. We will start with the weak until operator, $\mathcal{U}$, which is defined by the conversion:

$$P_{\geq p}\ [\phi_1\ \mathcal{U}^{\leq k}\ \phi_2]\ \equiv\ \neg P_{<1-p}[\neg\phi_2\ U\ \neg(\phi_1 \vee \phi_2)]$$

Intuitively, $\phi_1\ \mathcal{U}^{\leq k}\ \phi_2$ means that $\phi_1\ U^{\leq k}\ \phi_2$ will hold or ϕ_1 will be true for the next k steps. We can now build the operators from Computational Tree Logic (CTL) [2]:

$$AG\phi \equiv P_{\geq 1}\ [\phi\ \mathcal{U}^{\leq \infty}\ \neg true]$$

$$AF\phi \equiv P_{\geq 1}\ [true\ U^{\leq \infty}\ \phi]$$

$$EG\phi \equiv P_{>0}\ [\phi\ \mathcal{U}^{\leq \infty}\ \neg true]$$

$$EF\phi \equiv P_{>0}\ [true\ U^{\leq \infty}\ \phi]$$

We can also express bounds on the familiar operators from Linear temporal logic(LTL): To say "ϕ will be true within k steps" we can write:

$$F^{\leq k}\phi\ \equiv\ true\ U^{\leq k}\ \phi$$

To say "ϕ must be true for the next k steps" we can write:

$$G^{\leq k}\phi\ \equiv\ \neg\left(F^{\leq k}\neg\phi\right)$$

2 Semantics of PCTL

The semantic truth of a PCTL statement is evaluated with respect to a discrete-time Markov chain (DTMC).

The (non-probabilistic) semantic rules for the state formulas are very similar to those of propositional logic:

$$s \models a \equiv a \in L(s)$$

$$s \models \phi_1 \wedge \phi_2 \equiv s \models \phi_1 \text{ and } s \models \phi_2$$

$$s \models \neg a \equiv s \models a \text{ is false}$$

Above, $s \models \phi$ means formula ϕ is satisfied in state s or ϕ is true in state s. L is the function from states in the DTMC to propositional variables so that $a \in L(s)$ if the variable a is true in the state s.

Similar to CTL, the path formulas for PCTL are evaluated over paths in the given model. Formally, $s \models P \sim_p [\phi]$ means:

$$Pr_s \{\omega \ \in \ Paths\,(s) \ | \, \omega \models \phi\} \ \sim p$$

where $\sim \ \in \{<, >, \leq, \geq\}$.

More intuitively speaking: "the probability, from state s, that ϕ is true for an outgoing path satisfies $\sim p$".

Suppose we are given an infinite path of states in a DTMC, denoted $\omega = s_0, s_1, s_2, ...$, then we have the following equivalent relations:

$$\omega \models X\phi \ \equiv \ s_1 \models \phi$$

$$\omega \models \phi_1 \ U \ \phi_2 \ \equiv \ \exists i, s_i \models \phi_2 \text{ and } \forall j < i, s_j \models \phi_1$$

$$\omega \models \phi_1 \ U^{\geq k} \ \phi_2 \ \equiv \ \exists i \leq k, s_i \models \phi_2 \text{ and } \forall j < i, s_j \models \phi_1$$

3 Examples

To see how DTMC can satisfy PCTL statements, consider the markov chain shown in the figure below, where a_1 and a_2 are propositional variables and s_0 through s_3 are state names [2]. A propositional variable is listed inside a state if it is true at that state.

We will evaluate different PCTL statements on this model.

$$s_0 \models a_1 \wedge a_2$$

This is true, since a_1 and a_2 are listed within state s_0

Define $\omega_1 =$ "$s_0, s_1, s_2, s_0, s_1, s_2, ...$", where the path loops over s_0, s_1, s_2. Consider the following statement:

$$\omega_1 \models Ga_1$$

The statement is true, as a_1 is satisfied in all three states visited in ω_1.

$$\omega_1 \models X(a_1 \ U^{\leq 3} \ a_2)$$

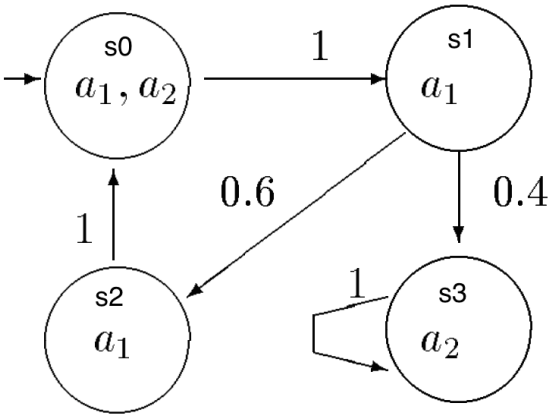


Figure 1: A simple markov chain.

This statement is also true, since in the next step (on state s_1), we will be on the path "$s_1, s_2, s_0, s_1, s_2, ..$", where "$s_1$ and s_2 satisfy "a_1 until we reach s_0, where a_2 is satisfied. This occurs in less than 3 steps, so the bound on the until operator is satisfied.

$$s_0 \models P_{<0.5}\ [XXa_2]$$

Again the above statement is true, since in the next state we are guaranteed to be at state s_1, after which there is a 0.4 chance of reaching s_3, where a_2 is satisfied.

4 Model-Checking PCTL

The input to a PCTL model checking system consists of a DTMC(discrete-time Markov chain) model and a specification represented in PCTL formula (ϕ). The output of the system is the set of states: $Sat(\phi) = \{s \in S \mid s \models \phi\}$.

A DTMC model is denoted by (S, s_{init}, P, L) where S is the set of all states in the model, s_{init} is the initial state, P is the matrix containing the probability of transitioning from one state to another. L is the labeling function mapping states to their properties.

Model-checking a PCTL statement works by splitting the statement into sub-statements and checking each of these statements before checking the larger statement. Let $Sat(\phi)$ return the set of states which satisfy the non-probabilistic state-formula ϕ. We can define Sat over the set of all states (denoted by S) in a DTMC by:

$$Sat(true) = S$$

$$Sat(a) = \{s \mid s \in L(a)\}$$

$$Sat(\neg\phi) = S \setminus Sat(\phi)$$

$$Sat(\phi_1 \wedge \phi_2) = Sat(\phi_1) \cap Sat(\phi_2)$$

$$Sat(\phi_1 \vee \phi_2) = Sat(\phi_1) \cup Sat(\phi_2)$$

In order to compute $P_{\sim p}[\psi]$, for each state $s \in S$, we must calculate the probability (denoted by $Prob(s, \psi)$) that from state s, ψ is true for outgoing paths. Then we gather all the states that satisfy $\sim p$.

$$Sat(P_{\sim p}[\psi]) = \{s \in S \mid Prob(s, \psi) \sim p\}$$

There are different ways to calculate $Prob(s, \psi)$ for each form of ψ.

If $\psi = X\phi$, we create a vector x of size $|S|$ such that $x_i = 1$ if $s_i \models \phi$ and $x_i = 0$, otherwise. We then create a $|S| \times |S|$ matrix, called P, so that entry a_{ij} equals the probability of moving from state s_i to s_j. The value of $Prob(s, X\phi)$ then becomes the i^{th} entry in the vector $P \cdot x$ [8].

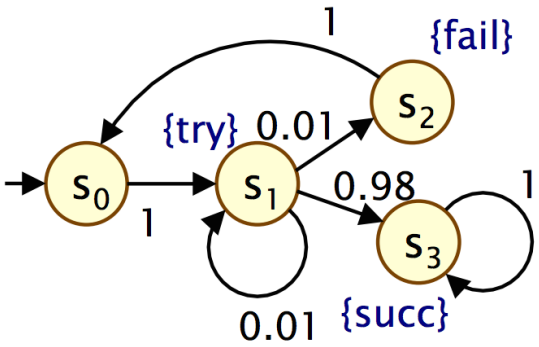


Figure 2: An example of a markov chain for calculating $Prob(s, X(\neg try \vee succ))$ [8].

For example, consider the markov chain in the figure above, and the formula $\phi = (\neg try \vee succ)$. This formula is satisfied in states s_0, s_2, and s_3. Therefore, we can calculate the vector for $Prob(s, X\phi)$ by the computation in Figure 4

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.01 & 0.01 & 0.98 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.99 \\ 1 \\ 1 \end{bmatrix}$$

Figure 3: The operation $P \cdot x$ to calculate $Prob(s, X(\neg try \vee succ)$ [8].

$$Prob\left(s, \phi_1\ U^{\leq k}\ \phi_2\right) = \begin{cases} 1 & \text{if } s \in S^{yes} \\ 0 & \text{if } s \in S^{no} \\ 0 & \text{if } s \in S^{?} \text{ and } k = 0 \\ \sum_{s' \in S} P\left(s, s'\right) \cdot Prob\left(s', \phi_1\ U^{\leq k-1}\ \phi_2\right) & \text{if } s \in S^{?} \text{ and } k > 0 \end{cases}$$

Figure 4: The recursive formula for computing $Prob(s, \phi_1 U^{\leq k} \phi_2)$. $P(s, s')$ is the probability that state s transitions to s' [8].

$$Prob\left(s, \phi_1\ U\ \phi_2\right) = \begin{cases} 1 & \text{if } s \in S^{yes} \\ 0 & \text{if } s \in S^{no} \\ \sum_{s' \in S} P\left(s, s'\right) \cdot Prob\left(s', \phi_1\ U\ \phi_2\right) & \text{otherwise} \end{cases}$$

Figure 5: The system of equations for computing $Prob(s, \phi_1 U \phi_2)$. $P(s, s')$ is the probability that state s transitions to s' [8].

In order to compute $Prob(s, \phi_1\ U^{\leq k}\ \phi_2)$, we will partition S into $S^{yes} = Sat(\phi_2)$, $S^{no} = S \setminus (Sat(\phi_1) \cap Sat(\phi_2))$, and $S^{?} = S \setminus (S^{yes} \cup S^{no})$. The value of $Prob(s, \phi_2\ U^{\leq k}\ \phi_2)$ then becomes the value of the recursive formula shown in Figure 4.

To compute the value of $Prob(s, \phi_1\ U\ \phi_2)$, we again partition S into $S^{yes} = Sat(P_{\geq 1}\ [\phi_1\ U\ \phi_2])$, $S^{no} = Sat(P_{\leq 0}\ [\phi_1\ U\ \phi_2])$, and $S^{?} = S \setminus (S^{yes} \cup S^{no})$. We then create a system of equations as denoted by Figure 4. This system can be computed by standard methods, such as Gaussian elimination and L/U factorization.

5 Quantitative Properties of Systems

Rather than simply returning a "true" or "false" when given a model and specification, it may be useful to answer quantitative questions about systems [7]. For example, consider the query "what is the probability that formula ψ is satisfied?", denoted by $P_{=?}\ [\psi]$. This particular question is fairly easy to answer, as model checkers already calculate the probability ψ occurs to compare them to specifications.

The PRISM probabilistic model checker[1] [6, 3, 5, 4] developed at Oxford can answer other quantitative questions about a model and a specification. It mainly does this through the use of a rewards system, which provides instantaneous and cumulative rewards.

Instantaneous rewards are given based on the current state. They can represent properties

[1] `http://www.prismmodelchecker.org/`

such as minimizing the size of a message queue at any time-step, and are calculate by the state reward function:

$$\rho : S \to \mathcal{R}_{\geq 0}$$

Cumulative rewards are based on of sum of rewards given over a certain period. This includes minimizing power consumption or time-steps. They are calculated using both the state reward function and transition reward function given below.

$$\iota : S \times S \to \mathcal{R}_{\geq 0}$$

6 Applications

In the past decade, probabilistic model checking has founds its application in a wide range of areas. For instance, the PRISM model checker [6] has been applied to the Bluetooth device discovery protocol to identify the worst-case expected time to hear a message. It has also been applied to analyze the demand management protocol for small grids of houses accessing electricity. PRISM also identified a flaw in the program for not giving punishments for selfish behavior.

References

[1] Marta Kwiatkowska's Home Page . `http://www.cs.ox.ac.uk/marta.kwiatkowska/`.

[2] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.

[3] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[4] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: probabilistic symbolic model checker. In T. Field, P. G. Harrison, J. T. Bradley, and U. Harder, editors, *Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002, London, UK, April 14-17, 2002, Proceedings*, volume 2324 of *Lecture Notes in Computer Science*, pages 200–204. Springer, 2002.

[5] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: A tool for probabilistic model checking. In *1st International Conference on Quantitative Evaluation of Systems (QEST 2004), 27-30 September 2004, Enschede, The Netherlands*, pages 322–323. IEEE Computer Society, 2004.

[6] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[7] M. Z. Kwiatkowska and D. Parker. Advances in probabilistic model checking. In T. Nipkow, O. Grumberg, and B. Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 126–151. IOS Press, 2012.

[8] D. Parker. PCTL Model Checking for DTMCs . `http://www.prismmodelchecker.org/lectures/pmc/`.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 3/4: Modeling for Verification, Part I

Scribe: Marten Lohstroh *Editor: Omid Bagherieh*

In this lecture we discuss some basic terminology and notation that is common in the model checking world. We will be using this notation throughout the course. We discuss different modeling languages and what makes them less or more suitable for particular classes of problems. We then settle on a class of modeling formalisms known as transition systems. Transition systems are widely used in formal methods and they are very versatile. During this course, we will center our discussions around this formalism. Today we discuss transition systems and their composition in detail. Finally, we discuss examples that are semi-realistic and feature some subtleties that come up in modeling.

1 Basic terms

So far, we have informally used the term *system*, but formally it is used in a more restricted way. In the fields of verification and synthesis we use following definitions:

Definition 1.1 *The* **system** *is the artifact being verified or synthesized.*

Definition 1.2 *The* **environment** *is everything else the system may interact with.*

Definition 1.3 *The* **model** *is a composition of the system and an environment.*

In verification, you start with a model, and this model is the combination of a system and an environment. In other words, it is not possible to state that a property holds for a given system without assuming some environment.

The first thing that we need to keep in mind when talking about formal verification is the distinction between *open* and *closed* systems. Closed systems have outputs but no inputs, whereas open systems can have both inputs and outputs. Notice that one can construct a model based on an open system, which then by composing the system with an environment effectively becomes a closed system.

The generic template for the verification problem is the following:

Definition 1.4 *Verification: Given system S, environment E, property ϕ, does $S||E \models \phi$?*

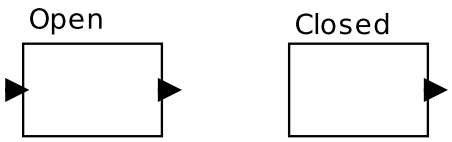


Figure 1: An open system and a closed system.

Here, $||$ is a binary operator that denotes composition, and $\models$ is a binary operator that denotes that the left-hand operand models the right-hand operand. For an expression $M \models \phi$, it is said that model M satisfies property ϕ.

The synthesis problem is phrased in identical terms:

Definition 1.5 *Synthesis: Given environment E, and property ϕ, find S s.t. $S||E \models \phi$.*

The question remains how S, E, and ϕ are to be implemented.

2 Modeling Languages

To answer this question, there is a whole lot to consider. What makes the choice of a modeling language good or bad? Let's say you are given a practical verification problem where the goal is to prove a set of assertions about a large C program. There are many tools available, but which one(s) do you pick? We list a number of things to keep in mind when making this decision:

- What characterizes the system? Is it sequential? Concurrent? Hybrid? Discrete? Probabilistic? For instance, if there is no use of probability in the system, then we wouldn't think of using a tool like PRISM[1], because we're not looking at a probabilistic verification problem.
- What is the kind of property to be verified? For instance, consider a real-time system. The implementation of such a system involves setting a timer, waiting for an amount of time and then taking some actions. But does the property mention time at all? If not, then do we need to model the real-time aspects at all? Probably not.
- What is the type of environment? In the domain of cyber-physical systems, consider the software in a device that interacts with its physical environment; a pace maker, for instance. Do we need a heart model? What about the battery?

[1] http://www.prismmodelchecker.org/

- What is the level of abstraction? A lot of the verification tools today just take the code as-is. However, especially for large programs, it may turn out that the model checker runs for a long time without yielding results; we need to provide an abstraction to make the verification process tractable.

- What about modularity? Do we model the whole system monolithically, or can we model parts and compose them? Importantly, will the composition preserve validity of properties proven for its constituents; are components *compositional*? What form of composition do we use? Not all tools support all forms.

- What is the computational engine underlying the tool at hand (e.g., SAT, BDDs, SMT), and does that engine match the problem well? Formulating the problem a certain way can improve performance when tailored to the underlying computational engine.

- Can we interact with the system directly, or do we have to translate it into something else to make it accessible to the model checker? For instance, if we want to use SPIN[2] on a C program, then we need to translate the program into a PROMELA model[3]. Then what is the fidelity of that model? Is it sound? Is it complete?

- How do we cope with uncontrolled sources of non-determinism? Let's assume we want to verify a property on the TCP network protocol. Even if we know the state machines running on every node, how do we model the communication? We need to model the transport of packets, but can they arrive out of order? Can packets get lost? Is there a bound on the communication delays?

- What is the desired accuracy and precision of aspects of the model? What is the discretization? What is the precision of the arithmetic? Does the model suffer from quantization errors?

- How robust is the model with regard to imprecision? Especially in the context of combining formal methods with machine learning, where models are learned from dataIn this case, it is important that the verification procedure can cope with a significant amount of imprecision.

Note that this is not an exhaustive list. Chapter 3 of the Handbook of Model Checking [2] elaborates further on the topic of system modeling and offers a guide for choosing the right modeling language.

[2] http://spinroot.com/

[3] There also exist tools that mechanically extract verification models from implementation level C code.

2.1 Transition Systems

As the name suggests, a transition system is some formal artifact that has notions of *state* and *transitions*. Without nailing it down to a particular domain over which the variables of the system take value, we can think of transition systems as a sort of meta-formalism that fits many scenarios.

2.2 A model for closed systems

We can define a closed system the following 3-tuple:

Definition 2.1 *A closed system $M = (S, S_0, \delta)$, where S is a set of states, S_0 is a set of initial states, and $\delta \subseteq S \times S$ is a transition relation.*

Note that in this simple definition, the output should be though of as an abstraction of the state. We assume that some of the state is observable and some of the state is not. The observables state are used to construct the output. Moreover, the transition relation δ need not be a function. If the transition relation is not a function i.e., more than one next states are possible given some current state, then the system is non-deterministic.

2.3 A model for open systems

For open systems, this becomes a 6-tuple:

Definition 2.2 *An open system $M = (S, S_0, \delta, I, O, \rho)$, where S is a set of states, S_0 is a set of initial states, I is a set of inputs, O is a set of outputs, $\delta \subseteq S \times I \times S$ is a transition relation, and ρ is an output function.*

The definition of ρ may vary. For Mealy machines, the output function maps from states to outputs $\rho : S \to O$, whereas a Moore machine produces outputs on transitions, hence $\rho : S \times I \to O$. These formalisms were invented in the mid 1950s by George Mealy and Edward Moore, respectively, both of them working at Bell Labs at the time.

Assume that we model a traffic light controller as a closed system. In this example, we have three states: $S = \{red, green, yellow\}$. The initial state S_0 is *red*, from *red* you go to *green*, from *green* you go to *yellow*, and from *yellow* you go to *red*. In the case of the open system model, as shown in Figure 2, the arcs between the states denote our transition relation δ. We have one input **tick** that may be *present* or *absent*, and outputs **go** and **stop** that may be *present* or *absent*. Our output function ρ is defined as follows: $red \xrightarrow{tick/go} green$, $green \xrightarrow{tick/stop} yellow$, and $yellow \xrightarrow{tick/stop} red$.

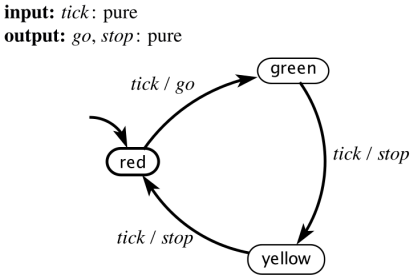


Figure 2: A simple traffic light controller.

2.4 Kripke structures

A third way to represent a transition system, and one that is often used in model checking, is a Kripke structure, which is formally defined as follows:

Definition 2.3 *$M = (S, S_0, \delta, L)$, where S is a set of states, S_0 is a set of initial states, $\delta \subseteq S \times S$ is a transition relation, and $L : S \to 2^{AP}$ is a labeling function.*

In a Kripke structure, named after the Logician Saul Kripke who proposed it in the early 1960s, the labeling function maps a state to a set to atomic propositions (AP). Intuitively, the labeling function tells you what part of the state is observable. Additionally, the transition relation δ is required to be left-total i.e., for every state there has to be at least one next state. The reason behind this is that in model checking we also wish to consider infinite executions. Note that a deadlock state can be modeled using a self-loop.

If we model the traffic light example using a Kripke structure, we can assume two propositions: *stop* and *go*. We then label the structure as follows: $red \xrightarrow{go} green$, $green \xrightarrow{stop} yellow$, and $yellow \xrightarrow{stop} red$.

In this course, we often assume our systems to be closed. More background on open systems is available in chapter 3 of Introduction to Embedded Systems [1].

3 Composition

For all the state machine formalisms discussed in the previous section, we assume an environment that triggers the system to react. Hence, timing aside, we can discuss the transition system's behavior in complete isolation. But when we discuss the composition of two transition systems, we need to be specific about the interaction mechanism between the two.

There exist many such composition mechanisms. When we arise above all nuances, we recognize two broad categories: synchronous composition and asynchronous composition.

3.1 Synchronous composition

Two synchronous systems move in lock-step.

Definition 3.1 *$M1 = (S1, S1_0, \delta 1)$ and $M2 = (S2, S2_0, \delta 2)$.*

Then $M = M1||_s M2 = (S, S_0, \delta)$, where $S = S1 \times S2$, $S_0 = S1_0 \times S2_0$, $\delta \subseteq S \times S$, and $\delta((s_1, s_2), (s_1', s_2')) = \delta_1(s_1, s_1') \wedge \delta_2(s_2, s_2')$

Note that there are cases where two composed transitions share some state. Think of a multi-threaded program on a shared memory system. In that case, we can think of the state of M1 as $S_1 = L_1 \times G$ where G denotes the global state. Similarly, we have $S_2 = L_2 \times G$. This implies that when M1 reacts, it may affect M2 and vice versa. This poses a problem, as the two machines may attempt to modify the same state at the same time. Hence, two synchronously composed state machines that share state cannot be allowed to access the same state at the same time; the behavior in that case is undefined.

3.2 Asynchronous composition

Two systems step "independently". The composition is identical to Definition 3.1, except the transition relation is different.

3.2.1 Interleaving semantics

In any given step of M, $M1$ steps or $M2$ steps, but never both together.

Definition 3.2 *$M = M1||_a M2 = (S, S_0, \delta)$, where $S = S1 \times S2$, $S_0 = S1_0 \times S2_0$, $\delta \subseteq S \times S$, and $\delta((s_1, s_2), (s_2', s_2')) = \delta_1(s_1, s_1') \wedge (s_2 = s_2') \vee \delta_2(s_2, s_2') \wedge (s_1 = s_1')$*

Using an interleaving semantics, the shared state problem discussed in Section 3.1 goes away. Hence, asynchronous composition is a good fit for multi-threaded programs.

