SS 2025 15.05.2025

Exercises to the lecture
Semantics
Sheet 5

Prof. Dr. Roland Meyer
Jan Grünke Delivery until 28.05.2025 at 15:00

Excercise: Implement IC3

Implement the IC3 algorithm for checking whether a given hardware circuit satisfies a safety property. Your implementation should accept circuits in the AIGER format that is used in the Hardware Model Checking Competition (HWMCC) and it should either return `SAFE` together with an inductive invariant, or `UNSAFE` along with a counterexample to safety.

Setup

To help you get started, we provide a CMake project template that includes the following:

- A C library for parsing the AIGER file format (from `https://github.com/arminbiere/aiger?tab=readme-ov-file`).
- A tool called `aigToZ3.h`, which transforms a parsed AIGER circuit into a symbolic transition system represented in Z3.
- A basic C++ interface `ic3.h` and example type implementations for a minimal IC3 implementation.

You are free to modify the provided template as needed to suit your implementation. It is intended to let you focus on implementing the core IC3 algorithm without dealing with tasks like file parsing. Each part of the setup will be explained in detail in the following sections of this document. Further information that may support you in implementing IC3 can be found in the following reference:

> Aaron R. Bradley. "SAT-Based Model Checking without Unrolling." In *Proceedings of the 12th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2011. `https://people.eecs.berkeley.edu/~alanmi/publications/2011/fmcad11_pdr.pdf`

Submission

Submit your code as zip or git repository together with instructions on how to build and run it. Your implementation should be tested on examples from the HWMCC benchmark suite that are contained in the `benchmarks` folder.

Z3 Solver

This project requires the Z3 SMT solver for SAT solving. You can find installation instructions in the official Z3 GitHub repository:

- `https://github.com/Z3Prover/z3`

The provided CMake template links Z3 automatically if it is installed on your system.

Symbolic Transition Systems

The behavior of a sequential circuit is captured using a symbolic transition system, represented using Z3 formulas. A symbolic transition system is described by the following C++ struct:

```
struct SymbolicTransitionSystem {
 z3::expr_vector variables;
 z3::expr_vector primedVariables;
 z3::expr transitions;
 z3::expr init;
 z3::expr bad;
};
```

Explanation of properties:

- `variables`: A vector of Z3 expressions representing the state variables of the system (e.g. latches).
- `primedVariables`: A vector of Z3 expressions representing the next-state versions of the variables. Each variable in `variables` has a corresponding next-state version in this vector.
- `transitions`: A formula over `variables` and `primedVariables` that defines the transition relation of the system.
- `init`: A formula over `variables` defining the initial states of the system.
- `bad`: A formula over `variables` representing the negation of the safety property. The system is `SAFE` if no reachable state satisfies this formula.

Interface of `aigToZ3.h`

The `aigToZ3` library provides a set of utility functions to construct and manipulate symbolic transition systems derived from AIGER files. Below is an overview of the main functions and their intended use:

```
z3::expr prime(const SymbolicTransitionSystem &ts, z3::expr expr);
z3::expr expr(const Cube &cube);
z3::expr expr(const std::vector<Cube> &cubes);
std::optional<Cube> SAT(const z3::expr &expr);
SymbolicTransitionSystem parse(const std::string &filepath);
```

- `prime`: Given an expression and a symbolic transition system, this function replaces all state variables in the expression with their corresponding next-state (primed) variables.
- `expr(const Cube&)`: Constructs a Boolean formula representing the conjunction of literals in a `Cube`.
- `expr(const std::vector<Cube>&)`: Constructs a Boolean formula representing the disjunction of multiple cubes. Each cube is treated as a conjunction; the result is a disjunction of those conjunctions.
- `SAT`: Checks the satisfiability of a given Boolean formula using Z3. If the formula is satisfiable, it returns a `Cube` representing a satisfying assignment; otherwise, it returns `std::nullopt`.
- `parse`: Reads an AIGER file from the specified `filepath` and returns a corresponding `SymbolicTransitionSystem` represented using Z3 expressions.

Interface of `ic3.h`

The `ic3.h` header defines the interface for implementing the IC3 (PDR) algorithm.

```
typedef std::vector<Cube> Frame;
typedef std::vector<Frame> Frames;

bool ic3(TS ts);
bool isInitial(TS ts, const Cube &cube);
std::optional<Cube> getBad(TS ts, const Frame &frame);
bool searchPathToInit(TS ts, Frames &frames, Cube &cti);
std::optional<Cube> getPre(TS ts, const Frames &frames, const Cube &cube, int relativeTo
Cube generalize(TS ts, const Frames &frames, const Cube &cube, int frameIndex);
void blockCubeAtFrame(Frames &frames, int frameIndex, const Cube &cube);
bool isInductive(TS ts, const Frames &frames);
void newFrame(Frames &frames);
```

- `ic3`: Runs the IC3 algorithm on the given transition system. Returns `true` if the system is safe, or `false` if a counterexample is found.