We can express compositions of transition systems as a Kripke structure, where the labeling function is defined as follows:

Definition 3.3 *$L(s) = \{(l_1, l_2) \mid l \in L_1(s_1) \wedge l_2 \in L_2(s_2)\}$*

4 Examples

Two examples are discussed in this lecture. See chapter 3 of [1] for the first one, which is about the traffic light. The second example features an interrupt service routine (ISR) and a main loop. We can model the ISR and main loop as state machines where the states represent locations in the code. To model the composition of these two state machines, neither a synchronous, strict or non-strict interleaving semantics suffice. The model is incomplete because the scheduler, which is responsible for preempting the main loop and giving control to the ISR, is not modeled. Moreover, the model does not specify where the interrupt assertion comes from; this is an open system. The solution is to synchronously compose the ISR and the main loop with a scheduler component that decides when the ISR steps and when the main loop steps. The subsystem comprising these three components can then be composed asynchronously with an environment that asserts the interrupt. Refer to Chapter 3 of the Handbook of Model Checking [2] for more details.

References

[1] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. http://leeseshia.org, 1.5 edition, 2011.

[2] Sanjit A. Seshia, Natasha Sharygina, and Stavros Tripakis. Modeling for verification. In Edmund M. Clarke, Thomas Henzinger, and Helmut Veith, editors, *Handbook of Model Checking*, chapter 3. Springer, 2014.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 3/9: Formal Specification and Temporal Logic

Scribe: Negar Mehr *Editor: Shromona Ghosh*

Topics on modeling and how transitions of a system are modeled were covered previously. In this lecture, we discuss how properties of a system can be mathematically and formally defined. This is a very important step in the verification of a given system. We focus on one such specification language, Linear Temporal Logic in this lecture . The following topics will be covered:

1. Trace / run
2. Property, kinds of properties
3. Linear Temporal Logic (LTL)
4. Computation Tree Logic
5. Büchi-Automaton

1 Reactive System

Reactive System is one that maintains an **ongoing interaction** with its environment. It never terminates and is designed to run forever. Algorithms which require a finite or terminating run such as sorting algorithm cannot be considered for such systems.

2 Run and Trace

Given a Kripke structure, $M = (s, s_0, \delta, L)$, the following is defined for M:

a) *Run* (π) (execution) of M is a sequence of states $s_0, s_1, \ldots$ where

i) $s_0 \in S_0$

ii) $\forall i \geq 1, \delta(s_{i-1}, s_i)$

Remark: By default, a run is infinite; however, we sometimes use finite runs.

b) *Trace* of M is a sequence of labels observed during an execution of M such as $L(s_0), L(s_1), L(s_2), \ldots$.

Remark: The term"trace" is sometimes overloaded to also mean a "run". While trace refers to the observable behavior of the system, run refers to the sequence of states the system goes through in an execution.

Example: Consider the traffic signal light discussed earlier in Figure 1. An instance of a run for this system is

$$R, G, Y, R, G, Y$$

while a trace of this system is of the form

$$\{stop, go, stop, stop, go, stop\}$$

.

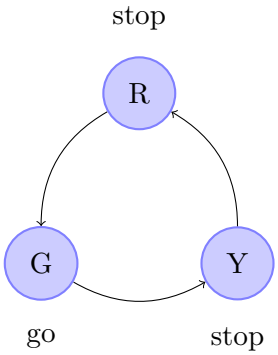


Figure 1: Run vs. Trace for a traffic signal light

3 Property

Definition 3.1 *A property ϕ is a set of runs (traces)*

- *A run π satisfies property ϕ if $\pi \in \phi$.*
- *A run π violates ϕ if $\pi \notin \phi$.*

This set-based definition of property implies that if ϕ_1 and ϕ_2 are two different properties, their corresponding sets of runs should be different.

Example: For the traffic signal light example, an instance of a property can be:

R is always followed by G.

Properties are divided into two categories: *Safety* and *Liveness*.

Definition 3.2 *ϕ is a safety property if every infinite run π that violates ϕ has a finite prefix π^j that violates ϕ for every possible extension.*

Intuitively we say, "nothing bad happens" [2]. For violation of safety properties, we have a finite length error trace.

Definition 3.3 *ϕ is a liveness property if every finite–length run satisfying ϕ can be extended to an infinite run satisfying ϕ.*

Intuitively we say, "eventually something good happens"[2]. For violation of liveness properties, we have infinite length error trace and need infinite runs of the system.

Example:

- "The grant signal must be asserted some time after the request signal is observed".

  The only way this property can be violated is having sequence of the form $req, \neg gt, \neg gt, \neg gt, \ldots$ which is infinite $\Longrightarrow$ It is a liveness property.

- "Every request must receive an acknowledgement (ϕ_1), and ($\wedge$) the request must remain asserted until it is acknowledged (ϕ_2)".
  This property is conjunction of two properties ϕ_1 and ϕ_2. Violation of ϕ_1 requires observation of infinite–length sequences while ϕ_2 can be violated by finite–length sequences; thus, ϕ_1 is a liveness property while ϕ_2 is a safety property. Such a property is neither liveness or safety. This is because a safety properties has only finite–length error traces and liveness properties have infinite length–length error traces. This system can have both, so we cannot say that it is a safety or liveness property.

4 Temporal logic

Temporal logic is the logic used for specifying properties of systems related to behavior of the system over time. There exist various types of temporal logics such as:

$$\underbrace{\text{LTL}, \text{CTL}, \text{CTL}^*}_{\text{propositional temporal logic}} \quad , \quad \underbrace{\text{MTL}, \text{TCTL}}_{\text{real time extension}} \, , \text{STL}, \text{PCTL}, \text{PLTL}, \ldots$$

where propositional temporal logics are extensions of propositional logics $\Longrightarrow$ LTL, CTL and CTL* include all standard logical propositions.

4.1 Linear Tempral Logic

Linear temporal logic can express behavior of the system over a single time–line. It employs temporal operators to capture such behavior of the system:

Temporal Operators:

- "globally", "always" denoted by $\Box$ or **G**:
- "eventually", "finally" denoted by $\Diamond$ or **F**:
- "in the next state" denoted by $\bigcirc$ or **X**
- "until" denoted by **U**

LTL Grammar: Any propositional logic formulation of the form:

$$\phi := p \in \text{AP} \mid \neg \phi \mid \phi \wedge \phi$$

or we can nest temporal operators:

$$\mathbf{G}\phi \mid \mathbf{F}\phi \mid \mathbf{X}\phi \mid \mathbf{U}\phi$$

Example:

1. "p is always true" $\Longrightarrow \mathbf{G}p$
2. "p is always true and q is always false" $\Longrightarrow (\mathbf{G}p \ \wedge \ \mathbf{G}\neg q)$ or $\mathbf{G}(p \wedge -q)$.

A run is denoted by π and it refers to the set of states:

$$\pi = s_0, s_1, s_2, s_3, \ldots$$

A run π satisfying a property ϕ is denoted by:

$$\pi \models \phi$$

The following is easy to verify:

- $\pi \models \neg\, \phi \quad \text{iff} \quad \pi \not\models \phi$
- $\pi \models \phi_1 \wedge \phi_2 \quad \text{iff} \quad \pi \models \phi_1 \ \wedge \pi \models \phi_2$
- $\pi \models \mathbf{G}p \quad \text{iff} \quad \forall i \geq 0,\, p(s_i)$ (Figure 2)
- $\pi \models \mathbf{G}\phi \quad \text{iff} \quad \forall i \geq 0, \pi_i \models \phi$, where π_j is suffix of π starting at state j:

$$\pi_j \triangleq s_j, s_{j+1}, s_{j+2}, \ldots$$

Figure 2: $\mathbf{G}\phi$ where ϕ is p

- $\pi \models \mathbf{F}\phi \quad \text{iff} \quad \exists i \geq 0\,, \pi_i \models \phi$
- $\pi \models \mathbf{X}\phi \quad \text{iff} \quad \pi_1 \models \phi$
- $\pi \models \phi_1 \mathbf{U} \phi_2 \quad \text{iff} \quad \exists j \geq 0 \quad , \quad \pi_j \models \phi_2 \ \wedge\ 0 \leq i < j,\ \pi_i \models \phi_1$ (Figure 3)

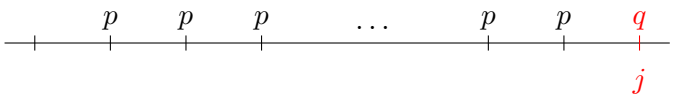


Figure 3: $\pi \models \phi_1 \mathbf{U} \phi_2$ where ϕ_1 is p and ϕ_2 is q

Question: Is $\mathbf{G}p \wedge \mathbf{G}\neg q \equiv \mathbf{G}(p \wedge \neg q)$?

This is a question of LTL satisfiability (validity) that can be given to an LTL satisfiability solver to validate their equivalence. Since this is a trivial LTL satisfiability problem, we can easily demonstrate it is true:
Proof of $\Longrightarrow$ *side:*

Since any specification is a trace we should have

$$\forall \pi \ni \pi \models \mathbf{G}p \wedge \mathbf{G}\neg q \quad \Longrightarrow \quad \pi \models \mathbf{G}p \wedge \pi \models \mathbf{G}q$$

$\pi \models \mathbf{G}p \wedge \mathbf{G}\neg q \quad \Longrightarrow$

$\forall i \geq 0\ p(s_i) \quad \wedge \quad \forall j \geq 0\ \neg q(s_j)$

$\forall i \geq 0\ p(s_i) \quad \wedge \quad \neg q(s_i)$

$\forall i \geq 0,\ (p \wedge \neg q)(s_i) \implies \pi \models \mathbf{G}(p \wedge \neg q)$

Proof of the $\Longleftarrow$ side follows the same reasoning.

Note: Temporal operators can be nested as well.

Example:

1. **GF**p : It is always the case that sometime in future p is true (Figure 4).
   $\forall i \geq 0\ \pi_i \models \mathbf{F}p$
   $\forall i \geq 0\ \exists j \geq i\ \ni\ p(s_j)$

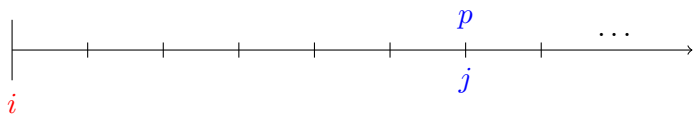


Figure 4: **GF**p

   GFp is interpreted as "infinitely often p" which is a widely–used property characterizing the notion of fairness; that is, p is true infinitely often in a run.

2. **FG**p : Sometime in future p holds always (Figure 5).

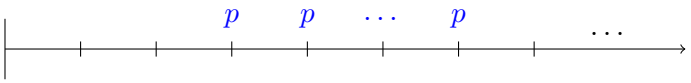


Figure 5: **FG**p :

   FGp is used for characterizing steady state stability properties.

4.1.1 Expressing LTL operators using Until

Most of LTL operators can be expressed in terms of *Until* using $\neg$, $\wedge$ and $\vee$. For instance, every **G** can be written in terms of **F** and every **F** can be expressed using **U**:

$$\mathbf{G}p = \neg\mathbf{F}\neg p$$

$$\mathbf{F}p = true\ \mathbf{U}p$$

It should be noted that **X** cannot be written in terms of **U**.

Note: Let's assume for a Kripke Structure M, a requirement is given by an LTL specification ϕ. The question is when does M satisfy ϕ?
M satisfies ϕ when *all* runs of M satisfy ϕ. LTL cannot encode properties of the form *at least one* run of M must satisfy ϕ. LTL cannot represent quantification over executions(paths). *Computation Tree Logic(CTL)* is the logic capable of encoding requirements using quantifiers.

4.2 Computation Tree Logic(CTL)

CTL is a branching-time logic used to visualize computations of a non-deterministic system where every path through the tree is a computation. CTL formulae can be divided into two types:

State Formulae ψ: $s \models \psi$ where s is the state.
Path Formulae ϕ: $\pi \models \phi$ where π is the path.

State formulae contain path quantification operators which are defined as:

- **A** : For all paths (from the state s)
- **E** : There exists a path (from the state s)

In order to illustrate how CTL specifications are interpreted, let's consider the two following examples:

$s_0 \models \mathbf{AG}(\mathbf{EF}\, reset) \equiv$ For all path starting at s_0, it is always the case that for every state in the path, there exists one path along which in future we get back to reset (Figure 6).

$s_0 \models (\mathbf{EF}\, reset) \equiv$ is used to imply that there exists one path which goes to reset.

Note: In CTL formulation every **A** or **E** must be followed by exactly *one* temporal operator **G**, **F**, **U** or **X**. For instance, we cannot write a CTL specification as $\mathbf{A}(\mathbf{FG}\phi)$. For the cases where arbitrary nesting of operators is necessary to express the required property, CTL^* can be incorporated.

4.3 CTL*

CTL^* is the logic where **A** or **E** does not need to be associated with exactly one LTL temporal operator and any arbitrary nesting of operators is allowed. An example of a

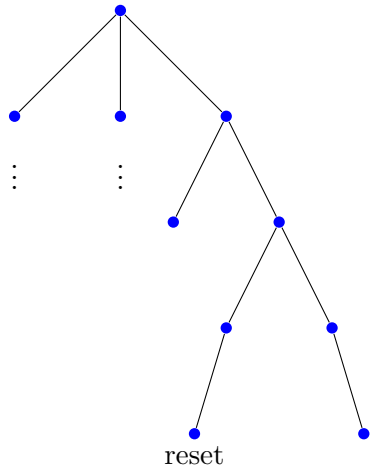


Figure 6: CTL Representation of a system with one of the states defined as *reset*

CTL^* formula is $\mathbf{A}\phi$, $\phi \in LTL$. In fact, as figure7 shows LTL and CTL are both subsets of CTL^*. Intersection of LTL and CTL is caused by the specifications which can be expressed equivalently in CTL and LTL.

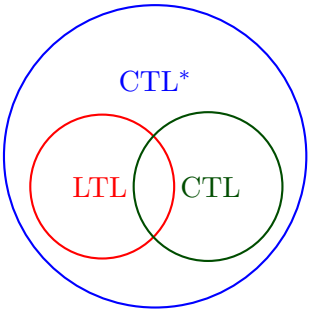


Figure 7: Venn Diagram of LTL, CTL and CTL*

4.3.1 LTL vs. CTL

A CTL formula ϕ_{CTL} and an LTL formula ϕ_{LTL} are equivalent if they are satisfied by the same Kripke structures [1].

$$\phi_{CTL} \equiv \phi_{LTL} \Longleftrightarrow [M \models \phi_{CTL} \Leftrightarrow M \models \phi_{LTL}]$$

Not all CTL specifications can be expressed by LTL. An example of such class of CTL specifications is the ones necessitating **E** which implies the effectiveness of CTL. Although CTL

can be really expressive, it is found that only simple CTL equations are comprehensible, nontrivial equations are hard to understand; thus, prone to errors [3]. As a result LTL is more widely used.
However, since CTL model checking can sometimes be faster than LTL, it still can be used as a way to approximate LTL. In the cases where CTL specification is weaker than its LTL counterpart, CTL version can be used for finding bugs. When CTL specification is stronger, it is used for verifying correctness:

Example:

- **AGEF** p is weaker than **GF** p (Figure 8), useful for finding bugs.

Figure 8: LTL being stronger than CTL

- **AFAG** p is stronger than **FG** p (Figure 9), useful for verifying correctness.

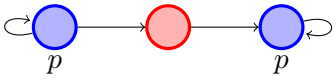


Figure 9: LTL being weaker than CTL

Despite the fact that a large number of specifications can be expressed using LTL or CTL, there are still some class of properties that cannot be encoded using syntax of CTL or LTL. As an example, specifications such as absence of deadlock which is an oft-cited property cannot be described using LTL or CTL ideas. For such class of specifications *Büchi Automata* can represent the desired property.

4.4 Büchi Automaton

A deterministic finite automaton is a tuple $DFA = (s, \Sigma, \delta, s_0, A)$ where:

s: Finite set of states

Σ: Finite set of input alphabets

δ: Transition relation

s_0 : Finite Set of initial states

A: Finite set of accepting states

The set of all the words accepted by an automaton is called the language of that automaton. For instance, for the automaton shown in Figure 10, where $\Sigma = \{0, 1\}$, its language is all finite-length binary strings with an odd number of 1s (Accepting states are the ones denoted by double circles). The way we define DFA's is that we run it over finite number of traces. Consideration of infinite-length traces leads to Automata on Infinite Words.

The most intuitive type of automaton is *Büchi Automaton* which extends a finite au-

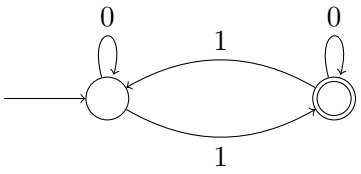


Figure 10: Deterministic Finite Automaton (also interpretable as a Büchi Automaton)

tomaton to infinite inputs. It accepts an infinite input sequence if there exists a run of the automaton that visits (at least) one of the accepting states infinitely often. In other words, B accepts a run, $\omega \in \Sigma^\omega$, if it visits states in F infinitely often where F is the set of all acceptable states.

Büchi Automaton = $(S, \Sigma, \delta, I, F)$

s: Finite set of states

Σ: Finite set of input alphabets

δ: Transition relation

s_0 : Finite Set of initial states

F: Finite set of accepting states

The automaton shown in Figure 10 is an example of a Büchi Automaton where its language if interpreted over infinite words, is all infinite-length binary strings with an odd parity of 1s or infinitely many 1s.

4.4.1 Deterministic vs. Non-Deterministic

In deterministic finite automata, transition relation is in fact a transition function i.e from a given state and given input, we can determine the behavior of the system. For non-deterministic automata, transition function is replaced by a transition relation, there exists more than one possible transition corresponding to a given input for at least one state of the system. DFAs(deterministic finite automata) are equivalent to NFAs(non-deterministic finite automata) in terms of the language of words they express; however, deterministic Buchi Automata are strictly less expressive than non-deterministic Buchi automata.

4.4.2 LTL and Büchi Automata

An LTL specification ϕ is expressed by a Büchi Automaton A_ϕ if all traces(runs) satisfying ϕ are accepted by A_ϕ. Following examples illustrate how a Büchi Automaton can be generated from an LTL specification:

Example:

- Automaton for $\mathbf{G}p$, (Figure 11)

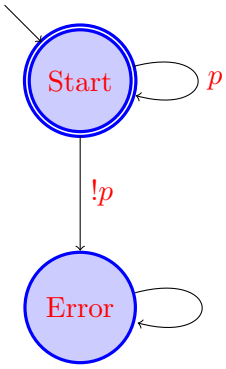


Figure 11: Automaton for $\mathbf{G}p$

- Automaton for $\mathbf{F}p$, (Figure 12)

- Automaton for $\mathbf{GF}p$, (Figure 13)

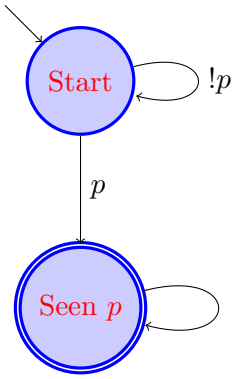


Figure 12: Automaton for $\mathbf{F}p$

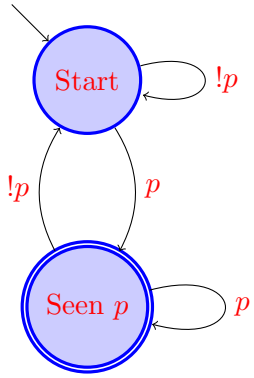


Figure 13: Automaton for $\mathbf{GF}p$

Although LTL specifications can be expressed as Büchi Automata, the converse is not true. All Büchi Automata may not represent an LTL formula. Consider the automaton in Figure 14.

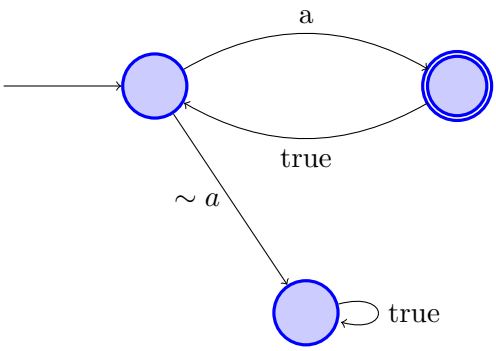


Figure 14: Automaton without LTL counterpart

The language it accepts is of the form

$$a\ a\ a\ a \cdots$$

$$a\ \neg a\ \ a\ \ \neg a \cdots$$

which cannot be expressed using LTL. One may recommend $a \wedge \mathbf{G}(a \implies \mathbf{XX}a)$ as the LTL representation but even this does not work since it allows languages like

$$a\ a\ a\ \neg a\ \ a \cdots$$

which are not accepted by the automaton.
In spite of the fact that Büchi Automata are more expressive than LTL formulae, it is not clear whether writing LTL formulas is easier than automata. For instance, when the automaton gets too big LTL formulation is preferred. There are also certain languages which accept only LTL formalism.
Universal Automata: Universal Automata are the ones which accept a word if all resulting paths of the given word visit final states infinitely often.

References

[1] Robert Bellarmine Krug. CTL vs. LTL. Lecture Slides, May 2010. `http://www.cs.utexas.edu/users/moore/acl2/seminar/2010.05-19-krug/slides.pdf`.

[2] Ekkart Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.

[3] Thomas Schlipf, Thomas Buechner, Rolf Fritz, Markus Helms, and Juergen Koehl. Formal verification made easy. *IBM Journal of Research and Development*, 41(4.5):567–576, 1997.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 3/11: Explicit-state Model Checking

Scribe: Ankush Desai *Editor: Sanjit A. Seshia*

Emerson [1] describes the history of model checking and how model checking was *discovered* while solving the problem of synthesizing a correct model from its temporal specification. In this lecture, we will introduce the subject of *explicit-state model checking*. As the name suggests, in *explicit-state model checking* we *explicitly* enumerate states of a system and check whether some property holds on the resulting state graph. Much of this lecture was covered using slides, and so the notes below are written to augment the posted slides.

1 Explicit State Model Checking

Model Checking is an automatic approach of verifying correctness of a system. Its a method to establish that a given system meets its specification. In other words, If the system is defined as a finite state-transition M and the specification ϕ to be checked is represented using temporal logic (recall, LTL, CTL from last lecture) then the model checking problem is to algorithmically check if $M \models \phi$.

If the system is represented using Kripke structure $M : (S, I, R, L)$, then M can be considered as a graph where nodes in the graph are the explicit states (S) of the system and edges are the elements in R using which the system transitions from one state to another. Model-checking can be considered as a graph algorithm that efficiently traverses the graph and for each visited state asserts that the temporal property is satisfied.

Consider the simple case of checking $M \models Gp$. The model checker enumerate all the states of M from some initial state which yields the state graph G_M and for each node in graph G_M check the state property p.

In the case of *explicit-state* model checking, the state of the system is *explicitly* represented and all the paths or states in the systems are *explicitly* enumerated by traversing all state-transitions. SPIN [2] was the the first explicit-state model checker applied to industrial scale protocols and it was able to find many design bugs. Explicit-state model checking suffers from the problem of state-space explosion. Another approach to model checking is using *symbolic* representation which will be covered in the next lecture where states are represented *symbolically*. The problem of state space explosion is mitigated in symbolic model checking by representing states compactly using BDDs.

The graph search algorithm in explicit-state model checker can be implemented using either

depth-first search (DFS) or *breadth-first* search (BFS). Both the approaches have their own advantages, BFS algorithm always returns the shortest counter-example but it suffers from the problem of having to store all the states at the next depth in a queue which can grow exponentially with the depth of the graphs. DFS does not face this problem as it stores only the states that are in the depth-first search stack. The problem is explained in more details in the context of iterative search in [3].

Suppose we use depth-first search for exploration, then to avoid the re-exploration of previously visited states, we should store the set of visited states. In most of the model checkers to save memory, compressed representation (canonical hashing) of the state is stored for each visited state instead of storing the entire concrete state. Storing the set of all the visited states could be expensive, the stateless model checkers store only the states in its search-stack. The termination of search in that case can be detected by using only the states in search-stack, but at the expense of exponential path enumeration.

Steps for checking if M satisfies ϕ using a model checker:

1. Compute the Büchi automaton B corresponding to $\neg\phi$.
2. Compute the Büchi automaton A corresponding to the system M.
3. Compute the *synchronous* product P of A and B.
    - Product computation defines "accepting" states of P based on those of B.
4. Check if some "accepting" state of P is visited infinitely often.
    - If so, we've found a bug
    - If not, no bug in M with respect to ϕ

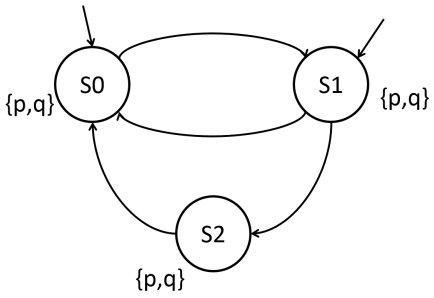


Figure 1: Kripke Structure

Consider a simple kripke structure presented in Figure 1. The corresponding Büchi automaton is as shown in Figure 2. Note that both the kripke structure and the Büchi

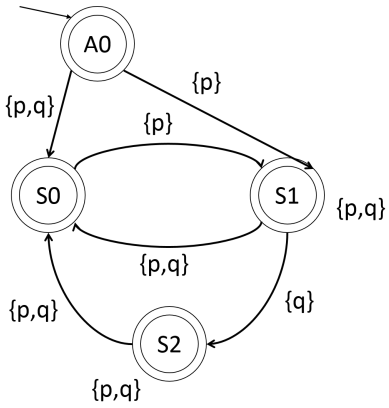


Figure 2: Büchi Automaton

automaton have the same state-labels, alphabets and transitions. The difference between the two structures is that the Büchi automaton has labels on transitions and all the states are accepting states. The automaton has hidden transition to an error state on alphabets for which transitions are not explicitly defined. To explain this in more details consider the steps for constructing Büchi automaton from the Kripke structure:

1. Given kripke structure $M = (S, S_0, R, L)$. $L : S \rightarrow 2^{AP}$, AP is the set of atomic propositions.

2. Contruct Büchi automaton.

   $A = (\Sigma, S + \{\alpha_0, err\}, \Delta, \{\alpha_0\}, S + \{\alpha_0\})$ where :

   - Alphabet, $\Sigma = 2^{AP}$
   - Set of states $= S + \{\alpha_0, err\}$, α_0 is a special start state and err is a error state.
   - All states are accepting states except err
   - Δ is transition relation of A such that:
     $\Delta(s, \sigma, s')$ iff $R(s, s')$ and $\sigma = L(s')$
     $\Delta(\alpha_0, \sigma, s)$ iff $s \in S_0$ and $sigma = L(s)$

As pointed out earlier, the Büchi automata A and B are composed using synchronous product. In such setting, the problem of verifying $M \models \phi$ is reduced to the reachability problem of checking if error state err is reachable in the product automaton of A and B. The counter example when ϕ is a safety property is an finite length trace from initial state to the err state that violates the property ϕ. If ϕ is a liveness property then the counter example is a lasso, where the cycle in the lasso represents the infinite length trace.

References

[1] E.Allen Emerson. The beginning of model checking: A personal perspective. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 27–45. Springer Berlin Heidelberg, 2008.

[2] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[3] Abhishek Udupa, Ankush Desai, and Sriram Rajamani. Depth bounded explicit-state model checking. In Alex Groce and Madanlal Musuvathi, editors, *Model Checking Software*, volume 6823 of *Lecture Notes in Computer Science*, pages 57–74. Springer Berlin Heidelberg, 2011.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: Daniel Bundala*

Lecture 3/18: Symbolic Model Checking

Scribe: Ben Mehne *Editor: Marten Lohstroh*

In this lecture we discuss how to check models over state spaces that are too large to handle via explicit model checking. The lecture briefly reviews explicit model checking before transitioning to symbolic model checking, and discusses fixed points[1]—a means of determining stopping criteria for symbolic model checking algorithms. All of the material of this lecture is covered in more depth in [1].

1 Review: Explicit Model Checking

Explicit model checking is the process of enumerating possible models and checking if they satisfy some property. For example, consider:

Example: Kripke Structure

Let the Kripke Structure M be as in Figure 1. A classical problem would be checking if

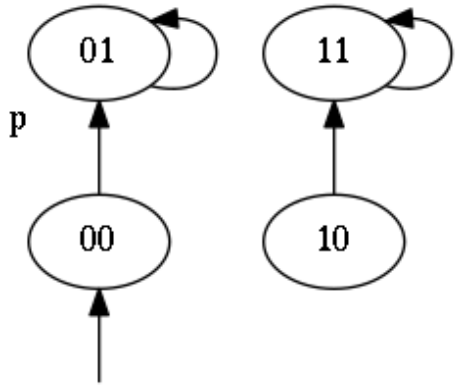


Figure 1: A simple Kripke structure. Note that state 01 has label p.

you can reach p from the initial state. In this case, you can do a graph traversal and decide whether that state is reachable. Generally, simple graph traversals are not practical for evaluating whether a large Kripke structure satisfies a property.

[1] A fixed point is sometimes shortened to *fixpoint*, or called an *invariant point*.

Example: Source Code

Consider a Kripke structure derived from the following C source code:

```
1 void f(int x){
2         int y = x & 0x123456;
3         assert(y!=0);
4 }
```

The question here could be whether the assert on line 3 is ever violated. In other words, φ : can we reach a state where $y = 0$ at line 3?

It is possible to encode this problem in a Kripke structure M that remembers every possible value of x and every (resulting) possible value of y, which would result in a graph like in Example 1. For instance, the state transitions in M would be something like:
$\{x = 0, y = 1\} \to \{x = 0, y = 0\}$,
$\{x = 1, y = 0\} \to \{x = 1, y = 0\}$,
...

This leads to a very large state space. If x and y are 32-bit integers, $|M| = 2^{64}$. This size is untenably large and intractable to traverse.

Today we discuss how to prove properties in these cases: symbolic model checking lets us reason about Kripke structures that are too large to operate on directly.

2 Symbolic Model Checking

Symbolic Model Checking is a method to handle Kripke structures with large state spaces.

There is a classical approach on how to solve cases with large state spaces:

2.1 Characteristic Functions

Consider a finite universe (a set) with n elements.

1. Represent each element using $\lceil \log n \rceil$ bits
2. For each subset S of the universe, consider it as a function $f_S(x) = 1 \iff x \in S$, where x is a $\lceil \log n \rceil$ bit vector
    - f_S is the characteristic function of S; it returns 1 (or "true") when the value x represents is in S.

3. Instead of enumerating states in S, store (and operate on) f_S

   - Recall from the Feb. 4th lecture, that BDD's can efficiently represent any boolean function. f_S can be formulated as boolean function on the $\lceil \log n \rceil$ input bits that evaluates to whether those input bits represent an element in the set S.

In Example 1, the elements $\{00, 01, 10, 11\}$ would be represented using 2 bits since there are 4 elements. The subset of those states labeled p would be represented by a function

$$P(x) = \begin{cases} 1 & : \text{ if } x = 01 \\ 0 & : \text{ otherwise} \end{cases}$$

2.1.1 Set operations

We can represent typical set operations on two sets S and T via operations on their characteristic functions f_S and f_T.

- $S \cup T \mathrel{\hat{=}} f_S \vee f_T$
- $S \cap T \mathrel{\hat{=}} f_S \wedge f_T$
- $S = \emptyset \mathrel{\hat{=}} f_S \equiv 0$

One way we can use this structure is as follows: if we wanted to determine if there is any value in set S that starts with a 0, we could detect if $\exists_x \ . \ f_S(0 \parallel x) = 1$.

Kripke structures have a transition function δ. This can be represented via a characteristic function like

$$T(x, x') = \begin{cases} 1 & : \text{there is a transition from } x \text{ to } x' \text{ in one step} \\ 0 & : \text{ otherwise} \end{cases}$$

T can also be represented by a BDD.

The characteristic function corresponding to Example 1 is: $T(x, x') = (x'_2 = x_2) \wedge (x'_1 = 1)$ where x_1 is the first bit in x and x_2 is the second bit.
We may try to solve φ from the same example by iteratively expanding from the initial set: Let S_0 represent the initial states and $T(x, x')$ and $P(x)$ be as above.

Question: Can we reach p in two steps?
This is the same as asking (or asserting): $\exists_a \exists_b \exists_c \ . \ S_0(a) \wedge T(a, b) \wedge T(b, c) \wedge P(c)$.

To solve this problem we may build a set of states S_1 which are one step (transition) away from S_0, and then a set S_2 that is two steps away. Then we check if $S_2 \cap P = \emptyset$, in which case we conclude that p is reachable in two steps.

This algorithm can be made more generic to handle an arbitrary number of steps:

while True **do**
  $S_{i+1} \leftarrow S_i \cup \{y | \exists_x \ . \ S_i(x) \wedge T(x, y)\}$
  if $S_{i+1} \cap P \neq \emptyset$ **then**
    return True
  end if
  if $S_{i+1} = S_i$ **then**
    return False
  end if
end while

Remark: $\exists_x \ . \ Q(x, y, z) = Q(0, y, z) \vee Q(1, y, z)$. This allows us to derive the set in the S_{i+1} assignment line, after computing the BDD of $S_i(x) \wedge T(x, y)$.

Remark: Eventually, $S_{i+1} = S_i$ because there is a finite number of states; this is a fixed point.

This algorithm computes forward reachability from the initial S_0 set. The worst case for this algorithm is the depth or width of the graph in the Kripke structure.

A similar algorithm can be constructed that computes backward reachability; by starting at P and applying the transitions backwards, you can search towards the initial states.

Question: How do we handle CTL formulas, and how do we adjust the above algorithm to determine if a Kripke structure satisfies some given formula?

The above algorithm is a specific case of finding a fixed point. The general form can be used to determine if a Kripke structure satisfies arbitrary CTL formulas.

2.2 Fixed Points

Definition 2.1 *Fixed Point: For some universe Σ, let $f : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ some function that goes from a subset of Σ to a subset of Σ. Then a set $S \in \mathcal{P}(\Sigma)$ is a fixed point of f if $f(S) = S$.*

Remark: Note that fixed points can exist for any a function that maps to and from the same set. A function for which the domain and codomain is the same—a function that maps to and from the same set—is called an *endofunction*.

For example, if Σ is the set of states of a Kripke structure, and $F(Z) = Z \cup \{y | \exists_x \ . \ x \in Z \wedge T(x, y)\}$, then the set of all reachable states from S_0 is a fixed point.

In Example 1, we have $Z = \{00, 01\}$, and $f(Z) =$ the reachable states from any state in $Z = \{00, 01\} = Z$, which makes Z a fixed point. Additionally, if Z is the set of all states from the same example, it is also a fixed point. Intuitively, one can think of Z, in this case, as some partition of the state graph that does not have any outgoing paths.

2.2.1 Tarski's Theorem

In order to use fixed points to prove that Kripke structures satisfy CTL formulas, we need to know whether a fixed point exists. In particular, we are interested in finding a least or greatest fixed point. To guarantee the existence of such a fixed point, our function f needs to be monotonic.

Definition 2.2 *Monotonic Function: A function f is monotonic if, whenever you have two sets $X \subseteq Y$, then $f(X) \subseteq f(Y)$.*

Remark: Generally, monotonic (order-preserving) functions operate on partially ordered sets $(\Sigma, \leq)$, not just power sets. Note that $\subseteq$ is a total ordering on $\mathcal{P}(\Sigma)$.

Theorem 2.1 *Knaster-Tarski Theorem: If $f : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ is monotonic then:*

- *f has a least fixed point, called $\mu_Z \ . \ f(Z)$. This is a fixed point that is contained in any other fixed point. It is, trivially, $\bigcap\{Z | f(Z) \subseteq Z\}$.*
- *f has a greatest fixed point, called $\nu_Z \ . \ f(Z)$. This is a fixed point that is contains all other fixed points. It is, trivially, $\bigcup\{Z | f(Z) \supseteq Z\}$.*

Remark: The Knaster-Tarski theorem states that the set of fixed points of f in $\mathcal{P}(\Sigma)$ (a complete lattice), is also a complete lattice. Because a complete lattice cannot be empty (all its subsets must have a supremum and an infimum), this theorem implies there is indeed a least and greatest fixed point). The Kleene fix-point theorem provides a constructive procedure to find these fixed points.

We can calculate the least and greatest fixed points as follows:

- $\mu_Z \ . \ f(Z) = \emptyset \cup f(\emptyset) \cup f(f(\emptyset)) \ldots = \bigcup_{i=0}^{\infty} f^i(\emptyset)$
- $\nu_Z \ . \ f(Z) = \bigcap_{i=0}^{\infty} f^i(\Sigma)$ (where Σ is the entire set)

2.3 CTL via Fixed Points

We can express CTL formulas as fixed points. For example, we can express AGp (Allways Globally p) in terms of a fixed point:

A state s satisfies AGp if it satisfies both of these properties:

1. s (trivially) satisfies p
2. every successor of s satisfies AGp

These two properties, combined, correspond to the greatest fixed point formula $\nu_Z \ . \ (p \wedge \{s | \text{all successors of } s \in Z\}) = \nu_Z \ . \ (p \wedge AXZ)$ (recall that X is the CTL representation for the $neXt$ state by transition).

In Example 1, only state 10 satisfies AGp.

This μ calculus allows us to express other CTL formulas:

- $EFp = \mu_Z \ . \ (p \vee EXZ)$—there Exists a path that Finally is p
- $E(pUq) = \mu_Z \ . \ q \vee (p \wedge EX(pUq))$—there Exists a path that is p Until it is q
- $EGp = \nu_Z \ . \ (p \wedge EXZ)$

Remark: $pUq = q \vee (p \wedge X(pUq))$

Question: How do we know whether it is the least or greatest fixed point?

Generally we can determine this by counterexample. For example, imagine that the EFp formula was a greatest (instead of least) fixed point. Following the method to calculate the greatest fixed point formula in 2.2.1, the entire set would be the fixed point. During the first iteration, the $f(Z) = p \vee EXZ$ would be evaluated on the universe set Σ. This results in the same set: the universe set. You can see this because for every state (in the universe Σ), it either is p or it is in the universe (as all states are in the universe). By stepping through the formulas to calculate the fixed points and seeing if the result is reasonable, we can determine which of the two options is correct (since there are only two options of importance: a least or greatest fixed point).

2.3.1 Nested CTL Formulas

At this point, we know how to calculate fixed points for simple CTL formulas. To derive the fixed points for a CTL formula with CTL subformulas, we can perform the following:

1. Calculate the fixed points to the innermost subformula of the CTL formula
2. The resulting fixed point is a set of states that, intuitively, forms a predicate representing when that subformula is true
3. In the entire formula, replace the subformula by this predicate
4. If the resulting formula is a single predicate, then you have the resulting set. Otherwise, proceed to step 1

Remark: The characteristic function of the predicate, which represents the fixed point set of states that satisfies the subformula, can be represented by a BDD since it is a set of elements from the universe of states.

In general, fixed point calculation for CTL formulas can be performed via BDDs. The methods for calculating fixed points from 2.2.1 can be performed by iteratively evaluating the CTL formula on a characteristic function of the initial set (for the least fixed point, this is the empty set function which always returns 0 and for the greatest fixed point, this is universe set function which always returns 1).

For further discussion of all of the topics from this lecture, see [1] (available in the BCourses files for this class).

Next lecture we will discuss witness generation.

References

[1] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. pages 124–175. Springer-Verlag, 1993.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 3/30: Bounded Model Checking with SAT/SMT, Abstraction

Scribe: Sarah Chasins *Editor: Mike McCoyd*

This lecture covers the following topics:

- CTL: Counterexample/Witness Generation
- Bounded Model Checking for LTL
- Abstraction in Model Checking

This lecture first covers how, once one has finished the fixpoint computation for symbolic model checking, one identifies a counterexample or a witness for the target property.

Next, the lecture discusses Bounded Model Checking — symbolic model checking not with BDDs but with SAT or SMT. We discuss it in the context of LTL rather than CTL. One can use this approach with CTL, but would learn about a different class of techniques that way.

Finally, the lecture covers abstraction, perhaps the single most important technique that allows model checking to scale today.

1 Counterexample and Witness Generation

We will discuss counterexample and witness generation in the context of model checking Kripke structures.

1.1 A Brief Review of the Fixpoint Computation

Suppose that for Kripke structure $M = (S, S_0, R, L)$, we want to check EG p. We need the greatest fixpoint. The fixpoint formulation for EG p is:

$\nu Z.p \wedge EXZ$

The way to read this is that Z is the fixpoint. EG p is all states where p is true and there exists a next state where Z is true. This is a greatest fixpoint, so we traverse the lattice

of sets starting from True at the top of the lattice. True is our initial approximation. The recurrence relation mirrors the formula above.

$Z_0(v) = True$

$Z_k(v) = P(v) \wedge \exists v'.[R(v, v') \wedge Z_{k-1}(v')]$

We stop when $Z_k = Z_{k-1}$.

Each Z_i is a BDD representing a set of states, so one can perform the check for set equality by checking whether the two BDDs are the same. In the process of the fixpoint computation, we compute the sets $\{Z_0, Z_1, ...Z_k\}$. We will use these sets for the ultimate goal of model checking, which is to determine for the target property whether there exists a counterexample or a witness.

1.2 Witnesses and Counterexamples

We start by defining witnesses and counterexamples.

Witness: an execution path that demonstrates satisfaction of the property ϕ by model M.

Counterexample: an execution path that demonstrates violation of the property ϕ by model M.

1.2.1 The witness computation problem

In this subsection we address how to use the fixpoint computation's output to obtain, in the EG p case, a witness. The general approach is very similar for other properties.

Given: S_0, R, and $\{Z_0, Z_1...Z_k\}$ (the history of our fixpoint computation)

In the case of EG p, each Z_i is a set of states such that each state in that set satisfies p, and has a path of length $(i-1)$ steps (transitions) from it comprising states satisfying p. Think of Z_i as the set of states for some i between 0 and k where we know that if we have a state s in Z_i, then we know there is some path of length $i-1$ where every single state along that path satisfies p. There could be other branches, but we are only interested in the fact that there exists one.

Since Z_k is the fixpoint, we can say something more about Z_k. We can say that from any state in Z_k, there exists an infinite path along which all states satisfy p. So we will see a looping path. We do not explicitly seek a loop, but the following procedure for identifying successors will reveal it.

Approach to computing witness:

- Check if $S_0 \subseteq Z_k$. If not, the property does not hold. If yes, select $s_0 \in S_0$.

- Pick next state to s_0 that also satisfies Z_k. This is an existential quantifier problem. We want a state t such that ψ holds on t. Here s is a fixed state. $\phi(t) = R(s,t) \wedge \psi(t)$. One can use a SAT solver to get this, just finding any state that satisfies both of these. The reason we do this is that we are seeking an infinite path, a looping path, and we know every element of our sequence has to be in Z_k.

- Repeat the previous step up to k times. We may even be able to do it fewer times. But by the time we reach k repetitions, we must have reached an item that was already seen, and thus identified a loop.

To understand why this works, recall that the fixpoint computation terminated in k steps. This means that in k steps we were able to see every single state that satisfies EG p from initial states.

In practice, one does not simply repeat the second step k times. Rather, one checks whether the new state is equal to one of the predecessor states. If yes, we have found a loop. If no, the search continues.

1.3 Fairness

When one looks for counterexamples, one wants to identify counterexamples that could actually happen, not paths that only appear because of abstraction, that sort of thing. Consider concurrent programs; the scheduler has some procedure for choosing which processes run when, but maybe we abstract it with simple non-determinism. That approach is appealing because we are not required to model as much. However it might also mean that one process is allowed to be starved of execution, in a way that a real scheduler would not allow, giving rise to spurious counterexamples if some process never runs. There are multiple ways to deal with this. We could add constraints to the scheduler, maybe in the form of additional LTL or CTL properties. The issue with that approach is that the problem blows up, becomes more complicated. A more elegant way of dealing with this is not to impose these assumptions in the problem, but rather to look for fair counterexamples.

Fairness: 'fairness constraints' are defined as predicates, usually a set of predicates $\{f_1, ...f_k\}$ that must hold infinitely often along any path.

These fairness constraints offer a clean way to restrict counterexamples, for instance to counterexamples in which a process is never starved.

1.3.1 Fairness for EG p

The fixed point formulation of EG p with fairness predicate $\{f\}$, where $\{f\}$ is a singleton set, one fairness predicate is: $\nu Z.p \wedge (EX\ E[p\ U\ (f \wedge Z)])$.

We can interpret this to mean that p must be true, and there exists a next state where from that state there exists a path where p is true and remains true until a state where the fairness condition holds, and that state also satisfies EG p with fairness. Essentially this means we seek a witness path where f holds infinitely often, and that constraint is woven into the fixpoint computation.

For a thorough treatment of the topics in this section, see "Verification tools for finite-state concurrent systems" [2].

2 Bounded Model Checking

You can think of Bounded Model Checking (BMC) as symbolic model checking without BDDs. Formally the problem is this:

Given: A kripke structure M, an LTL property ϕ, and an integer bound k.

Question: Is there an execution of M of length at most k that violates ϕ? We can write this: $M \models_k \phi$.

The first obvious question is how does one know that the k value is good enough? If we find no bugs with k but increase the bound to $k+1$, we might find a bug. Early in its development, some said this was just a good testing strategy, not a verification technique. However now BMC engines can be used for proving properties and for synthesis. The BMC encoding is the most useful idea that comes out of this, so think of what follows as a way of encoding the question of whether a structure satisfies a property for k steps.

The basic approach is to unroll the transition relation k times and add appropriate constraints.

2.1 Finite Length Counterexamples

Consider $\phi = \mathrm{G}\ p$. The BMC encoding to SAT or SMT is:

$$BMC_k^{M,\phi}(v_0, v_1, ..., v_k) \triangleq S_0(v_0) \wedge R(v_0, v_1) \wedge R(v_1, v_2) \wedge ... \wedge R(v_{k-1}, v_k) \wedge (\bigvee_{i=1}^{k} \neg p(v_i))$$

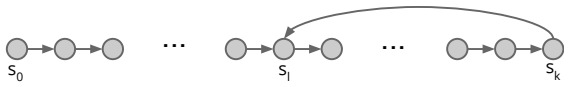


Figure 1: The structure of a (k,l)-loop.

2.2 Infinite Length Counterexamples

For safety properties, there is a finite length counterexample, but for liveness, the counterexample length is infinite.

Consider $\phi =$ GF p. The only way to see a violation is if there exists an infinite execution in which eventually not p applies throughout. So we look for bounded length executions that have a loop in them, a (k,l)-looping path. A path with a (k,l)-loop starts at initial state s_0, eventually reaches a state s_l, then reaches a state s_k which has an edge back to s_l. We illustrate such a path in Figure 1. Such a path is a counterexample for GF p if no state in the loop satisfies p, since the program can loop in it forever.

The BMC encoding to SAT or SMT is:

$$BMC_k^{M,\phi}(v_0, v_1, ..., v_k) \triangleq S_0(V_0) \wedge R(v_0, v_1) \wedge ...R(v_{k-1}, v_k) \wedge \bigvee_{l=0}^{k-1} [R(v_k, v_l) \wedge \bigwedge_{i=l}^{k} \neg p(v_i)]$$

We can read this as requiring that there must be a loop back from state k to state l, and all the states along l to k must not satisfy p. If bounded model checking finds such a (k,l)-loop, the property is not satisfied. If it does not reveal such a loop, then we can only be sure that there is no loop for our chosen bound.

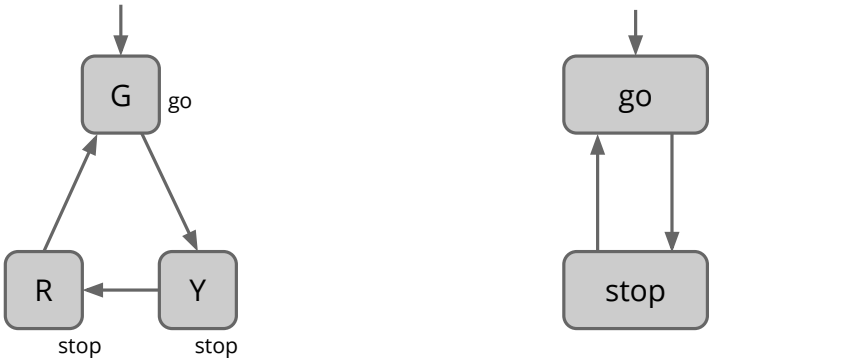
There are more efficient encodings for making this faster that are not covered in this lecture. This encoding is very useful as a tool, and it is used in many areas, even including AI planning and control.

For a thorough treatment of the topics in this section, see "Bounded Model Checking" [1].

3 Abstraction

At a high level, abstraction is the process of hiding information. This is advantageous in the context of model checking because hiding irrelevant information allows us to check smaller models, more amenable to model checking techniques.

The original model is called the concrete model, $M = (S, S_0, R, L)$. In abstraction, the aim is to find an abstract model $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, L)$. Notice that we use the same labeling



(a) The stoplight example. (b) The stoplight example, abstracted

Figure 2: Abstracting the stoplight example.

function. Usually one picks a labeling function based on what one wants to prove, so L typically should not be abstracted. This is not always the case, but for simplicity's sake, this lecture will assume it does not change.

The goal, informally, is to generate $\hat{M}$ from M such that $|\hat{M}| \ll |M|$ and $\hat{M} \models \phi$ iff $M \models \phi$.

The approach is to compute an abstraction function $\alpha : S \to \hat{S}$. This function maps a concrete state to an abstract state. It induces an equivalence relation amongst elements of S. $\forall s_1, s_2 \in S$, $s_1 \equiv s_2$ iff $\alpha(s_1) = \alpha(s_2)$.

For example, we might abstract the stoplight model in Figure 2a into the model in 2b, mapping both the R and Y states to the stop state.

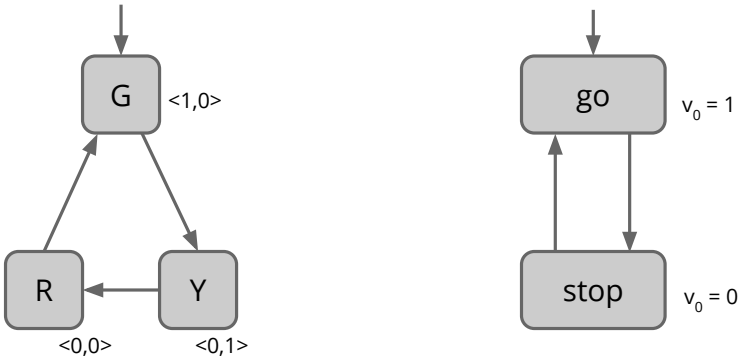
The goal that $\hat{M}$ be smaller is a soft constraint. Often abstraction techniques do not in fact guarantee that property. In contrast, the restriction that proving a property on $\hat{M}$ means it must hold on the real M is a hard constraint.

Given α, we can define $\hat{M}$ in terms of S, S_0, R, and L.

$\hat{S_0} = \{\hat{s_0} | \exists s_0 . s_0 \in S_0 \wedge \alpha(s_0) = \hat{s_0}\}$

$\hat{R} = \{(\hat{s}, \hat{t}) | \exists s, t . s, t \in S \wedge R(s, t) \wedge \alpha(s) = \hat{s} \wedge \alpha(t) = \hat{t}\}$

If there exists a transition from any element in s's equivalence class to any element in t's equivalence class, then there is a transition from $\hat{s}$ to $\hat{t}$. Thus this is an existential abstraction.



(a) The stoplight example. (b) The stoplight example, abstracted

Figure 3: The stoplight example with a two variable encoding.

3.1 Localization Abstraction

The main idea in computing the abstraction function, α, is to use ϕ. In particular, this lecture discusses one way of doing this, using localization abstraction.

The central technique that underpins localization abstraction is partitioning the state variables in the system into visible variables and invisible variables. $\hat{M}$ only models updates to visible variables. Invisible variables do not appear in the abstract model; they are assumed to take any arbitrary value. We delete all updates for them.

This technique can be applied with any subset of variables, but an arbitrary subset will not necessarily preserve the property of interest.

As an example, consider the stoplight example again. We could use the two bit encoding depicted in Figure 3a to describe the states. If we then defined an α function such that $\alpha(\langle v_0, v_1 \rangle) = v_0$, essentially ignoring one of the bits, we would be using the abstraction in Figure 3b. With this abstraction in place, we could check $\phi \triangleq G(v_0 \Rightarrow X \neq v_0)$.

One method for computing abstractions is Cone of Influence, the classic technique, and the state-of-the-art approach through the 90s. The high-level idea is to compute syntactic dependencies between propositions in ϕ and other variables in M. Essentially, one computes the static slice of the model with respect to the variables that appear in the target property.

Let us consider the stoplight example again.

Suppose $\phi \triangleq G(v_0 \Rightarrow X \neq v_0)$.

We get the next values of each variable as follows:

$v_0' := ITE(\bar{v_0}\bar{v_1}, 1, 0)$

$v_1' := ITE(v_0\bar{v_1}, 1, 0)$

In this case, both v_0 and v_1 would have to be visible, according to the Cone of Influence approach. There would be no abstraction.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[2] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 124–175. Springer Berlin Heidelberg, 1994.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 4/1: Abstraction, Interpolation, Inductive Invariant Checking

Scribe: Rafael Dutra *Editor: Baihong Jin*

In this lecture, we review two approaches for Abstraction Model Checking, one counterexample-based (CEGAR) and one proof-based (PBA). In addition, we also introduce the techniques of Interpolation and Inductive Invariant Checking.

Remember that a Kripke structure over a set AP of atomic propositions is a 4-tuple $M = (S, S_0, R, L)$, where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a left-total transition relation and $L : S \to 2^{AP}$ is a labeling function. An abstraction function $\alpha : S \to \hat{S}$ defines a transformation from a concrete Kripke structure $M = (S, S_0, R, L)$ into an abstract one $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, L)$.

The lecture will focus on Localization Abstraction, in which variables are partitioned into two subsets, visible and invisible. Other examples of abstractions are Predicate Abstraction, based on Atomic Propositions (AP) and Abstract Interpretation, based on Abstract Domains.

For the Localization Abstraction, a *cone of influence* reduction can be applied to remove all variables from the model that do not affect the system properties being checked. Only variables that are present in the properties and variables that influence them need to be considered.

1 Counterexample-Guided Abstraction Refinement (CEGAR)

CEGAR [2] is an approach to automate both the abstraction and the refinement processes, that was shown to be very effective in practice. The idea of CEGAR is to first hypothesize an abstraction α and then check if the abstract model $\hat{M}$ obtained satisfies the property ϕ. If $\hat{M} \models \phi$, then it is safe to conclude that ϕ is also satisfied by the concrete model, since the existential abstraction $\hat{M}$ is an over conservative representation of the possible behaviors of M. And if $\hat{M} \not\models \phi$, the idea is to use some feedback from counterexamples to refine the abstraction α.

Figure 1 shows the diagram of the CEGAR approach. First, an abstract model $\hat{M}$ is constructed based on model M, property ϕ and one initial abstraction function α. For the initial abstraction, we may try to abstract away as many variables as possible, to obtain a simple $\hat{M}$. This model is passed to the model checker, that checks whether it satisfies the property ϕ. If $\hat{M} \models \phi$, we are done, as discussed earlier. Otherwise, the model

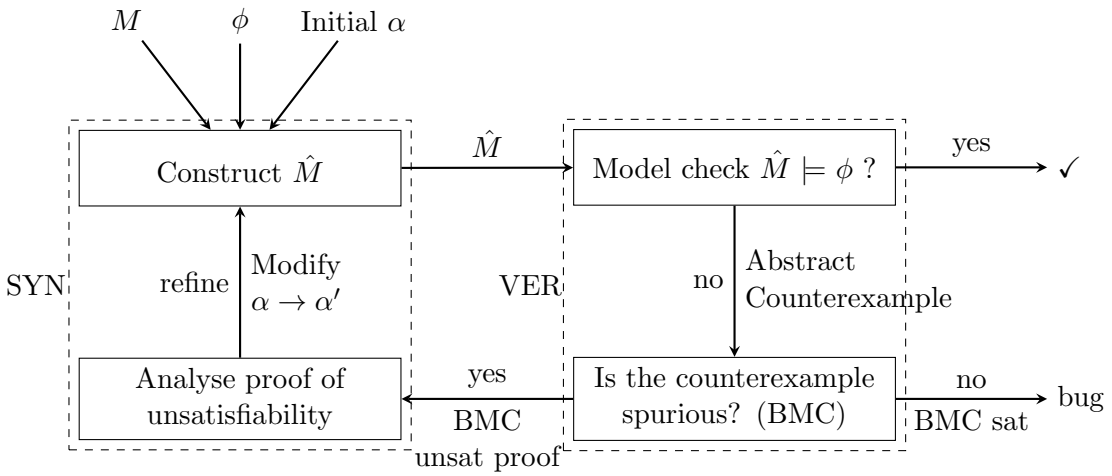


Figure 1: CEGAR diagram

checker provides an abstract counterexample and we use Bounded Model Checking (BMC) to determine if this is a spurious counterexample or if it represents a concrete bug. If a bug is found, the process terminates, but if the counterexample is spurious, the BMC returns an unsatisfiability proof. This proof is then analysed to refine the abstraction α and the cycle restarts.

CEGAR can be seen as an example of Counterexample-Guided Inductive Synthesis (CEGIS). The synthesizer and the verifier components are shown as dashed rectangles in Figure 1. The main contributions of CEGAR are the methods to detect spurious counterexamples with BMC and the method to analyse proofs of unsatisfiability to generate a refinement of the abstraction (the two bottom rectangles in Figure 1).

The refinement of α is done by transforming it into an α' that satisfies

$$\forall s, t \in S \quad \alpha'(s) = \alpha'(t) \Longrightarrow \alpha(s) = \alpha(t) \tag{1}$$

$$\exists s, t \in S \quad \alpha(s) = \alpha(t) \wedge \alpha'(s) \neq \alpha'(t) \tag{2}$$

Basically, (1) says that all states that are merged in α' are also merged in α and by (2) we know that there are some states merged in α that are not merged in α'. So α' is a strict refinement of α. For example, in the case of $\alpha = \{(G, go), (Y, stop), (R, stop)\}$ a valid refinement would be $\alpha' = \{(G, go), (Y, stop1), (R, stop1)\}$, in which the *stop* states are no longer merged.

In the case of Localization Abstraction, the program variables are partitioned into visible and invisible ones. Let the n program variables be $v = (v_0, v_1, \ldots, v_{n-1})$, of which the first m variables $(v_0, v_1, \ldots, v_{m-1})$ are visible (they can affect the property being checked). In

this case, the concrete state is a valuation of all variables $s = (s_0, s_1, \ldots, s_{n-1})$. On the other hand, the abstract state is a valuation to only visible variables $\hat{s} = (s_0, s_1, \ldots, s_{m-1})$, where invisible variables $v^- = (v_m, v_{m+1}, \ldots, v_{n-1})$ are not considered.

If the property being checked is $\phi = G\, p$, then an abstract counterexample with k transitions is given by a run $\hat{s_0} \to \hat{s_1} \to \cdots \to \hat{s_k}$, with $\hat{s_0} \in \hat{S_0}$ and abstract state $\hat{s_k}$ satisfies $\neg p$.

To check if the counterexample is spurious, CEGAR generates a SAT formula

$$\psi(v_0^-, v_1^-, \ldots, v_k^-) = S_0(\hat{s_0}, v_0^-) \wedge R((\hat{s_0}, v_0^-), (\hat{s_1}, v_1^-)) \wedge \cdots \wedge R((\hat{s_{k-1}}, v_{k-1}^-), (\hat{s_k}, v_k^-)) \wedge \neg p(\hat{s_k}),$$

that is checked by the Bounded Model Checker. If the formula is satisfiable, we have found a concrete violation of the property ϕ. Otherwise, a proof of unsatisfiability is obtained.

The proof is analysed by representing it with a Directed Acyclic Graph (DAG) that starts with a set of initial clauses and performs resolution until a contradiction is obtained. The clauses that do not participate in the proof are pruned away and one invisible variable is selected remaining clauses (the clauses that lead to the conflict). This variable is then made visible in the refined abstraction. Since the number of variables is finite, the CEGAR loop is guaranteed to converge for Localization Abstraction: in the worst case, all variables will become visible and the 'abstract' model will actually be the same as the concrete model.

For other abstractions, it may not be simple to guarantee termination of the CEGAR loop. For example, for Predicate Abstraction, there are usually infinitely many possible predicates, such as $x \geq c$ for every variable x and constant $c \in \mathbb{Z}$.

2 Proof-Based Abstraction (PBA)

The idea of Proof-Based Abstraction (PBA) is method for abstraction refinement based on the proofs of unsatisfiability. Unlike CEGAR, it does not learn from counterexamples. Figure 2 shows the diagram for PBA. The method begins by applying a Bounded Model Checker (BMC) to check if property ϕ is satisfied by traces with some bounded length k. If $M \not\models_k \phi$, BMC will return a concrete counterexample of length at most k. Otherwise, BMC returns a proof of unsatisfiability, which shows that no trace of length at most k violates ϕ. In this case, the proof is analysed to decide which unvisible variables should be made visible, like it was done in CEGAR. The new refined abstract model is checked. If $\hat{M} \models \phi$, we can safely conclude that the concrete model also satisfies ϕ. Otherwise, the model checker produces a counterexample trace, that must have length greater than k, since the property was verified for k steps. We then increase the value of k, to try to find a violation that may be present only in larger traces.

N. Amla et al. [1] compared PBA vs. CEGAR and showed that none of the techniques was able to dominate the other on all sets of benchmarks. There are some problems more suitable for PBA and some more suitable for CEGAR.

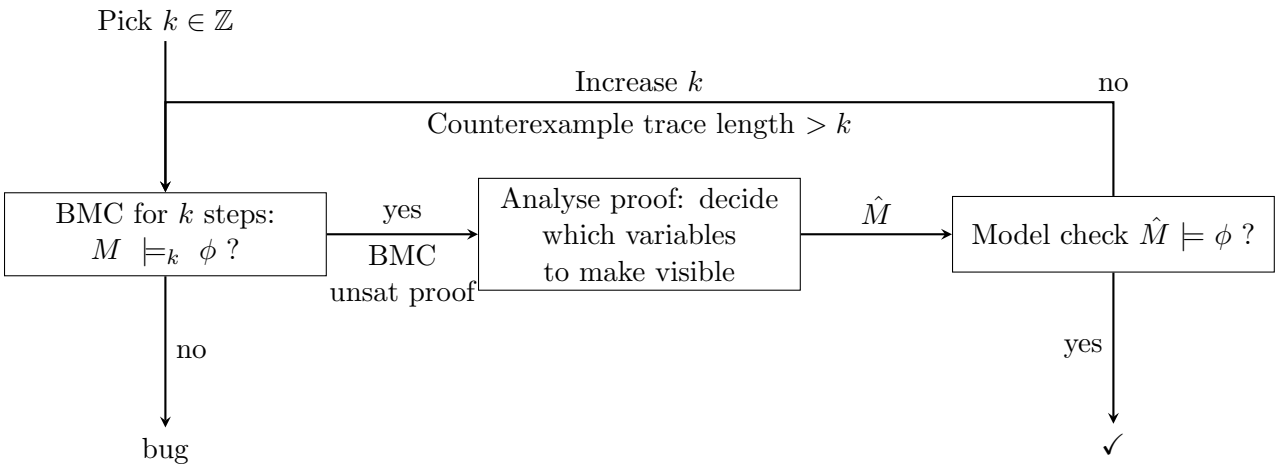


Figure 2: PBA diagram

3 Interpolation

Definition 3.1 *Given an unsatisfiable formula $A \wedge B$, a Craig Interpolant from A to B is a formula I that satisfies*

- $A \Longrightarrow I$,
- $I \wedge B$ *is unsatisfiable (in other words,* $I \Longrightarrow \neg B$*),*
- I *uses only common variables of A and B (in other words,* $FV(I) \subseteq FV(A) \cap FV(B)$*).*

For example, for the formulas $A = x \wedge y$ and $B = \neg y \wedge z$, over boolean variables x, y, z, $I = y$ is a Craig Interpolant from A to B.

An interpolant can be obtained by analysing the proof of unsatisfiability of $A \wedge B$, creating a boolean circuit representing the proof by resolution and using it to identify the relevant clauses. For boolean satisfiability problems (SAT), Craig Interpolants always exist and are also SAT formulas. For more complex theories, the Craig Interpolant may not exist or may only be expressible in some more powerful theory that includes, for example, quantifiers or non-linear operators.

Craig Interpolants can be used for unbounded symbolic model checking, as proposed by McMillan [3]. For example, to check if the property $\phi = G\, p$ holds, we may create a formula BMC_k that expresses a path with k transitions that violates p in its final state as

$$BMC_k = S_0(v_0) \wedge R(v_0, v_1) \wedge R(v_1, v_2) \wedge \cdots \wedge R(v_{k-1}, v_k) \wedge \neg p(v_k).$$

By considering $A = S_0(v_0) \wedge R(v_0, v_1)$ and $B = R(v_1, v_2) \wedge \cdots \wedge R(v_{k-1}, v_k) \wedge \neg p(v_k)$, we can obtain an interpolant $I(v_1)$.

Let S_1 represent the set of states that can be reached after one transition. Then $S_1(v_1) = \exists v_0 \quad S_0(v_0) \wedge R(v_0, v_1)$. From the definition of the Craig Interpolant, it is easy to see that $S_1 \subseteq I$, which means that I is an over-approximation of the set of states represented by S_1. Moreover, the set of states in I still have the property that they do not lead to a violation of ϕ in the next $k-1$ steps, otherwise $I \wedge B$ would be satisfiable.

An algorithm for Interpolation-Based Model Checking is sketched below, where Z represents an approximation of the set of reachable states.

1. $Z(v) \leftarrow S_0(v)$
2. Check BMC from Z with k steps:
   (a) case satisfiable:
      - if $Z = S_0$: return counterexample
      - if $Z \neq S_0$: "abort", increment k and restart

   (b) case unsatisfiable: continue to 3.
3. $oldZ \leftarrow Z$

   $Z \leftarrow Z \cup I$
   - if $oldZ = Z$: return "property verified"
   - if $oldZ \neq Z$: goto 2.

The algorithm starts with Z as the set of initial states and procedes by running a Bounded Model Checker with k steps. If BMC returns satisfiable and Z is still equal to S_0, then we found a concrete violation of the property in k steps, and return this counterexample. If BMC returns unsatisfiable, then the set Z is expanded by taking a union with the interpolant I. After this operation, it is still impossible to reach a violation in $k-1$ step, due to the interpolant property. However, for k steps, there is no guarantee. If, after the operation $Z \leftarrow Z \cup I$, the set Z does not grow, we have reached a fixed point and then we can guarantee that the property ϕ is satisfied. This happens because, in this case, we have $I \Rightarrow Z$, so from $Z(v_0) \wedge R(v_0, v_1)$, we obtain $I(v_1)$, which implies $Z(v_1)$. We can continue this argument and prove, by induction, that the property ϕ is satisfied for any number of steps, since all reachable states have been explored. Note that, in this case, Z is an inductive invariant.

If the set Z grows, we repeat the process with the expanded set Z. Note that if BMC returns satisfiable but the set Z is strictly larger than S_0, then the satisfying assignment may not be a real counterexample, because of the over-approximation of initial states. So in this case we abort the algorithm and restart it with an incremented bound k.

4 Inductive Invariant Checking

Similarly to the algorithm for Interpolation-Based Model Checking, we can apply induction proofs with more general inductive invariants. For example, to prove a property $\phi = G\,p$, the simplest proof by induction would be proving the base case $S_0(v) \Rightarrow p(v)$ and the induction step $p(v) \wedge R(v, v') \Rightarrow p(v')$. The main idea of inductive invariants is that sometimes the proof can be made easier by using a strong induction hypothesis that includes an induction invariant ψ. In this case, the induction step that needs to be proved becomes $\psi(v) \wedge p(v) \wedge R(v, v') \Rightarrow p(v') \wedge \psi(v')$. Strengthening the induction hypothesis can avoid exploring some states that are not reachable in a concrete execution.

References

[1] Nina Amla and Ken L. McMillan. A Hybrid of Counterexample-Based and Proof-Based Abstraction. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 260–274. Springer Berlin Heidelberg, 2004.

[2] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[3] Ken L. McMillan. Interpolation and SAT-Based Model Checking. In Jr. Hunt, Warren A. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 2003.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: Sanjit A. Seshia*

Lecture 4/6: Deductive Verification: k-induction and IC3/PDR

Scribe: Liang Gong *Editor: Baihong Jin*

This document summarizes the latter half of the lecture on deductive verification. More specifically, we first introduce some general concepts on *proof by induction*; then we introduce the *IC3/PDR algorithm*[1].

1 Proof by Induction

In the context of model checking, *induction* is a method of proof typically used to establish that a given property formula holds for all reachable states. It is often done in two steps. The first step, known as the *base step*, is to prove that the set of initial states (S_0) satisfies a given property P:

$$S_0 \Rightarrow P$$

The second step, known as the *inductive step*, is to prove that if the given property P holds for the current set of states, then property P still holds for the set of states obtained after one step of transition (R is the transition relation):

$$P \wedge R \Rightarrow P'$$

From these two steps, we can infer that the given property holds for all states that are reachable (according to the transition relation R) from the set of initial states.

Example 1: Consider the following example with two variables: $x, y \in \mathbb{Z}$

$$\begin{aligned} S_0 &\triangleq x = 1 \wedge y = 1 \\ R &\triangleq x' = x + 1 \wedge y' = y + x \\ P &\triangleq y \geq 1 \end{aligned}$$

To prove the example by induction, we need to construct its base step and induction step.

Base Step, where on the left hand side is the initial condition:

$$x = 1 \wedge y = 1 \quad \Rightarrow \quad y \geq 1$$

[1]Due to the complexity of the algorithm and the time constraint in class, some specific details of the IC3/PDR algorithm are not introduced in the class and therefore not documented in this lecture notes. If you are interest in digging into more details, the original paper [1] is a good reading.

Induction Step:

$$y \geq 1 \wedge x' = x + 1 \wedge y' = y + x \quad \Rightarrow \quad y' \geq 1$$

Unfortunately, the induction step constructed following $P \wedge R \Rightarrow P'$ does not hold in this case. If we send the negation of the formula to a SMT solver, we will get an assignment (e.g., $y = 1, x = -10$) indicating the formula in the induction step does not hold.

To fix this problem, *invariant strengthening* is often deployed by adding *auxiliary invariants* to further constrain the variables. In this case, we add an auxiliary invariant $x \geq 1$ (denoted by Inv) as an additional strengthening invariant.

Before we can use the auxiliary invariant Inv, we must first show that Inv itself is inductive:

$$S_0 \Rightarrow Inv$$

$$Inv \wedge R \Rightarrow Inv'$$

More specifically, we need to show the follow formula is true for this example:

$$x = 1 \wedge y = 1 \quad \Rightarrow \quad x \geq 1$$

$$x \geq 1 \wedge x' = x + 1 \wedge y' = y + x \quad \Rightarrow \quad x' \geq 1$$

Finally, we add the auxiliary invariant into the induction step:

$$P \wedge Inv \wedge R \Rightarrow P' \wedge Inv'$$

i.e., the following formula, which is true, in this example:

$$y \geq 1 \wedge x \geq 1 \wedge x' = x + 1 \wedge y' = y + x \quad \Rightarrow \quad y' \geq 1 \wedge x' \geq 1$$

The induction step holds after the auxiliary invariant is added.

In this case, the auxiliary invariant is simple and involves only one variable. However, sometimes the auxiliary invariant may consist of multiple assertions and each one of these assertions may not be inductive on its own as shown in the following example.

Example 2: Consider the following example involving two variables: $x, y \in \mathbb{Z}$

$$S_0 \triangleq x = 1 \wedge y = 1$$

$$R \triangleq x' = x + y \wedge y' = y + x$$

$$P \triangleq y \geq 1$$

In this example, the inductive strengthening of P is $x \geq 0 \wedge y \geq 1$. However, neither $x \geq 0$ nor $y \geq 1$ is inductive on its own.

Generally speaking, it is likely that induction strengthening is needed when applying proof by induction. As a rule of thumb of using proof by induction with inductive strengthening: 1) the auxiliary invariant must be proven as inductive; 2) each induction step incorporating the auxiliary invariant should be proven as inductive.

2 The IC3/PDR Algorithm

IC3 stands for **I**ncremental **C**onstruction of **I**nductive **C**lauses for **I**ndubitable **C**orrectness[2]. Ever since its proposal by Aaron R. Bradley in 2011 [1], IC3/PDR has been considered as one of the fastest SAT-based model checking algorithms[3]. The algorithm combines SAT-based symbolic model checking, incremental proof, over-approximation and proof by induction techniques.

Similar to the interpolation-based model checking (ITP) algorithm introduced in the previous lectures, IC3 also computes over-approximations of the sets of reachable states. However, the interpolation-based approaches require unrolling the model based on the transition relations in order to obtain the over-approximations, which often leads to large query formulas for the underlying SAT solver. In contrast, IC3 obtains the over-approximations by performing relatively more but smaller local checks without unrolling.

2.1 Preliminaries on Symbolic Model Checking

IC3 is a SAT-based symbolic model checking algorithm. In this section, we introduce some background knowledge of symbolic model checking as well as some key observations that spark the idea of over-approximation upon which IC3 and interpolation-based approaches are developed.

In model checking, we want to check if a property P holds for a model M. The model M is described by:

- A set V of Boolean variables. The state space consists of $2^{|V|}$ states.
- The set of initial states denoted by S_0.
- The transition relation R.

The property P is a Boolean formula, which can be considered as the set of states that satisfy P. The reachable state space S^* is the set of states that can be reached from S_0 by taking any number of transitions defined by R.

Suppose that property P holds in the model ($M \models P$), it means that the set of all reachable states S^* is contained by the set of states that satisfy P:

$$S^* \subseteq P$$

[2]It is also known as the **P**roperty **D**irected **R**eachability (PDR) algorithm.
[3]It is a symbolic model checking algorithm that uses SAT solving as a subroutine.

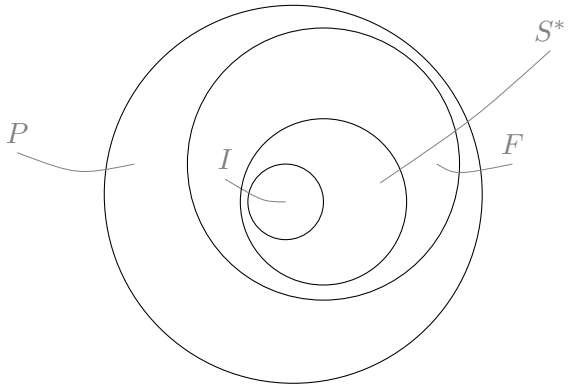


Figure 1: An intuitive view of over-approximation in symbolic model checking. F is an over-approximation of the set of reachable states to be described in Section 2.2.

This is equivalent to the following implication relationship:

$$S^* \Rightarrow P$$

This can be checked by checking the satisfiability of $S^* \wedge \neg P$. An intuitive view of those concepts are shown in Figure 1.

2.2 What is IC3?

In the previous section, we have described S^*, which is the set of all states that can be reached from S_0 by taking any number of transitions defined in R. If we take a transition from any state in S^*, we will still arrive at a state in S^*:

$$S^* \wedge R \Rightarrow S'^*$$

Therefore, S^* is a fix point for the transition relation R.

Observations: For a set of states F, with $S_0 \in F$, if F is a fix point, it must contain S^*:

$$S_0 \subseteq F \wedge (F \wedge R \Rightarrow F') \Rightarrow S^* \subseteq F$$

Therefore, F is an over-approximation of S^*. If are can find such a set F that $F \subseteq P$, we then prove that $M \models P$.

The Main Idea of IC3: If we compute a sequence of sets of states:

$$S_0, F_1, F_2, F_3, ..., F_k$$

such that $\forall i, F_i$ is an over-approximation of the set of states reachable from S_0 in i steps or less, and each F_i satisfies the property P. If there exists a i such that $F_i = F_{i+1}$ then a fix point is reached, which leads to the conclusion that $M \models P$.

Properties of IC3:

- The IC3 algorithm is proposed for a finite state system.
- All sets are represented in Boolean formulas and every F_i is represented as a Boolean formula in CNF.
- $S_i \subseteq F_i$, where S_i is the set of states reachable in i or fewer steps, i.e., F_i is an over-approximation of S_i.
- $F_i \subseteq E_{k-i+1}$, where E_{k-i+1} represents the set of states that are $k-i+1$ steps from the error states (states that satisfy $\neg P$).
- F_i satisfies the following four properties:
  - $F_0 = S_0$
  - $\forall i, F_i \Rightarrow F_{i+1}$
  - $\forall i, F_i \wedge R \Rightarrow F_{i+1}$
  - $\forall i, F_i \Rightarrow P$, i.e., every step from F_0 to F_k is safe.

When does IC3 prove GP?

If for some i, $F_i = F_{i+1}$, in conjunction with the following properties from IC3:

$$S_0 \Rightarrow F_i$$

$$F_i \wedge R \Rightarrow F_{i+1}$$

We can show that F_i is inductive. Since we also have the property: $\forall i \in [0..k], F_i \Rightarrow P$, according to proof by induction, $\mathbf{G}P$ is satisfied.

When does IC3 find a bug?

Given the sequence:

$$S_0, F_1, F_2, F_3, ..., F_k$$

If the following formula does not hold:

$$F_k \wedge R \Rightarrow P$$

Then the SAT solver has produced a counterexample, which includes a state $(s \in F_k) \wedge (s \in \neg P)$, i.e., F_k is one step away from violating P. Here we call s as *counterexamples to the inductiveness* (CTI) of the given property P. [4]

[4] More generally, a CTI can be viewed as a set of states represented by a conjunction of literals.

After obtaining a CTI (i.e., s), IC3 will try to find out if s is unreachable by trying to prove that $\neg s$ is inductive:

$$(S_0 \Rightarrow \neg s) \wedge ((F_i \wedge \neg s \wedge R) \Rightarrow \neg s')$$

In the formula, i is the maximal number such that $F_i \wedge \neg s \wedge R \Rightarrow \neg s'$.

- If the number i exists, then we know that $\neg s$ is inductive within i steps. then we know that all states in s is unreachable within i steps. Therefore we can further restrain F_i:

$$\forall i \in [0..i], F_i \leftarrow F_i \wedge \neg s$$

- If the number i does not exist, then we know that $\neg s$ is not inductive within k steps, and therefore the states in s are reachable within k steps. Therefore P does not hold in the model M.

In reality, the existing implementation of IC3 algorithm will try to generalize $\neg s$ to a clause c which contains subset literals from $\neg s$. The IC3 algorithms has a number of heuristics to take s and refines it into c.

What if F_k is not inductive? Suppose the property P holds within k steps:

$$F_k \wedge R \Rightarrow P$$

Unfortunately, if F_k is not inductive:

$$F_k \wedge R \not\Rightarrow F_k'$$

IC3 will try to obtain a new inductive F_k by propagating those inductive subset clauses from F_i to F_{i+1}, $i \in [1..k]$.

Key observation: Each F_i is written in the CNF form (e.g., $c_1 \wedge c_2 \wedge ...$) and therefore is a conjunction of clauses (a.k.a., *predicate*s). Each one of those clauses appears in F_i because it is true for at least i steps.

The high-level idea is to compute the largest subset of clauses from F_i, for all $i \in [1..k]$ (The subset of clauses is indicated by C), s.t.:

$$F_i \wedge R \Rightarrow C'$$

C will also be conjoined into F_{i+1}:

$$F_{i+1} \leftarrow C \wedge P$$

This step is called *predicate abstraction* [1, 2], because it enlarges (abstract) the set by taking the subset of those predicates (i.e., clauses).

After this step, IC3 will obtain a new frontier F_{k+1}. If $F_k = F_{k+1}$, then F_k is inductive.

References

[1] A. R. Bradley. Sat-based model checking without unrolling. In R. Jhala and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[2] A. R. Bradley. Understanding IC3. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2012.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 4/6: Compositional Verification

Scribe: Ankush Desai *Editor: Rafael Dutra*

In the previous lectures we studied various verification approaches, such as *explicit-state model checking*, *symbolic model checking*, *bounded model checking* and *proof by induction*. In all the above cases, we assumed that the system to be verified is represented as a flat state transition system. But in practice most systems/programs are composed of multiple components/modules that are built hierarchically.

In this lecture, we will discuss a method for reducing the complexity of verifying systems composed of many interconnected components. The basic intuition is the following: if we can prove that each component satisfies its local property, and if we can also deduce that the conjunction of these local properties implies the overall global specification, then we can safely conclude that the composite system will also satisfy the global specification. Note that the complexity of verifying each component separately can be orders-of-magnitude lower than the complexity of verifying the composite system. The challenge of using *compositional verification* is that one must also prove, using sound inference rules, that verifying *local* component specifications implies the verification of the *global* specification for the composed system.

We will consider the *assume-guarantee* style of reasoning for compositional verification.

1 Assume-guarantee (AG) reasoning

The AG reasoning was first introduced by Pnueli in [1]. In this paper, Pnueli uses triples (just like *Hoare* triples) of the form $\{\phi\}M\{\psi\}$, which specify "under the assumption that input satisfies ϕ, the component M is guaranteed to satisfy ψ". This is similar to the Hoare logic, in which ϕ and ψ are called *precondition* and *postcondition* respectively.

To further explain the relation between Hoare logic and AG, let us consider a simple sequential imperative program (Fig. 1) that explains how pre-post conditions can be used for modular reasoning. The preconditions are assumptions made by the module on its input variables and postconditions are the guarantees provided by the module on its output variables. The modular verification of F consists of checking the following implication:

$$\phi(x) \wedge F(x, y) \rightarrow \psi(x, y)$$

```
pre φ(x) : {x ≥ 0}
function F(int x)
{
        y = x + 5;
        return y;
}
post ψ(x, y) : {y ≥ x}
```

Figure 1: Simple Imperative program with *pre-post* conditions

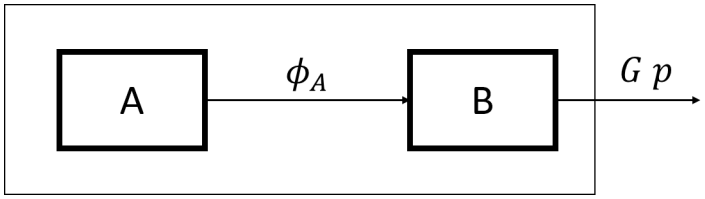


Figure 2: System with no cyclic dependency

The *Hoare logic* based modular reasoning is used for finite, terminating and not reactive programs. *Assume-guarantee* based reasoning is generally used in the context of reactive systems.

Consider the system composed of A and B as shown in Fig. 2. ϕ_A represents the assumption made by component B on output of component A. $\mathbf{G}p$ represents the property that must be satisfied by the output of component B (which is same as the output of the composed system). Equation 1 presents the inference rule for assume-guarantee reasoning of the system.

$$\frac{A \models \phi_A \quad B||\phi_A \models \mathbf{G}p}{A||B \models \mathbf{G}p} \tag{1}$$

The advantage of doing the verification in this manner is that we don't need to generate and examine the composite state space of the system $A||B$.

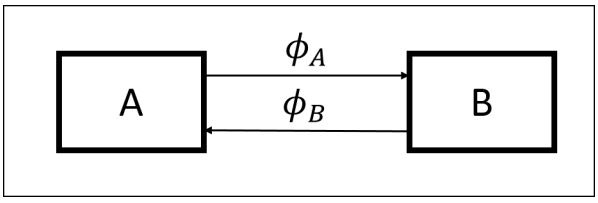


Figure 3: System with cyclic dependency

Fig. 3 presents a closed system where components A and B have *cyclic* dependency. ϕ_A represents the guarantee provided by A on its output which is assumed by B and ϕ_B represents the guarantee provided by B on its output which is assumed by A. $\mathbf{G}$p is the global specification of the system. Equation 2 presents the inference rule for the system with cyclic dependency.

$$\frac{A||\phi_B \models \phi_A \quad \phi_A||B \models \phi_B \quad \phi_A \wedge \phi_B \rightarrow \mathbf{G}p}{A||B \models \mathbf{G}p} \tag{2}$$

The check $\phi_A \wedge \phi_B \rightarrow \mathbf{G}p$ is performed using LTL satisfiability solver. A simple instance where such reasoning can be used is for example when $p \equiv \alpha \wedge \beta$, $\phi_A \equiv \mathbf{G}\alpha$ and $\phi_B \equiv \mathbf{G}\beta$.

Here, we assume that component A satisfies ϕ_A only under the assumption that ϕ_B holds on its inputs and B satisfies ϕ_B only if ϕ_A holds on its inputs. This apparent circular dependency can lead to unsound reasoning. To break the circular reasoning and generate a well founded proof, we use induction over time. If the temporal evolution of the system is taken into consideration, the output of a system at time t depends on its inputs at time $t-1$ which in turn depends on the inputs at time $t-2$ and so on. This intuition is used to break the circular dependency. If $P_t \Rightarrow Q_{t+1}$ represents "P holds until time t implies that Q holds at time $t+1$" which can also written using LTL formula $\neg(P\mathbf{U}\neg Q)$. We can then modify the Equation 2 to remove circular reasoning as shown in Equation 3.

$$\frac{A \models (\phi_{B_t} \Rightarrow \phi_{A_{t+1}}) \quad B \models (\phi_{A_t} \Rightarrow \phi_{B_{t+1}}) \quad \phi_A \wedge \phi_B \rightarrow \mathbf{G}p}{A||B \models \mathbf{G}p} \tag{3}$$

Note that the base of the induction is the fact that ϕ_A and ϕ_B are true at the initial time 0, expressed as $A \models \phi_{A_0}$ and $B \models \phi_{B_0}$. This is already included in the conditions of equation 3 if we use the convention that $\phi_{A_{-1}} = \phi_{B_{-1}} = true$.

References

[1] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In KrzysztofR. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 123–144. Springer Berlin Heidelberg, 1985.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 4/8: Inductive Learning and Verification

Scribe: Eric Kim *Editor: Andrew Head*

In these notes, we discuss 3 topics. First, we point out the connection between verification and synthesis, and how inductive learning is used in verification. Second, we define the *learning* problem, and discuss types of learning problems. Last, we introduce query-based learning, relating it to the task of constructing a direct finite automaton to replicate an observed system.

1 Verification "=" Synthesis

Many verification algorithms require additional information in order to prove properties of large systems. This expertise often comes in the form of domain expertise and structure in the system or property from a human expert. In fact, many artifacts are synthesized in the process of model checking:

a) *Inductive Invariants*: In inductive invariant checking, the induction hypothesis often need to be strengthened.

b) *Abstraction Functions*: In counterexample-guided abstraction refinement [3], an appropriate abstraction is synthesized to prove or disprove a property.

c) *Assume-Guarantee Reasoning*: The assumptions and guarantees between components need to be generated beforehand and usually require expert knowledge.

d) *Ranking Functions*: Ranking functions help establish proofs that a program eventually terminates. These are analogous to the role that inductive invariants play in verifying safety properties.

e) *Theory Lemmas*: In Lazy SMT, an appropriate SAT encoding is constructed by determining and adding lemmas to the underlying theory.

When the additional information is non-trivial and difficult to input, we seek to design algorithms that *learn* how to synthesize the appropriate verification artifact from examples.

2 Learning Basics

2.1 Inductive and Deductive Learning:

For our purposes in verification, learning can be thought of as synthesis using examples. Many common learning algorithms useful for formal methods are not taught in machine learning courses today.

Many types of learning exists and can broadly be put into two different paradigms 1) *induction*, which is the process of obtaining rules from observed examples and, 2) *deduction* the process of applying rules to draw conclusions about specific instances.

2.2 Terminology:

Broadly, for a learning problem where we seek to classify examples, we have:

- A *Domain* $\mathcal{D}$: the set of all possible examples.
- A *concept* $\mathcal{C} \subseteq \mathcal{D}$: the set of examples that the learner needs to correctly identify / classify by observing examples from $\mathcal{D}$. For example, if $\mathcal{D}$ is the set of all infinite traces, the learning algorithm needs to identify $\mathcal{C}$, the set of infinite traces of a finite state automaton.
- A *concept class*: the set of admissible concepts. It can typically be thought of as a power set of $\mathcal{D}$.

2.3 Types of Learning:

Machine learning algorithms are often constrained/classified according to their objective and what they are able to do. *Batch learning* algorithms are tasked with learning a concept or concept class from a set of examples. An algorithm is *exact* if it needs to correctly label all examples in $\mathcal{D}$ or not in $\mathcal{D}$, and is *approximate* if incorrect labels are permitted and the learning algorithm seeks to optimize an appropriate cost function. In formal methods, exact algorithms are much more prevalent.

Learning algorithms can also be *passive* or *active*. A passive algorithm cannot perform any actions and may only learn based off the examples provided to it. On the other hand, an active algorithm may choose examples to learn from, often by making queries to an oracle. An *oracle* is an entity – human, algorithm, or system – that provides learning examples.

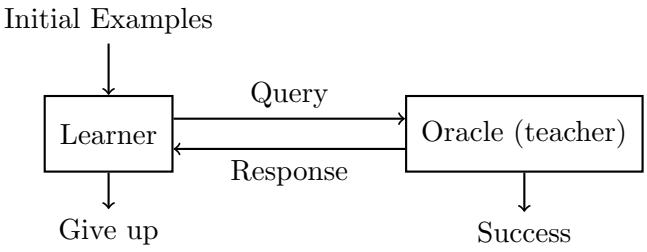


Figure 1: Query-based learning flowchart. From initial examples, a learning algorithm queries an oracle and incorporates the replies into its candidate concept.

3 Query-Based Learning

Conceptually, query-based learning can be defined as an interaction between two actors: a *learner* is trying to discover a concept, and makes queries to an oracle. The *oracle* (or teacher) provides positive and negative examples and feedback regarding whether the learner's concept is correct. Query-based learning follows three main steps (Figure 1):

1 The learner requests an initial set of examples.

2 The learner makes a query to an oracle of the form, "Do I have the right concept?", and receives a response from the oracle.

3 If the oracle responds no, the learner refines its concept with feedback from the oracle and returns to step 2, or gives up. If the oracle responds yes, then the learning algorithm terminates.

3.1 Types of Queries

There are four types of queries learners can make to their oracles during learning:

Query type	**Description**	**FSM example**
Membership	Is x in target concept $\mathcal{T}$?	Is trace x a positive example of concept $\mathcal{T}$?
Equivalence	Is candidate concept $\mathcal{C} \equiv \mathcal{T}$?	Are the languages $\mathcal{C}$ and $\mathcal{T}$ equivalent?
Subsumption	Is $\mathcal{C} \subseteq \mathcal{T}$?	Does target concept $\mathcal{T}$ contain all traces in $\mathcal{C}$
Sampling	Give a positive or negative example of $\mathcal{T}$.	Give a trace (not) contained in $\mathcal{T}$

For L^*, we consider only the *membership* and *equivalence* queries.

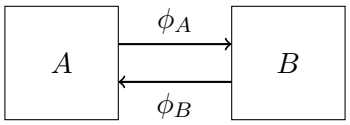


Figure 2: Example system to be compositionally verified. $\mathcal{A}$ and $\mathcal{B}$ are components of a composition, and ϕ_A and ϕ_B are the observable behaviors of each component.

3.2 An Application of Inductive Learning: Compositional Verification

One of the main hurdles in applying assume-guarantee techniques in compositional verification is determining the form of the assumption. Consider the system in Fig. 2 where we want to prove that system A satisfies property ϕ_A, but B and ϕ_B are either unknown or verifying systems A and B monolithically is prohibitive due to the state explosion problem. Generating an abstract model of B through its input-output properties allows us to avoid verification of "flat" or monolithic models while still verifying properties on the entire system. Cobleigh et al. [4] construct an appropriate abstraction, $\hat{B}$, over B's interface variables. Starting from an approximate assumption, the algorithm makes the abstraction more precise or *learns* the abstraction through repeated counterexamples and model checking.

3.3 Angluin's Algorithm

Angluin's L^* algorithm [1] demonstrates that one can learn a language equivalent automaton through query-based learning with membership and equivalence queries. In the space below, we offer a formal statement of the problem that L^* solves and walkthrough an example of the algorithm in learning a simple model.

We note that an algorithm cannot identify a deterministic finite automata with an equivalent language from membership queries alone. Briefly, membership queries are inadequate to identify an automaton's language because the language may be infinite but any algorithm will only have access to a finite set of queries.

3.3.1 Formal Statement

Let M be an a priori unknown deterministic finite automaton (DFA) defined by the tuple $M = (Q, q_0, \Sigma, \delta, L)$ where Q is a set of finite states, q_0 is an initial state, Σ is an input alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, and $L : Q \to 2^{AP}$ is an output function mapping states to sets of atomic propositions. Assume Σ and AP are known and there exists a way to initialize M to q_0.

Objective: Using membership and language equivalence queries to an oracle, learn an automaton $\hat{M}$ that has an equivalent language to M. Language equivalence is denoted by $\hat{M} \equiv M$.

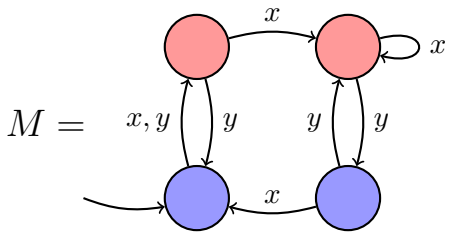
The set of examples are M's set of finite traces and the concept class is the set of all finite state automata with the same input alphabet Σ and atomic propositions AP.

3.4 Angluin's Algorithm Intuition

The learner initally only consists of a state q_0 and subsequently generates new states by supplying appropriate inputs to M and observing the outputs. Once the algorithm constructs an automaton $\hat{M}_1$ that is consistent with all observed outputs it queries the oracle to determine if the DFA is correct. If it is, then the learner is finished. Otherwise, the learner uses the counterexample provided to identify what new states to add. The algorithm continues to append new states and submit new queries "$\hat{M}_i \equiv M$" until the oracle declares $M \equiv \hat{M}_i$.

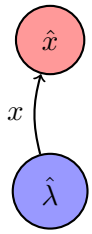
3.4.1 Example of Angluin's Algorithm

Consider the true DFA M depicted below with $\Sigma = \{x, y\}$ and $AP = \{$ • , • $\}$.

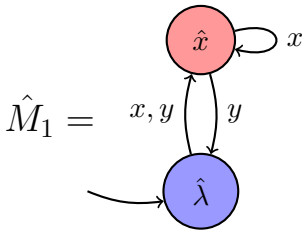


Let λ denote the empty string and $\hat{M}_i$ represent the ith DFA the learner queries for equivalence with the oracle. The learner first observes that M outputs blue at its initial state and creates a state $\hat{\lambda}$ in $\hat{M}_1$.

After inputting x and observing red, the learner adds another state $\hat{x}$ with a red label to $\hat{M}_1$.



After inputting sequences λ, x, y, xx, xy into M, the learner finally obtains a DFA $\hat{M}_1$ that is valid because it contains transitions out of all states for all inputs. It queries the oracle for equivalence of the languages admitted by the two DFAs M and $\hat{M}_1$.



Because $\hat{M}_1$ and M admit different sets of output traces, the oracle returns the input sequence $xxyx$ that serves as a counterexample to the DFA language equivalence query.

Automaton	Input Sequence	Output Sequence
$\hat{M}_1$	$xxyx$	● ● ● ● ●
M	$xxyx$	● ● ● ● ●

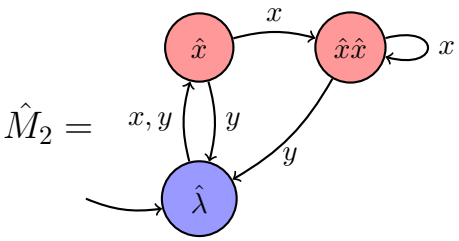
In order to identify the location of the output trace discrepancy's cause, the learner decomposes the counterexample into prefixes and suffixes. The suffixes represent "experiments" to be run.

Prefix	Suffix
λ	$xxyx$
x	xyx
xx	yx
xxy	x
$xxyx$	λ

The table below reveals that state $\hat{x}$ in $\hat{M}_1$ is inadequate to express the true behavior of M for experiment xyx.

State in current concept $\hat{M}_1$	Experiment	Result from M	Expected Result from $\hat{M}_1$
λ	$xxyx$	●	●
x	xyx	●	●
x	yx	●	●

Starting from state $\hat{x}$ in $\hat{M}_1$, the input sequence yx sees red as its final output. Because $\delta(\hat{x}, x) = \hat{x}$ for candidate $\hat{M}_1$, one would expect for a red to be the final output for any input sequence x^*yx (any string yx preceded by a finite number of x's), but this is not observed in practice on M. Thus, Angluin's algorithm adds an additional state $\hat{x}\hat{x}$ to get rid of this discrepancy. After experiment inputs xxx and xxy, we arrive at a second candidate DFA $\hat{M}_2$.



The oracle provides a counterexample $xxyx$ to the equivalence query $\hat{M}_2 \equiv M$. After modifying the candidate DFA another time, Angluin's algorithm recovers the DFA M. The L^* algorithm is guaranteed to learn the minimum DFA $\hat{M}$ that is language-equivalent to M. By minimum, we mean that it has the least possible number of states.

The complexity of Angluin's L* algorithm is $O(n^2(m+1) + m^2l)$, where n is the number of states in the minimum DFA, m is the number of transitions, and l is the length of the largest counterexample. The $n^2(m+1)$ term is a result of the queries and m^2l term arises from the analysis.

Researchers have extended the L* technique beyond learning determinite finite automata. For example, Angluin and Fisman [2] describe a technique for learning ω-automata.

References

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[2] Dana Angluin and Dana Fisman. Learning regular omega languages. In *Algorithmic Learning Theory - 25th International Conference, ALT 2014, Bled, Slovenia, October 8-10, 2014. Proceedings*, pages 125–139, 2014.

[3] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[4] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Psreanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619, pages 331–346, 2003.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: Dorsa Sadigh*

Lecture 4/13: Synthesis from Temporal Logic

Scribe: Omid Bagherieh and Baihong Jin *Editor: Dexter Scobee and Benjamin Mehne*

In this lecture, an overview on reactive synthesis from temporal logic specification is presented. The objective is to come up with a model which can guarantee satisfaction of a given property ϕ. This property is assumed to be represented in terms of linear temporal logic. Model checking techniques are utilized in order to investigate if a given model satisfies our desired property ϕ. In the case that this property is violated, the model needs to be redesigned, See Figure 1. Few disadvantages of the model checking technique have been illustrated here

- State explosion: There could be an exponential increase in the number of variables.
- Designing model (P) can be hard and expensive.
- Should $P \not\models \phi$, then we need to redesign the model which is also hard and expensive.

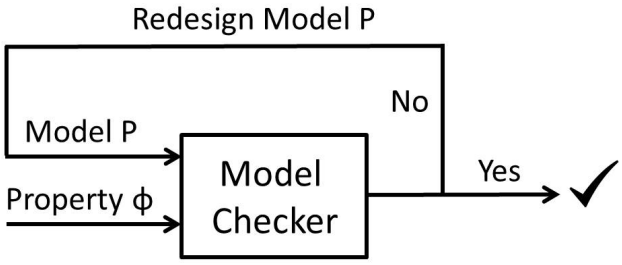


Figure 1: Model checker which investigates if model P satisfies property ϕ. SMV and SPIN are two tools used in model checking.

In the model checking technique, in addition to property ϕ, we always need a model of the system (P) to start with. This model is usually hard to obtain. Therefore, in this synthesis technique, the idea is to use a property ϕ and synthesize a model which is guaranteed to satisfy this property.

1 Idea of synthesis

As mentioned earlier, the objective of synthesis is to design a model which is guaranteed to satisfy our desired property. Figure 2 shows that given a property ϕ over sets of inputs

and outputs, a model P needs to be designed such that it satisfies it. However, in model checking only the correctness of an already designed model is investigated. Therefore, the advantage of synthesis is that no verification is required (it is correct by the design), and also no re-design is necessary.

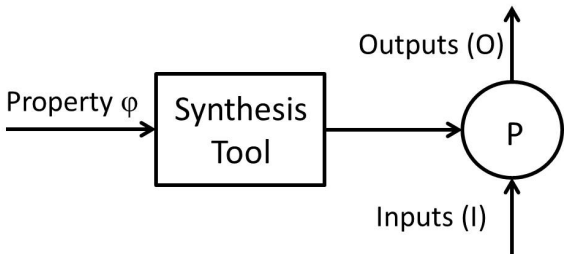


Figure 2: Synthesizing model P given property ϕ.

1.1 Synthesis overview

In order to determine the feasibility of synthesis in general, Church's problem is defined. This problem includes the following two statements,

- Realizability: Existence of a finite state system that realizes behavior ϕ.
- Synthesis: Construction of the system if specifications are realizable.

In the case, that the specification are feasible to synthesize, we try to find a winning strategy (f). Winning strategy is defined to be the strategy which satisfies the given specifications over the sets of inputs and outputs.

- Winning strategy; $f : (2^I)^* \rightarrow 2^O$
- Specifications (ϕ); statement in linear temporal logic over $I \cup O$

where I and O respectively denotes inputs and outputs of the model. Additionally, $*$ denotes that the input set is finite.

1.2 History

A brief history of synthesizing models for a given specification is itemized in chronological order.

- Church (1957): He defines the realizability problem initially for circuits. This problem is know as Church's problem.

- Büchi and Landweber (1969) proved the following statements.

  - Realizability is decidable.
  - If a winning strategy exists, then it is finite state.
  - Game view of synthesis (two-player game): Assuming environment is the first player (I) and system is the second player (O). In this case, if system wins, it is realizable, and if environment wins, it is unrealizable.

- Rabin (1972): Simpler solution for game view of Synthesis proposed by Büchi and Landweber problem.

- Pnueli and Rosner (1989): Rosner proposed linear temporal logic, and they suggested synthesis using linear temporal logic specification (2 exponential time-complex).

- Piterman, Pnueli, Sa'ar (2006): GR(1) synthesis (synthesis using the GR(1) fragment of LTL) is not two exponential time-complex. (translation from specification to automaton is polynomial time, see Section 3).

As Büchi proposed, the synthesis problem can be considered as a two-player game, where environment plays first (input i_k) and the system responds to its action (output o_k). Given a property ϕ, the system is called satisfiable if for some given input (environment action), there exists an output (system action) such that property ϕ is satisfied. In other words,

$$\exists \hat{i} = i_0, i_1, i_2, .. \tag{1}$$
$$\exists \hat{o} = o_0, o_1, o_2, .. \tag{2}$$
$$s.t.\ P \models \hat{i} \cup \hat{o} = \phi. \tag{3}$$

Satisfiablity is considered to be a subset of realizability. Because in realizability, the winning strategy should exist for all possible inputs rather than some given set of inputs. Therefore, realizability can be defined as;

$$\forall \hat{i} = i_0, i_1, i_2, .. \tag{4}$$
$$\exists \hat{o} = o_0, o_1, o_2, .. \tag{5}$$
$$s.t.\ P \models \hat{i} \cup \hat{o} = \phi. \tag{6}$$

In the next section, an example on satisfiability and realizability is elaborated.

1.3 Example

Consider a coffee machine which can grind or brew coffee. The input to this system is pressing its button. Grinding or brewing coffee will be the output of this system.

$$I = \{button\},\ O = \{grind, brew\}. \tag{7}$$

We assume that a cup of coffee is made after two time steps of grinding and three time steps of brewing. Therefore, our desired LTL specifications for having one cup of coffee can be written as;

$$G(button \rightarrow ((grind, \neg brew), X(grind, \neg brew), XX(\neg grind, brew), \\ XXX(\neg grind, brew), XXXX(\neg grind, brew))) \tag{8}$$

where X represents the next time step. The model is defined using triple P, where value '1' represents presence of the corresponding input or output, and value '0' represents its absence. The following model is for getting a cup of coffee when the button is pressed only once.

$$P = \left\{ \begin{array}{c} Input : button \\ Output : grind \\ Output : brew \end{array} \right\} = \left\{ \begin{array}{c} 0 \\ 0 \\ 0 \end{array} \right\}, \left\{ \begin{array}{c} 1 \\ 1 \\ 0 \end{array} \right\}, \left\{ \begin{array}{c} 0 \\ 1 \\ 0 \end{array} \right\}, \left\{ \begin{array}{c} 0 \\ 0 \\ 1 \end{array} \right\}, \left\{ \begin{array}{c} 0 \\ 0 \\ 1 \end{array} \right\}, \left\{ \begin{array}{c} 0 \\ 0 \\ 1 \end{array} \right\} \tag{9}$$

In the first time step, there is no input or output present. In the second time step, someone presses the coffee machine button, and as a consequence it will grind coffee for two time steps and then brew it for another three time steps. Here, the button is only pressed in the second time step. Therefore, this system is satisfiable, since we are able to find a model P which satisfy the specifications. Now, let's consider the case that someone presses the coffee machine button in both the second and third time steps.

$$P = \left\{ \begin{array}{c} 0 \\ 0 \\ 0 \end{array} \right\}, \left\{ \begin{array}{c} 1 \\ 1 \\ 0 \end{array} \right\}, \left\{ \begin{array}{c} 1 \\ 1 \\ 0 \end{array} \right\}, \left\{ \begin{array}{c} 0 \\ ? \\ ? \end{array} \right\} \tag{10}$$

According to the specification given in 8, the input commands issued in the second and third time steps both ask to grind coffee. However, in the fourth time step, the input from the second time step asks for $(\neg grind, brew)$ and the input from the third time step asks for $(grind, \neg brew)$. Therefore, this specification is unrealizable. This example explains a system, which is satisfiable but not realizable for a given LTL specification.

2 LTL Synthesis

2.1 The General Synthesis Flow

Fig. 3 shows the general flow of synthesis from LTL specifications. The synthesis process is viewed as a 2-player *game* between the environment and the system. A game is constructed from some specification about the temporal behavior and a system with freedom (e.g. system dynamics), and then solved to get the synthesized system. During this process, the game construction is the most time-consuming part with doubly-exponential complexity in the general case, while the solving process requires polynomial time in general.

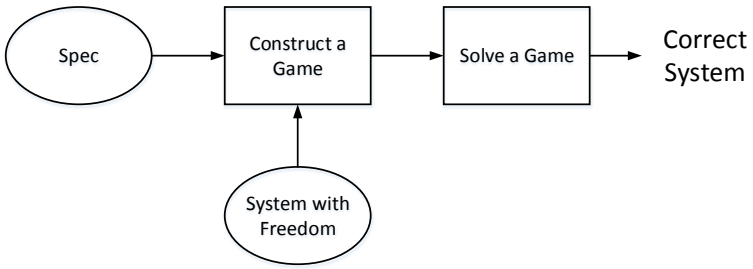


Figure 3: The general synthesis flow.

2.2 An Example of the Synthesis Process

Fig. 4 shows how to synthesize a simple LTL formula $\mathrm{G}(r \rightarrow \mathrm{X}g)$ describing a safety property into a state machine.

First, the safety property is transformed into a word automaton, which is then turned into a safety game. In the safety game, one player is the *system player*, while the other is the *environment player*. The environment player plays a move at the states represented by circles (q_0 and q_1 in this case), while the system player plays a move at the states represented by black dots.

In this example, at state q_0, no matter the environment player gives r or $\neg r$, the system player can choose any move (either g or $\neg g$, denoted as "*") to avoid losing the game. At state q_1, the system player needs to play g in either case, otherwise she will lose the game (denoted as $\perp$ in the graph). If a player has a *winning strategy*, then she can win the game by enforcing that strategy. The winning strategy of the system player corresponds to the Mealy machine that satisfies the specified temporal property.

A more general LTL formula such as $\mathrm{GF}p$ may result in a non-deterministic Büchi automaton. The aforementioned "automaton to game" construction only works for deterministic

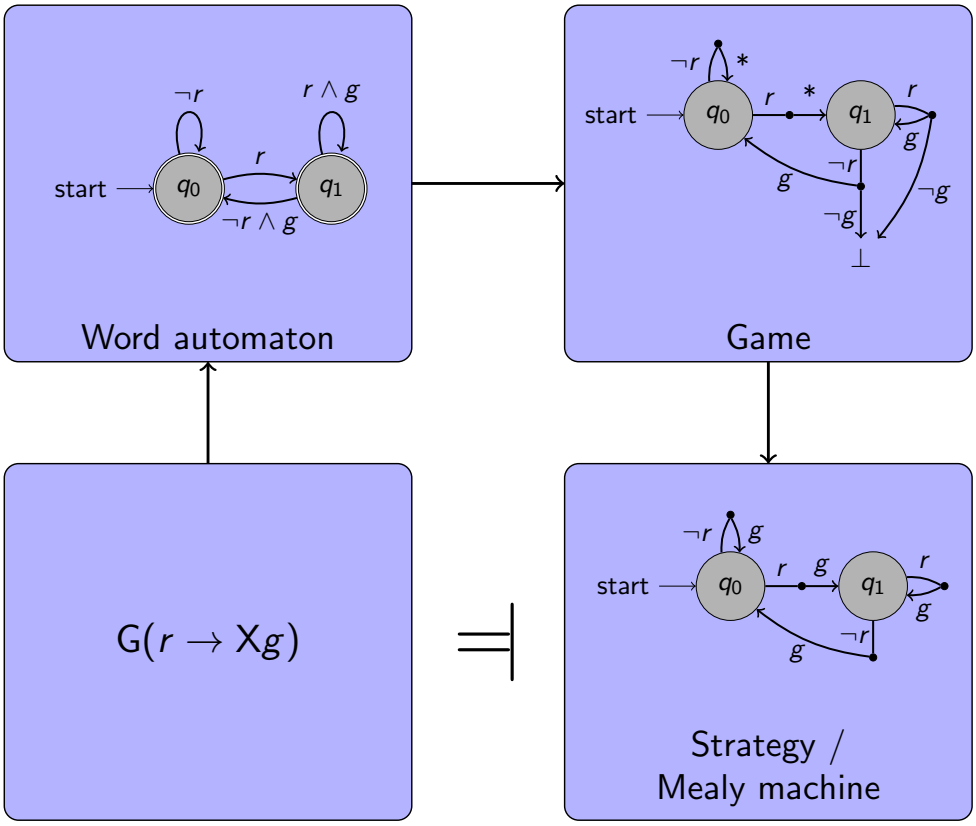


Figure 4: An example illustrating the general synthesis flow.

automata. As a result, richer automaton models such as parity automaton and game winning conditions are needed.

The critiques on LTL are generally focused on two aspects. First, the doubly-exponential complexity of synthesis algorithm. Second, it is often the case that designers do not have the correct and complete specification at the very beginning and need to iterate their design until satisfaction. As for the complexity issue, doubly-exponential is only the worst case complexity, for some subsets of LTL formulas, such as the GR(1) to be introduced shortly, the complexity can be much lower. Regarding the second issue, incremental synthesis and compositional synthesis may help.

Incremental synthesis can help reduce the runtime of the tool when there is a slight modification in the design specification. This can be particularly useful in the presence of time-varying environment assumption, and when a specification is unrealizable.

In compositional synthesis, some previously verified components can be directly reused in

the synthesis process, which can reduce the complexity of the synthesis process.

3 GR(1) Synthesis

3.1 How is GR(1) different from LTL?

GR(1) stands for the Generalized Reactivity formulas, which is actually a subset of LTL that can be solved in polynomial time, to be more specific $O(N^3)$ where N is the size of the automaton. The specifications we consider here are of the form $\phi = \phi_e \rightarrow \phi_s$. We require that ϕ_α for $\alpha \in \{e, s\}$ can be rewritten as the conjunction of the following parts.[7]

- ϕ_i^α is a Boolean formula characterizing the initial states of the implementation.
- ϕ_t^α is a Boolean formula characterizing the transition relation of the implementation.
- ϕ_g^α is a Boolean formula characterizing the goal states of the implementation.

It turns out that most practical specifications can be written in the above format.

4 Synthesis Tools

A number of LTL synthesis tools are available in academia, such as TuLiP[8], LTLMoP[6], RATSY[1], Acacia+[2] and Lily[5]. There are also some translation tools to transform LTL formulas into automata, such as Rabinizer [3] and LTL2BA[4].

References

[1] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. Ratsy–a new requirements analysis tool with synthesis. In *Computer Aided Verification*, pages 425–429. Springer, 2010.

[2] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for ltl synthesis. In *Computer Aided Verification*, pages 652–657. Springer, 2012.

[3] Andreas Gaiser, Jan Křetínskỳ, and Javier Esparza. Rabinizer: Small deterministic automata for ltl (f, g). In *Automated Technology for Verification and Analysis*, pages 72–76. Springer, 2012.

[4] Paul Gastin and D Oddoux. Ltl 2 ba: fast translation from ltl formulae to büchi automata, 2001.

[5] Barbara Jobstmann and Roderick Bloem. Optimizations for ltl synthesis. In *Formal Methods in Computer Aided Design, 2006. FMCAD'06*, pages 117–124. IEEE, 2006.

[6] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *Robotics, IEEE Transactions on*, 25(6):1370–1381, 2009.

[7] Nir Piterman, Amir Pnueli, and Yaniv Saar. Synthesis of reactive (1) designs. In *Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer, 2006.

[8] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M Murray. Tulip: a software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*, pages 313–314. ACM, 2011.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 4/15: Simulation, Bisimulation, and Termination

Scribe: Michael McCoyd *Editor: Daniel Fremont*

This lecture covers two unrelated topics:

- Simulation and Bi-Simulation [3]
- Verifying Termination

1 Simulation and Bi-Simulation

We have previously described the verification problem as taking a system, modeled as a Kripke structure, and a property, expressed in some temporal logic, and checking whether the system satisfies the property.

It is also useful for the specification to itself be a Kripke structure or other system description, so that we have two systems:

M_1 the implementation system, and

M_2 the specification system.

How do we formulate whether M_1 satisfies the specification in this case? There are several possibilities, which we first describe informally.

Trace equivalence The observable behavior of the systems, i.e. their sets of traces, must be identical:

$$\mathcal{L}(M_1) = \mathcal{L}(M_2).$$

This is like program equivalence, where all that matters is the input-output behavior, not what goes on internally. Trace equivalence is often too strict a notion of satisfying the "specification" M_2, since it requires that every possible behavior of M_2 also be a behavior of M_1. We can weaken it to:

Trace containment This requires that every possible behavior of M_1 be a behavior of M_2, but not vice versa. The spec can have more behaviors than the implementation, but the implementation always stays within the behaviors allowed by the spec:

$$\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2).$$

Simulation We can think of this as a game. In each round M_1 always goes first, M_2 follows M_1, and M_2 tries to produce the same output behavior as M_1. M_2 simulates M_1 if it can always match M_1's behavior, no matter what transitions M_1 makes:

"M_2 can match every step of M_1"

Bi-simulation The general idea is to use the same type of game as in simulation, but now in each turn we allow either player to go first. Despite the name, having a bi-simulation between M_1 and M_2 is *strictly stronger* than having simulations in both directions (we'll see an example later). We say M_1 and M_2 are bi-similar, M_1 bi-simulates M_2, or M_2 bi-simulates M_1, when:

"If M_1 steps, M_2 can match it, and
if M_2 steps, M_1 can match it."

Although we've been thinking of M_2 as a specification, all of these notions make sense when M_2 is another implementation. We could start with a highly abstract version of some program as M_2, and then add details to obtain M_1. For example we might want our program to output nonnegative integers, which could be expressed in LTL as $G(y \geq 0)$ if the output variable is y. We can model this as a program M_2 which makes a completely non-deterministic update to y, under the constraint that y is nonnegative. We could refine this with a deterministic program M_1 that counts up from zero.

$$G(y \geq 0)$$

$$M_2 : \boxed{y' \leftarrow \mathit{choose}(\mathbb{Z}^+)} \xrightarrow{\text{y}}$$

$$M_1 : \boxed{\begin{array}{ll} \text{initialize:} & y = 0 \\ \text{transition:} & y' = y + 1 \end{array}} \xrightarrow{\text{y}}$$

We can see that $\mathcal{L}(M_1) \subset \mathcal{L}(M_2)$, as M_1 can output only the sequence $\langle 0, 1, 2, \dots \rangle$, while M_2 can output any sequence of nonnegative integers. So M_1 would satisfy the "specification" M_2 with respect to trace containment, but not trace equivalence.

1.1 An Example

Consider the systems M_1 and M_2 shown in Figure 1:

$$M_1 = (S_1, s_{01}, R_1, L_1) \qquad L_1 : S_1 \to 2^{AP}$$
$$M_2 = (S_2, s_{02}, R_2, L_2) \qquad L_2 : S_2 \to 2^{AP}.$$

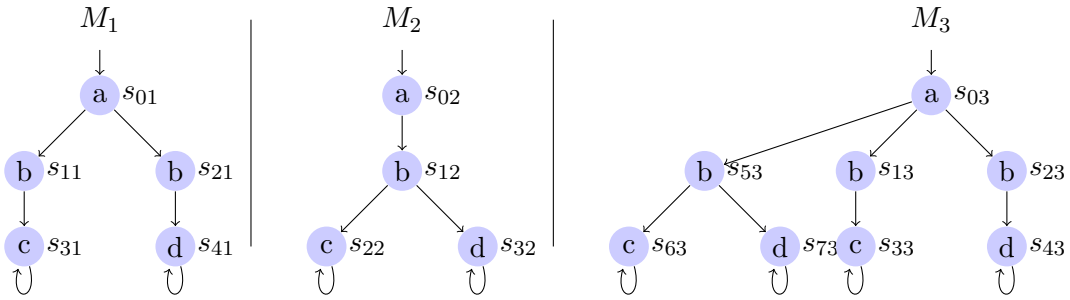


Figure 1: Systems M_1, M_2, and M_3 with labels a, b, c, and d.

The only difference between M_1 and M_2 is that the nondeterministic choice happens in the initial state in M_1, but in the second state in M_2. Let's see how these systems compare under the equivalence/refinement notions we've discussed above.

M_1 and M_2 are trace equivalent M_1 and M_2 have the same set of traces:

$$\mathcal{L}(M_1) = \{abc^\omega, abd^\omega\} = \mathcal{L}(M_2).$$

M_2 simulates M_1 If M_1 starts executing, it makes a choice, say to the left, outputting 'ab'. M_2 can match this by going down, matching with 'ab'. M_1 then goes down outputting 'c'. M_2 has two choices, and given that M_1 output 'c', it matches this by going left and outputting 'c'. From then on both systems output 'c' without changing state. Similarly if M_1 goes right initially. Thus no matter how M_1 chooses to step, M_2 can match it, so M_2 simulates M_1.

M_1 does not simulate M_2 M_2 starts by going down, outputting 'ab'. M_1 must make a choice between left or right, which forces all subsequent outputs to be either 'c' or 'd'. If M_2 then makes the opposite choice, outputting a 'c' if M_1 went right and a 'd' if M_1 went left, then M_1 cannot match it.

1.2 Simulation

To define simulation more formally we need the concept of a simulation relation. A relation $H \subseteq S_2 \times S_1$ is a **simulation relation** between M_2 and M_1 (note the order) if:

1. $(s_{02}, s_{01}) \in H$; (the initial states correspond)

2. for all $(s_2, s_1) \in H$, $L_1(s_1) = L_2(s_2)$; (corresponding states have the same labels)

3. for all $(s_2, s_1) \in H$ and $s_1' \in S_1$ such that $R_2(s_1, s_1')$, there is some $s_2' \in S_2$ such that:

   (a) $R_2(s_2, s_2')$,
   (b) $(s_2', s_1') \in H$.

   (every transition in M_1 has a corresponding one in M_2)

The idea, depicted in Figure 2, is that if the simulation relation H relates s_2 and s_1, then if M_1 makes a move s_1 to s_1', M_2 can match that with a move from s_2 to s_2' such that the destination states s_2' and s_1' are related by H.

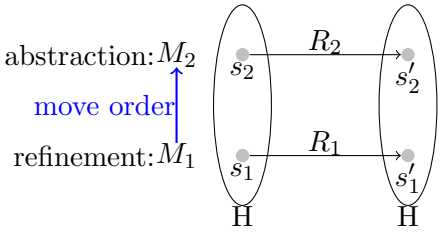


Figure 2: A simulation relation.

If a simulation relation between M_2 and M_1 exists, then M_2 simulates M_1. We also say that M_1 *refines* M_2.

A practical use of simulation is in compositional verification or design. There you start out with an abstract description of a component, and would like to show that your automatic synthesis tool always produces a refinement of this abstract description. If you can prove this for your transformation rules, you get a guarantee that the procedure does not add new behaviors to the system.

Simulation is more difficult to understand than language equivalence or containment, which are more intuitive. Understanding the connection between them can help.

Theorem 1.1 *If M_2 simulates M_1 then $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$.*

So if we are able to show simulation, you get language containment for free. This can be proved by induction on the number of steps in the trace of the system: see for example [1]. This theorem connects the simulation game to language containment, which is closer to our intuitive notion of correctness (restricting the behaviors of the system to be a subset of those allowed by the spec).

Do we have the other direction, i.e. if you have language containment do you have simulation? We showed that M_1 and M_2 in Figure 1 have the same language but simulation only

works one way. So simulation implies language containment, but language containment does not imply simulation.

Simulation connects to model checking by the following theorem:

Theorem 1.2 *If M_2 simulates M_1, then any property in LTL/ACTL* satisfied by M_2 is also satisfied by M_1.*

ACTL* is the subset of CTL* whose properties start with the operator A, e.g. A(FG p).

Proof: We prove the theorem for LTL properties. By Theorem 1.1, the set of traces of M_1 is contained in the set of traces of M_2, so every trace of M_1 is also a trace of M_2. Now if an LTL property Φ is satisfied by M_2, then every trace of M_2 satisfies Φ, and in particular the traces of M_1 satisfy Φ. So M_1 also satisfies Φ. ∎

A use of this is in the compositional design methodology where you start with an abstract transition system and keep refining it. Even if your refinements get fairly complicated, as long as your refinement rules preserve simulation you only need to prove the desired temporal properties on the original, most abstract specification.

1.3 Bisimulation

The definition of a bisimulation relation is very similar to that of a simulation relation, just adding one further condition. A relation $H \subseteq S_2 \times S_1$ is a **bisimulation relation** between M_2 and M_1 if:

1. $(s_{02}, s_{01}) \in H$; (the initial states correspond)
2. for all $(s_2, s_1) \in H$, $L_1(s_1) = L_2(s_2)$; (corresponding states have the same labels)
3. for all $(s_2, s_1) \in H$ and $s_1' \in S_1$ such that $R_2(s_1, s_1')$, there is some $s_2' \in S_2$ such that:
   (a) $R_2(s_2, s_2')$,
   (b) $(s_2', s_1') \in H$.

   (every transition in M_1 has a corresponding one in M_2)
4. for all $(s_2, s_1) \in H$ and $s_2' \in S_2$ such that $R_2(s_2, s_2')$, there is some $s_1' \in S_1$ such that:
   (a) $R_1(s_1, s_1')$,
   (b) $(s_2', s_1') \in H$.

   (every transition in M_2 has a corresponding one in M_1)

The picture for bisimulation, Figure 3, is very much the same as before. But now it goes both ways in terms of how we reason about the diagram. As before, if M_1 makes a step, M_2 can match it and the initial and final states are related by H. But now, if instead M_2 goes first, M_1 can also match it.

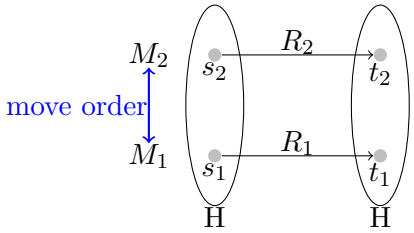


Figure 3: A bisimulation relation.

Returning to our examples in Figure 1, we saw earlier that M_2 could simulate M_1, but since M_1 cannot simulate M_2, this cannot not be a bisimulation. If we expand M_1 into the system M_3, this can simulate M_2. On the first move M_2 goes to s_{12}, and M_3 can choose to go to s_{53}, from which the simulation is clear. There are starting moves that M_3 could make that would prevent it from simulating M_1 or M_2. But there are some that do allow it, and the definition is that there exists a strategy that works — it is not necessary that all strategies work.

It is also easy to see that M_2 can simulate M_3. However, they are not bisimilar. The key is that the ordering of who steps first can change during the bisimulation game. Suppose on the first turn M_3 moves to s_{13}, to which M_2 must respond by moving to s_{12}. If M_2 goes first on the second turn, moving to the right and outputting 'd', M_3 cannot respond: it can only output a 'c'. This shows that it is possible for two systems to simulate each other but not be bisimilar.

While simulation implies trace containment, bisimulation implies trace equivalence:

Theorem 1.3 *If M_1 and M_2 are bi-similar, then $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.*

As we've seen in our earlier examples, the converse does not hold.

Bisimulation also implies a stronger result than simulation about preservation of temporal properties:

Theorem 1.4 *If M_1 and M_2 are bisimilar, then for any CTL* property Φ, M_1 satisfies Φ if and only if M_2 does.*

1.4 Computing simulation relations

Say we want to prove M_2 simulates M_1, perhaps so that we can check properties of interest on the simpler abstract model. Here is a basic procedure for automatically finding a simulation relation between M_2 and M_1:

1. Initialize $H = S_2 \times S_1$, i.e. put all possible pairs into H.
2. Iteratively prune pairs $(s_2, s_1) \in H$ that do not satisfy conditions 2 and 3 in the definition of a simulation relation. Continue until a fixed point is reached.
3. If the initial state pair (s_{02}, s_{01}) is still in H, then H is a simulation relation between M_2 and M_1. Otherwise, M_2 does not simulate M_1.

This is a very general procedure, covering any systems that can be modeled as Kripke structures, but it can be computationally expensive (the space $S_2 \times S_1$ could be huge, and many passes in step 2 could be required before reaching a fixed point). There is also a SAT-based formulation, but it is not as useful in practice.

2 Termination

We discuss the problem of termination testing at an intuitive level. This problem is clearly undecidable in general, being the famous halting problem, but in many cases it is tractable.

So far we have deliberately looked at non-terminating systems, i.e. systems which produce infinite traces, like Kripke structures. If a Kripke structure is finite state we can represent termination by adding a new "terminal" state with a self-loop, and transitioning to this state whenever the system is supposed to terminate. Then checking termination is simply checking whether the system eventually reaches this terminal state. However, if the model is not finite state, we need other strategies. We will focus on the case of programs.

2.1 The deductive approach to termination testing

We have noted several times in this class that many verification problems boil down to synthesizing an artifact. In the case of proving termination, we need to synthesize a **ranking function**, which somehow measures how far the program is from terminating. To prove non-termination, we find a **recurrence set**, a set of states from which the program cannot leave.

We will focus on ranking functions, beginning with an example.

```
0  while (x>0)       //  x is an unsigned integer variable
1    x = x - 1;
2  skip
```

Let S be the set of all possible states of this program, including which line we are on and the value of x. Observe that when $x = 0$, the program terminates, and otherwise x strictly decreases with every step of the program. Thus the function F which maps a state $s \in S$ to the value it assigns to x is a measure of how far from termination the program is: if the program terminates in state s then $F(s) = 0$; otherwise, s transitions to a new state s', and we have $F(s') < F(s)$. Since F continually decreases over time and the program terminates when it hits zero, the existence of F proves that the program must always terminate.

The function F is a simple example of a *ranking function*, a notion which goes back to Alan Turing [5]. Before we can define a ranking function precisely, we need a couple of definitions.

Definition 2.1 *A* well-founded relation *on S is a binary relation $R \subseteq S \times S$ such that there is no infinite sequence $s_0, s_1, s_2, \ldots$ of elements of S where $R(s_i, s_{i+1})$ for all $i \in \mathbb{N}$.*

For example, the relation $>$ over the natural numbers is well-founded: you cannot have an infinite sequence of natural numbers each of whom is smaller than the previous one. On the other hand, the relation $>$ over the integers is not well-founded, as shown for example by the infinite sequence $1 > 0 > -1 > -2 > \ldots$.

If x is the current value of the variable in our example program, and x' is its value after one step, then the relation $R(x, x') = x > x' \wedge x' > 0$ is well-founded. The relation cannot hold for all consecutive pairs in an infinite sequence, since the first condition of the conjunction forces the value of x to decrease, while the second prevents it from becoming zero. This can only happen in sequences of finite length.

Definition 2.2 *A* well-ordered set *$(D, <)$ consists of a set D and a well-founded relation $<$ on D which totally orders D (i.e. for any $a, b \in D$, either $a = b$, $a < b$, or $b < a$).*

For example, the set of natural numbers ordered by $<$ is a well-ordered set.

Definition 2.3 *A* ranking function *for a program with states S and transition relation R is a map $F : S \to D$ for some well-ordered set $(D, <)$ such that for all states $s, s' \in S$ where $R(s, s')$, we have $F(s') < F(s)$.*

In our earlier example of a ranking function F, the well-ordered set used is simply the natural numbers under the usual $<$ order. This is the most intuitive kind of ranking function: a

function mapping states to nonnegative integers which strictly decrease over time and so must eventually bottom out in a terminating state. But in general a ranking function is allowed to use any well-ordered set, which can be convenient for complex programs.

If a program has a ranking function F, it is guaranteed to terminate, since F maps the sequence of states reached by the program to a sequence in D which is decreasing according to the order $<$. Since by definition $<$ is well-founded, any decreasing sequence must be finite, and so the program must reach a state from which no more transitions are possible: it terminates. Conversely, if a program always terminates it must have a ranking function: for example, the function which maps a state to the longest possible number of steps remaining before termination. Of course, when trying to prove termination this function is unknown.

Notice that a program always terminating is the same as its transition relation being well-founded (when restricted to reachable states): the latter means that we cannot find an infinite sequence of transitions to take, which is how non-termination occurs. In practice, algorithms for proving termination work by trying to show the transition relation is well-founded, rather than explicitly working with ranking functions.

Recall that when we verified safety properties, such as $G(\phi)$, we could do a proof by induction by showing

1. $I(s) \rightarrow \phi(s)$, and
2. $\phi(s) \wedge R(s, s') \rightarrow \phi(s')$.

Often we would not be able to prove (2), because the property ϕ is inductive over reachable states but not all states. So we had to synthesize an auxiliary invariant to strengthen the inductive hypothesis enough to allow the proof to go through. We can use a similar procedure for proving well-foundedness of the transition relation and thus termination. Specifically, we try to find a transition invariant T such that we can prove:

1. $I(s) \wedge R(s, s') \rightarrow T(s, s')$
2. $T(s, s') \wedge R(s', s'') \rightarrow T(s', s'')$
3. T is well-founded

Conditions (1) and (2) ensure that T holds for all reachable transitions, and thus condition (3) means that R restricted to reachable states must be well-founded.

This is the classic approach to deductively proving termination. However, it has proved difficult to find a single well-founded relation proving termination of a program, as even for programs of modest size the relation may be extremely complex. The big advance made around 2004 (see [4, 2]) was the notion of "disjunctively well-founded relations". Here you

use a relation which is built up as a union of several well-founded relations, but which isn't necessarily well-founded itself. The advantage of this approach is that often each component relation can be quite simple, and synthesized independently from the other components. In practice this has allowed much more complicated software to be verified, for example actual device drivers.

References

[1] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[2] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.

[3] Raffaella Gentilini, Carla Piazza, and Alberto Policriti. From bisimulation to simulation: Coarsest partition problems. *Journal of Automated Reasoning*, 31(1):73–103, 2003.

[4] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proceedings of LICS*, pages 32–41. IEEE, 2004.

[5] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, 1949.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: Alexandre Donzé*

Lecture 4/22: Simulation-based Verification of Cyber-Physical Systems

Scribe: Julie Newcomb *Editor: Negar Mehr*

The motivating example for this lecture is designing a grading CPS model. Ordinarily, students write their control algorithms in LabVIEW or C code, then run their code on robots for TAs to grade. When the course was run on a MOOC, grading had to be done in simulation since students don't have access to robots; even if they did, TAs couldn't possibly inspect them all. Therefore, formal methods are needed to verify the correctness of students' designs. The aim is to check whether the design satisfies the assigned specification, and if not, what feedback can be given to the students to help fix their mistake. The first goal is handled through model checking and the second through providing counterexamples.

In classical model checking, the model is a finite state machine(FSM) and the specification is an LTL formula. In cyber-physical systems, the model is a hybrid system and the specification is in STL (signal temporal logic). Although hybrid systems can in theory be translated into FSM, in practice this doesn't scale.

Model checking STL formulas is intractable on complex cyber-physical systems. Finding a faithful abstraction that is small enough to work with is very difficult. Instead, CPS-Grader [4] uses simulation and monitoring. This can be thought of as an incomplete depth first search over the tool's behavior. This technique can find counterexamples or achieve some degree of confidence that the system works, but it can't prove that the system works in all circumstances.

1 Signal Temporal Logic

STL is an extension of LTL with real time and real value constraints. This table compares Linear Temporal Logic (LTL), Metric Temporal Logic (MTL), and Signal Temporal Logic (STL) for a simple request-grant property [1]:

LTL	$G(r \implies Fg)$	Boolean predicates, discrete-time
MTL	$G(r \implies F_{[0,5s]}g)$	Boolean predicates, real-time
STL	$G(x[t] > 0 \implies F_{[0,5s]}y[t] > 0)$	Predicates over real values, real-time

Signals are functions from $\mathbb{R}^+$ to $\mathbb{R}$. We denote the value of x at time t as $x[t]$.

Note: Atomic predicates in STL are inequalities over signal values at symbolic time t. (Since we are dealing with continuous values in STL, it doesn't make sense to define equalities; we

can approximate equalities by placing values between some tight bounds).

Remark: Temporal operators F, G and U are defined with time intervals. There is no next operator since "next" makes no sense without discrete time intervals.

When no time interval is defined in a STL formula ϕ, it is considered true if it is true at time 0. A subformula ψ is evaluated on future values depending on temporal operators.

Example:

- $\phi = G(x[t] < 0.5)$ is true iff $x[t] < 0.5$ is true when t replaced by 0.
- $\phi = F_{[0,3.5]}(x[t] > 0.5)$ is true iff $x[t] > 0.5$ is true when t is replaced by some time in [0,3.5].
- $\phi = G_{[0,3.5]}(\psi)$ is true iff ψ true at all times in [0,3.5].
- $\phi = G_{[2,6]}(|x[t]| < 2)$ is true iff the absolute value of $x[t]$ is less than 2 for all times in between 2 and 6.
- $\phi = G(x[t] > 0.5 \rightarrow F_{[0,.6]}(G_{[0,1.5]}x[t] < 0.5))$ It is always the case that after $x[t] > 0.5$, in less than 0.6 seconds, x settles under 0.5 for 1.5 seconds.

2 Monitoring

CPSGrader checks students' designs using STL monitoring. Below is an example of monitoring a trace $w = aaabbaaa$ over several LTL formulas. Each step is evaluated with respect to the future sequences of the trace.

	a	a	a	b	b	a	a	a
a	true	true	true	false	false	true	true	true
b	false	false	false	true	true	false	false	false
Xb	false	false	true	true	false	false	false	?
Ga	false	false	false	false	false	true?	true?	true?
Fb	true	true	true	true	true	false?	false?	false?

This method has a limitation; we don't always have the data to evaluate the trace on, which is the reason for existence of missing values on the table above. If we receive new data later, we will recalculate the past.

Note: Monitoring is easy for discrete time signals as efficient algorithms exist for dense time-signals [3].

In the motivation example presented previously, grading is based on test traces and fault monitoring. We gather system traces in the desired environment and write fault properties using STL to characterize faulty behaviors on the trace. In the algorithm defined below, traces are obtained from the simulator in the necessary environment; each trace is checked for the relevant faults. If faults are detected, CPSGrader will print out feedback and possibly return early if the detected fault is critical.

for each test trace **do**
  Get trace x from simulator
  for each fault with STL formula ϕ **do**
    Check whether $x \vDash \phi$
    Print feedback
    if fault is critical **then**
      return
    end if
  end for
end for

Overall, this approach works well for grading or for detecting if behavior fails in a particular aspect.

3 Falsification using quantitative semantics

Imagine we want to find inputs to an automatic transmission system that will falsify some pre-defined properties Such a simulation-based search has practical applications in the automotive industry.

Given a system

$u(t) \rightarrow SystemS \rightarrow S(u(t))$

we want to find some input signal where a property ϕ is not satisfied, or

$u \in U$ s.t. $S(u(t)) \nvDash \phi$

This is done with a quantitative satisfaction function ρ [2] such that:

$\rho^{\phi}(x,t) > 0 \implies x,t \vDash \phi$

$\rho^{\phi}(x,t) < 0 \implies x,t \nvDash \phi$

This can be interpreted as a robust measure that indicates how close or far we are to satisfaction. As defined above, the sign of the function's return-value indicates the satisfaction status, and its absolute value indicates tolerance. The value of ρ for ϕ is calculated by bottom-up of each subformula ψ of ϕ. The boolean operators for subformulas are calcu-

lated by taking the minimum of function values on the two subformulas for conjunction and the maximum of the function values on the two subformulas for disjunction. For temporal operators, G is conjunction on the time axis, while F is disjunction on the time axis.

However, it is non-trivial to compute this function for arbitrary formulas; the size of the robustness signal can grow exponentially with formula height.

Inputs are parameterized to restrict inputs to a subset of the input space. Thus, the falsification process can be written as:

$\rho_u \rightarrow SystemS \rightarrow x(t) \rightarrow$STL monitor$\phi \rightarrow \rho^{\phi}(x,t) < 0$

Finding a counter-example of the above form reduces to solving

$\rho* = \min_{\rho_u \in P_u} \rho^{\phi}(x,0)$ if $\rho* < 0$

Other topics in this field include online monitoring, parameter synthesis, requirements mining, controller synthesis with STL specs (BluSTL).

References

[1] Cps grader: A tutorial. `http://verifun.eecs.berkeley.edu/cpsgrader/getting-started/CPSGrader_tutorial.pdf`. Accessed: 28 April 2015.

[2] Efficient robust monitoring for stl. `http://www-verimag.imag.fr/~maler/Papers/slides-efficient-robust.pdf`. Accessed: 28 April 2015.

[3] Alexandre Donzé, Oded Maler, Ezio Bartocci, Dejan Nickovic, Radu Grosu, and Scott Smolka. On temporal logic and signal processing. In *Automated Technology for Verification and Analysis*, pages 92–106. Springer, 2012.

[4] Garvit Juniwal, Alexandre Donzé, Jeff C Jensen, and Sanjit A Seshia. Cpsgrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory. In *Embedded Software (EMSOFT), 2014 International Conference on*, pages 1–10. IEEE, 2014.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: M. N. Rabe*

Lecture 4/27: Hyperproperties

Scribe: Andrew Head *Editor: Benjamin Caulfield*

Hyperproperties are a formalization of system properties that are more powerful than trace properties. They are particularly useful in describing computer security policies.

In these notes, we review trace properties and limits to their expressivity. We provide a brief introduction to typical information flow security properties. Then we produce the definition of hyperproperties, and an accompanying specification language, HyperLTL, that can be used to describe hyperproperties for security applications.

1 Trace Properties Review

Systems generate behaviors, which we can define as sequences of a system's atomic propositions evaluated over its full execution. Therefore, a trace is an element of $(2^{AP})^\omega$.

In the past, we studied *trace properties*, which we simply called *properties*. A trace property is an acceptable subset of all possible behaviors of a system: $T \subseteq (2^{AP})^\omega$

We say that a system S satisfies trace property T if and only if all of its traces belong to the trace property:

$$S \models T \iff Traces(S) \subseteq T \tag{1}$$

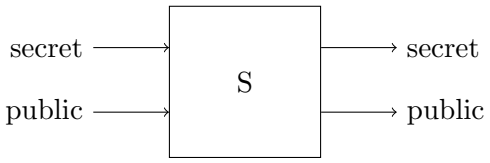
We can use trace properties to verify system behavior. We can describe some trace properties by invariants or linear temporal logic (LTL), for example:

- *Invariant*: $x < 5$ for all time
- *LTL*: $G(x < 5)$
- *CTL / CTL**: $AG(x < 5)$

Remark: While LTL can express some trace properties, they cannot express *all* trace properties. For example, it is impossible to express that a proposition holds true on all even-numbered cycles with no restriction on odd-numbered steps. Similarly, we will see that HyperLTL cannot be used to express all possible hyperproperties.

2 Introduction to Information Flow Security Policies

Hyperproperties are apt at describing security policies for systems involving a flow of information. For our execution model, we consider a system S with a public and secret input, as well as a public and secret output.



When checking the security of a system, our primary question is, *Does the output contain some way to find the secret input?* We assume an attacker can attempt varying the input to learn the secret, and that he has access to the program code and protocol specification. We want to avoid any use of the secret input in the public output.

Example: We show 3 algorithms of varying levels of security. Algorithm 1 leaks the secret by writing it to the public output. Algorithm 2 does not leak the secret, as the secret does not influence the output. Algorithm 3 shows a subtle case, where an attacker can infer *something* about the input secret from the output, even if she cannot learn its exact value.

Algorithm 1 This program leaks the secret by writing it directly to the public output.

```
x = read(secret)
write(public, x) {write output}
```

Algorithm 2 This program does not leak the secret — only the public input is written to public output via the temporary variable y.

```
x = read(secret)
y = read(public)
x = x + y
write(public, y)
```

Algorithm 3 If she knows the source code, an attacker for this program can find whether the secret is less than 3 by observing the public output.

```
x = read(secret)
y = read(public)
if x < 3 then
  write(public, y)
end if
```

Typically, information flow security has been verified by syntactic checks. Some variables are assigned a type of *secret*, and some analysis of the code determines how these values are used and whether they influence the public output. In this lecture, we consider how *hyperproperties* can be used to verify the security of information flows.

2.1 Key information flow properties

The following are information flow properties we might want to verify for system S:

- *Observational determinism (OD).*

$$\forall t, t' \in Traces(S) \quad t \underset{I_{public}}{=} t' \implies t \underset{O_{public}}{=} t' \tag{2}$$

  This is also called *non-interference*, which is an overloaded term in computer security.

- *Non-inference*: there is a diversity of traces that have different secrets but the same input-output behavior. In other words, you can't deduce the secret from knowing the input, output, and code.

$$\forall t \in Traces(S)\, \exists t' \in Traces(S) \quad s.t. \quad t \underset{I_{public}}{=} t' \wedge t \underset{O_{public}}{=} t' \wedge t \underset{I_{secret}}{\neq} t' \tag{3}$$

- *Integrity*: we restrict the attacker's ability to affect data that should be off-access — an attacker cannot affect critical system data or state.

3 Hyperproperties

Hyperproperties are sets of trace properties [2]. Equivalently, they can be considered as *sets of sets* of traces:

$$H \in 2^{(2^{AP})^{\omega}} \tag{4}$$

We say a system S satisfies hyperproperty H if the set of all traces for S belongs to H:

$$S \models H \iff Traces(S) \in H \tag{5}$$

In this way, we can see hyperproperties as a list of legal systems, with each system specified as a set of all of its possible traces.

Hyperproperties can be used to describe error-correcting codes. They can also specify protocols such as mutual exclusion when we attempt to ensure that all agents can gain equal access to a resource. Some probabilistic properties can also be expressed as hyperproperties.

Any trace property can be expressed by an equivalent hyperproperty. Given trace property T, an equivalent trace property is expressed by the power set of T:

$$H_T = \{T' \mid T' \subseteq T\} \tag{6}$$

For H_T defined in this way, we can show that it is equivalent to T as follows:

$$S \models T \iff Traces(S) \subseteq T \iff Traces(S) \in H_T \iff S \models H_T \tag{7}$$

Example: OD can be defined as a hyperproperty.

First, we define the input and output as atomic propositions.

$$I, O \subseteq AP, \quad I \cap O = \emptyset \tag{8}$$

The public and secret propositions cannot intersect:

$$I_{public}, I_{secret} \subseteq AP, \quad I_{public} \cap I_{secret} = \emptyset \tag{9}$$

$$O_{public}, O_{secret} \subseteq AP, \quad O_{public} \cap O_{secret} = \emptyset \tag{10}$$

We define the hyperproperty as the set of all trace properties where all pairs of traces have equivalent output given the same input:

$$\{T \subseteq (2^{AP})^\omega \mid \forall t, t' \in T : t \underset{I_{public}}{=} t' \implies t \underset{O_{public}}{=} t'\} \tag{11}$$

where we define trace equality as:

$$t \underset{X_{public}}{=} t' \quad := \quad \forall i \in \mathbb{N}, \bigwedge_{a \in X_{public}} a \in t(i) \leftrightarrow a \in t'(i) \quad \circ \tag{12}$$

Note that although all trace properties can be represented by an equivalent hyperproperty, *not all hyperproperties can be represented by a trace property.*

Theorem 3.1 *OD cannot be represented by a trace property.*

Proof*: Assume for contradiction that $T \subseteq (2^{AP})^\omega$ is equivalent to OD. As $T \neq (2^{AP})^\omega$, there must be at least one trace $t \notin T$. Define a Kripke structure K_t trace t and no other traces. This structure models OD as all of its traces, which is a single trace, have equivalent input and ouput behavior. Therefore, T omits a model that observes OD and thus cannot be equivalent to OD.* ∎

Additionally, LTL cannot express OD. Given that LTL specifies a valid set of traces that satisfy a property — a trace property — and that a trace property cannot express OD (see Theorem 3.1), it is clear that there is no LTL formula equivalent to OD.

4 HyperLTL

We can show that neither LTL nor CTL* can represent hyperproperties such as OD. Now let us consider *HyperLTL*, a specification language that can express some valuable hyperproperties, including OD [1]. HyperLTL has a similar syntax to LTL, with the addition of *path quantifiers*.

$$\gamma ::= \forall\pi\gamma \mid a_\pi \mid \neg\gamma \mid \gamma \cup \gamma \mid \gamma \cap \gamma \mid G\gamma \mid F\gamma \mid X\gamma \mid \gamma U \gamma$$

A trace assignment function Π can satisfy hyperproperties specified by HyperLTL.

Definition 4.1 Π *is a* trace assignment *function,* $\Pi : V \to Traces$*, that maps names or variables to traces.*

We can define several of the base HyperLTL operators as follows:

$$\Pi \models a_\pi \iff a \in \Pi(\pi)(0) \tag{13}$$
$$\Pi \models \forall\pi\gamma \iff \forall t \in (2^{AP})^\omega \quad \Pi\left[\pi \to t\right] \models \gamma \tag{14}$$
$$\Pi \models X\gamma \iff \Pi_{+n} \models \gamma \tag{15}$$
$$\Pi \models \gamma_1 U \gamma_2 \iff \exists i \geq 0 : \Pi[i, \infty] \models \gamma_2 \cap \forall 0 \leq j < i\, \Pi[j, \infty] \models \gamma_1 \tag{16}$$

In the above syntax, a_π states that an atomic proposition a holds in the first state of path π. Furthermore, we define the function $\Pi\left[\pi \to t\right](\pi')$ as

$$\Pi\left[\pi \to t\right](\pi') = \begin{cases} t & if \quad \pi = \pi' \\ \Pi(\pi') & otherwise \end{cases}$$

Additionally, we define the notation Π_{+n} as

$$\Pi_{+n}(\pi)(i) = \Pi(\pi)(i+1) \tag{17}$$

Essentially, the subscript $+n$ specifies that we should shift where the trace assignment function is applied by one step in the path.

Example: With this definition of HyperLTL, we can now express OD:

$$\forall\pi\forall\pi' G(\bigwedge_{a \in I_{public}} a_\pi \leftrightarrow a_{\pi'}) \implies G(\bigwedge_{a \in O_{public}} a_\pi \leftrightarrow a_{\pi'}) \tag{18}$$

Expressed in plain terms, for any two paths, at all steps both the public inputs and public outputs of each path should be the same.

Example: Consider a password checker, that ensures that two password fields are being filled equivalently up to the current character. In HyperLTL, one can use the *until* operator to express this.

Many security properties are *2-safety properties*, which consider pairs of execution traces [3]. Such properties can be model-checked by self-composition of a system.

Note that HyperLTL can also be extended to include CTL*.

References

[1] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and Csar Snchez. Temporal logics for hyperproperties. In *Principles of Security and Trust*, pages 265–284. Springer, 2014.

[2] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. December 2008.

[3] Tachio Terauchi and Alex Aiken. Secure Information Flow as a Safety Problem. In Chris Hankin and Igor Siveroni, editors, *Static Analysis*, number 3672 in Lecture Notes in Computer Science, pages 352–367. Springer Berlin Heidelberg, 2005.

EECS 219C: Computer-Aided Verification, Spr'15 *Lecturer: S. A. Seshia*

Lecture 4/29: Final Remarks

Scribe: Yasser Shoukry *Editor: Sanjit A. Seshia*

In this lecture, we recap the topics covered in the course and identify unifying themes that appears across many topics. One of the key unifying themes is to do verification "by reduction to synthesis" — i.e., that many verification problems can be solved by reducing them into synthesis problems. Some of the ideas in this lecture have been presented by the lecturer at other venues [3, 2].

1 Recap of Topics

- Computational engines for verification:
  - SAT solving
  - Binary Decision Diagrams (BDDs)
  - SMT solving
  - Syntax-Guided Synthesis (SyGus)
- Model Checking and Verification:
  - Modeling for verification
  - Temporal Logic
  - Model Checking:
    - Explicit state Model Checking (for properties specified using LTL)
    - Symbolic Model Checking (using BDDs for properties specified using CTL)
    - Bounded Model Checking (for properties specified using LTL)
    - Abstraction, Interpolation, ...
    - Compositional Reasoning
  - Deductive Verification
    - Inductive Invariant Checking (e.g. IC3 algorithm)
    - Termination
  - Simulation and Bisimulation
- Misc topics (given by guest lectures):

- Probabilistic Model Checking
- Synthesis from LTL
- Signal temporal logic
- Hyperproperties

2 Verification By Reduction to Synthesis

We start by recalling the notation used in the previous lectures. A *Kripke structure* M is a 4-tuple $M = (S, I, R, L)$ consisting of:

- S: set of states.
- $I \subseteq S$: set of initial states.
- $R \subseteq S \times S$ transition relation.
- $L : S \to 2^{AP}$ labeling function (where AP is a set of atomic propositions).

We also use Φ to denote properties described in some formal language (e.g. LTL or CTL).

2.1 Problem 1: Inductive Invariants

One of the most common verification problems is prove that a property is *invariant*, i.e. we ask the following question:

$$\textbf{verify:} \quad M \models \Phi \qquad \text{where} \quad \Phi = G\phi$$

A typical solution for this verification problem is to use inductive proofs which consists of two steps:

Step 1 Initial step: Check that the initial set satisfies ϕ, i.e., $I \Rightarrow \phi$.

Step 2 Inductive step: Compute the set of reachable states using the transition relation R. Let $\mathcal{R}_i$ be the set of reachable states at iteration i (with $\mathcal{R}_0 = I$) the induction step then checks that the following holds: $\phi_i \wedge \mathcal{R}_i \Rightarrow \phi_{i+1}$ for all iterations i.

The synthesis version of this problem can be formulated as searching for an auxiliary invariant ψ such that $\phi \wedge \psi$ is invariant. Indeed, if $\psi = \text{true}$, then we get back the original

problem again. This synthesis problem can be written as follows:

$$\begin{aligned} \textbf{synthesize:} \quad & \psi \in G \\ \text{subject to:} \quad & I \Rightarrow \phi \wedge \psi \\ & \phi_i \wedge \psi_i \wedge \mathcal{R}_i \Rightarrow \phi_{i+1} \wedge \psi_{i+1} \end{aligned}$$

where G is the class of invariants that is given by the user and represent his domain-specific knowledge. Such class of invariants G can be encoded as a syntactical grammar. Once the problem is formalized as shown above, we can use tools like syntax-guided synthesis (SyGuS) to generate these auxiliary inductive invariants.

2.2 Problem 2: Abstractions

Again, given a system M and a property Φ, the verification problem asks whether the system M satisfies Φ and we write it as:

$$\textbf{verify:} \quad M \models \Phi$$

A standard technique to reduce the complexity of the verification problem is to build an abstraction $\alpha(M)$ from the original concrete system M. To build $\alpha(M)$, we search for an abstraction function α that maps the original set of states S into the reduced abstract states $\hat{S}$, (i.e., $\alpha : S \to \hat{S}$). To incorporate domain specific knowledge, we restrict our search into the abstract domain G which can again be understood as a syntactic constraint that is encoded as a grammar (for example, one can be interested in rolling out the case that α is equal to the identity function). Hence, the synthesis problem can be written as:

$$\begin{aligned} \textbf{synthesize:} \quad & \alpha : S \to \hat{S}, \qquad \alpha \in G \\ \text{subject to:} \quad & \alpha(M) \models \Phi \Leftrightarrow M \models \Phi \end{aligned}$$

In the previous formulation, we ask for an abstraction that is both *sound* and *complete*. A *sound* abstraction (also known as conservative abstraction) ensures that the implication $\alpha(M) \models \Phi \Rightarrow M \models \Phi$ holds. In other words, the *sound* abstraction ensures that the abstraction function α preserves the property Φ.

While in many application soundness may be enough, in other applications, it is not. Recall that *sound* abstractions may introduce false counter examples (traces in the abstract model which violate the property Φ which have no corresponding traces in the original concrete system). Hence, if your objective is to find bugs in a software, then we want to make sure that such false counter examples do not exist in the abstract model. The *completeness* property of the abstraction plays this role. A *complete* abstraction ensures that the implication $\alpha(M) \models \Phi \Leftarrow M \models \Phi$ holds which in turn roles out such false counter examples. If the *completeness* property does not hold, we can rely on techniques like counter-example

guided abstraction refinement (CEGAR) to refine the abstraction. However, the refinement process may not terminate.

Note that one may be interested in obtaining multiple abstraction functions α that leads to a range from coarse-abstractions to fine-abstractions. This may be done by adding more constraints to the synthesis problem and by introducing a measuring function that allows us to measure how fine/coarse is the synthesized abstraction. A similar problem arises in the context of SyGuS where current SyGuS problems ask for *any* satisfying solution without specifying how "good" it should be.

2.3 Problem 3: Reachability Analysis for Symbolic Model Checking

Similar to the previous two problems, we are still interested in the verification problem of the form:

$$\textbf{verify:} \quad M \models \Phi$$

Now recall that for CTL properties we introduced the symbolic model checking technique. Symbolic model checking uses fixed point computations to find all reachable states Z that satisfy the property Φ. Hence, we can view this problem as a synthesis problem whose objective is to synthesize the reachable set Z while the set of initial states I is still included inside Z. This can be written as follows:

$$\begin{aligned} \textbf{synthesize:} \quad & Z \\ \text{subject to:} \quad & Z = SAT_M(\Phi) \\ & I \subseteq Z \end{aligned}$$

Note that in the previous formula, we ask for the exact synthesis of the reachable set. However, in some applications, this condition can be relaxed and we can focus on finding only an approximate set of the reachable states. For example, one can make use of partial order reduction techniques [1] which reduce the state space of the problem without loosing any information that is related to a specific property. Computing the reachable set on the reduced model will not result into the exact reachable set of the original model, yet it is enough to show that the property Φ holds.

We also note that this synthesis problem can be generalized to other domains other than symbolic model checking for CTL properties. For example, in probabilistic model checking, current model checkers use linear programming techniques in order to find a set Z that satisfy similar properties to that above. The previous synthesis problem can be generalized to such scenarios.

2.4 Problem 4: Compositional Reasoning

Compositional reasoning is one of the most important engineering concepts that facilitate verification of large scale systems. However, scientific advances in this domain is still limited. A major blocking step in performing compositional reasoning is the necessity of coming up with the assumptions and the guarantees for each sub-system.

In this setup, we have two subsystems M_1 and M_2 which are composed together. We discuss two cases (i) acyclic case and (ii) cyclic case.

2.4.1 Acyclic Case

As shown in Figure 1, the output of subsystem M_1 is fed to subsystem M_2. Our objective is to verify that property Φ holds o n the output of subsystem M_2. This can be written as:

$$\textbf{verify:} \quad M_1||M_2 \models \Phi$$

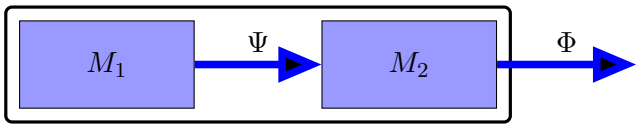


Figure 1: Acyclic composition of M_1 and M_2

In order to verify that property Φ holds, our objective is to find an intermediate proposition Ψ that guarantees the satisfaction of Φ. This intermediate property plays the role of the guarantees of M_1 and the assumptions on M_2. If we are able to automatically synthesize such proposition, then we can reduce the complexity of the compositional reasoning problem. This synthesis problem can be written as:

$$\begin{aligned} \textbf{synthesize:} \quad & \Psi \\ \text{subject to:} \quad & M_1 \models \Psi \\ & M_2 \models \Psi \Rightarrow \Phi \end{aligned}$$

2.4.2 Cyclic Case

In this setup, and for sake of simplicity, we focus on showing that the property $\Phi = G\phi$ is an invariant of the system $M = M_1||M_2$. Similar to the acyclic case, our objective is to automatically synthesize the intermediate propositions which plays the role of assumptions and guarantees on both M_1 and M_2. In particular, our aim is to find two intermediate properties ϕ_1 and ϕ_2 which are invariants of M_1 and M_2, respectively, and together they

guarantee the satisfaction of Φ (i.e., $\phi_1 \wedge \phi_2 \Rightarrow \Phi$). Moreover, because of the cyclic (or feedback) decomposition of the system, we need to use assume-guarantee reasoning to ensure that if certain assumption holds up to time t then we have a guarantee at time $t+1$. This synthesis problem can be written as:

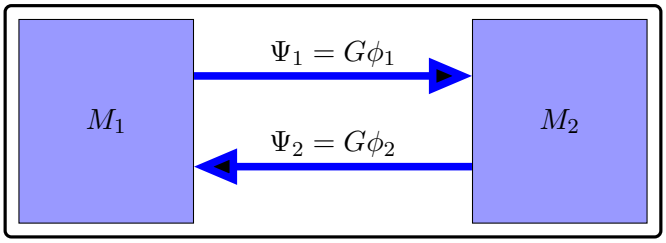


Figure 2: Cyclic composition of M_1 and M_2

$$
\begin{aligned}
\textbf{synthesize:} \quad & \phi_1, \phi_2 \\
\text{subject to:} \quad & I_1 \Rightarrow \phi_1 \\
& I_2 \Rightarrow \phi_2 \\
& M_1 \models \phi_2^{0\ldots t} \Rightarrow \phi_1^{t+1} \\
& M_2 \models \phi_1^{0\ldots t} \Rightarrow \phi_2^{t+1} \\
& \phi_1 \wedge \phi_2 \Rightarrow \phi
\end{aligned}
$$

Note that the above formulation applies to the synchronous composition of M_1 and M_2, since a common notion of time is employed. By including an explicit scheduler process, we might be able to use this rule even for asynchronous systems, but at the cost of some elegance of modeling.

In all cases (synchronous and asynchronous), the challenge is to learn the assumptions and guarantees. By incorporating domain specific knowledge as a syntactical constraint on the assumptions and guarantees, synthesis engines can be used to automatically synthesize them.

2.5 Problem 5: Proofs by Simulation/Bisimulation

In this problem, we are given two systems M_1 and M_2 and we would like to check if M_1 is simulated by M_2. This can be written as the following verification problem:

$$\textbf{verify:} \quad M_1 \leq M_2$$

Recall that one system is simulating the other one if there exists a simulation relation $H \subseteq S_1 \times S_2$ that satisfied certain constraints (e.g. every step of M_1 needs to be matched by a step in M_2, ... etc). Once a relation between S_1 and S_2 is created, it is direct to check if it is indeed a simulation/bi-simulation relation. However, instead of devising H manually, we can use synthesis tools to automatically generate them. This synthesis problem can be written as:

$$\begin{aligned} \textbf{synthesize:} \quad & H \subseteq S_1 \times S_2 \\ \text{subject to:} \quad & \text{the simulation/bi-simulation relation conditions} \end{aligned}$$

Note that all simulation/bi-simulation relation conditions can be casted as boolean functions. In particular, these conditions can be encoded as Quantified Boolean Logic (QBF) for which there exist efficient algorithms to check its satisfiability.

2.6 Problem 6: Termination

Given a software program, our objective is to check the termination of this program. This problem is more interesting when we have infinite systems. To show termination, we encode the program as a Kripke structure M with a special *terminal state*.

As discussed in the previous lecture, in order to show termination, our objective is to find an inductive invariant property ϕ (which is function of the current state) and a transition invariant T (which is function of the current as well as next state) such that T is well-founded. This can be directly encoded as a synthesis problem as:

$$\begin{aligned} \textbf{synthesize:} \quad & \phi, T \\ \text{subject to:} \quad & \phi \wedge R \Rightarrow T \\ & T \wedge R \Rightarrow T' \\ & T \text{ is well-founded} \end{aligned}$$

2.7 Problem 7: SMT Solving

Given a logical formula over multiple theories Φ_{SMT}:

$$\Phi_{SMT} ::= \bigwedge \left(\bigvee L_i \right)$$

where L_i are theory literals. We ask whether this formula is satisfiable:

$$\textbf{verify:} \quad \Phi_{SMT} \text{ is satisfiable}$$

Recall the *Eager* approach for solving SMT in which the SMT formula is encoded into an equi-satisfiable propositional logical formula. This propositional logical formula is then sent

to a SAT solver to check its satisfiability. Similarly, in the *Lazy* approach, the SMT formula is abstracted into a propositional formula which is then refined by iterating between the SAT solver and the Theory solver. In both cases, a unifying scheme is the generation of a propositional logic abstraction. Hence, we can reduce the SMT solving problem into the synthesis of the propositional abstraction as follows:

$$\begin{aligned} \textbf{synthesize:} \quad & \text{propositional logic formula } \Phi_{SAT} \\ & \text{such that:} \quad \Phi_{SAT} \text{ is Satisfied} \quad \Leftrightarrow \quad \Phi_{SMT} \text{ is Satisfied} \end{aligned}$$

One may need more constraints on the structure of Φ_{SAT}. For instance, in the *Lazy* approach, the propositional abstraction takes the form of:

$$\Phi_{SAT} ::= \left[\bigwedge \left(\bigvee l_i\right)\right] \wedge \left[\bigwedge T_j\right]$$

where l_i is the propositional literal that abstracts the theory literal L_i while T_j is the learnt theory lemmas. Again such syntactic constraints can be exploited using syntax guided synthesis (SyGus) tools.

3 Final Remarks

3.1 Solution Strategies

One solution strategy that appeared in many algorithms covered in this class is *counter-example guided learning*. Examples are (i) the Lazy SMT architecture (an extension of the DPLL algorithm to SMT) (ii) counter-example guided abstraction refinement (CEGAR) and (iii) counter-example guided inductive synthesis (CEGIS).

3.2 Some missed topics

Below are some items that we could not cover in this class.

- **Symmetry reduction:** Some systems consists of multiple copies of a smaller subsystem. This symmetric decomposition can be explored by reducing the model checking problem to checking one copy of the small subsystems. By using assume-guarantee reasoning, one can expand the proof to the full system.
- **Deductive verification of Liveness.**
- **Other Computation Models:** In this class we focused only on systems that are modeled as Kripke structures. However, there are model checking algorithms for other computation models (e.g. pushdown automata) which exploits the special structure of those models.

References

[1] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[2] Daniel Kroening, Thomas W. Reps, Sanjit A. Seshia, and Aditya V. Thakur. Decision procedures and abstract interpretation (dagstuhl seminar 14351). *Dagstuhl Reports*, 4(8):89–106, 2014.

[3] Sanjit A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. In *Proceedings of the Design Automation Conference (DAC)*, pages 356–365, June 2012.