- `isInitial`: Returns `true` if the given cube represents a state allowed by the system's initial condition.
- `getBad`: Searches the provided frame for a cube that intersects with the set of bad states. Returns such a cube if one is found, or `std::nullopt` otherwise.
- `searchPathToInit`: Attempts to find a path from a counterexample to the initial states. Returns `true` if a counterexample trace exists; otherwise, strengthens the frames to exclude that path and returns `false`.
- `getPre`: Computes a predecessor `pre` of `cube` satisfying the transition relation so that `pre` lies in the previous frame (which is a forward overapproximation), and ¬`pre` is relative inductive to the previous frame. Returns such a predecessor if it exists or `std::nullopt` otherwise.
- `generalize`: Returns a subcube of the given cube that remains unreachable from the initial states and prior frames.
- `blockCubeAtFrame`: Adds the given cube to the specified frame to block it, strengthening the overapproximation.
- `isInductive`: Checks whether there are equal consecutive frames (i.e., $F_{k+1} \subseteq F_k$). If so, returns `true`; otherwise, returns `false`.
- `newFrame`: Appends a new frame to the sequence of frames, allowing the algorithm to continue if no inductive invariant is yet found.

AIGER Format

You don't need to understand the AIGER format in detail—use the `aigToZ3.h` tool to work directly with a symbolic transition system. The AIGER format is a simple, compact format for describing sequential circuits using And-Inverter Graphs (AIGs). Each AIGER file encodes:

- A list of inputs,
- A list of latches (registers),
- A list of AND gates, and
- A single output (representing the property to be verified).

Latches represent stateful memory elements. Each latch stores a single Boolean value (initalized to `false`) that updates synchronously on each clock cycle. In AIGER, a latch is defined by its current value and the signal defining what value it will take in the next time step. At every time step:

1. All latch values are updated simultaneously according to their next-state functions, which are computed from the previous state.

2. Combinational logic (e.g., AND gates) is re-evaluated based on the new input and updated latch values.

Safety Semantics. The circuit is considered `SAFE` if the output is never equal to 1 in any reachable state.

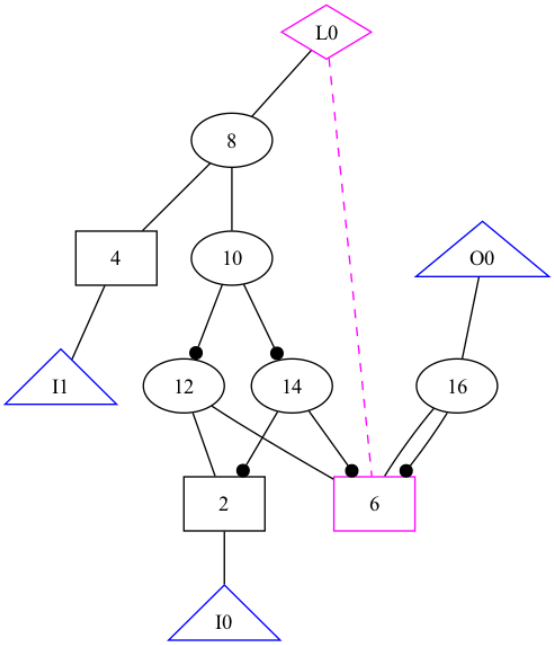
Circuit Visualization. The included AIGER parsing library also supports visualizing circuits defined in `.aig` files. This feature requires `graphviz` to be installed. You can find installation and usage instructions here:

- `https://graphviz.org/download`

You can use these tools to help debug or better understand the structure of the circuits. For more on the aiger format, see: `https://fmv.jku.at/aiger/`.

Example

Circuit. Below is a visual representation of a small AIGER circuit as rendered by the helper tools in the provided library:



Explanation:

- Triangles (blue): Inputs and outputs (e.g., `I0`, `I1`, `O0`).
- Diamonds (pink): Latches (e.g., `L0`).
- Rectangles: State elements.

- Ovals: AND gates or internal signals.
- Dots on edges: Indicate negation (inversion) of the signal being passed.

A possible trace where the values for inputs are chosen arbritrary is shown below:

Time	**Input** `I0`	**Input** `I1`	**Latch** `L0`	**Next** `L0`	**Output** `O0`
0	1	0	0	0	0
1	1	1	0	1	0
2	0	1	1	1	0

Symbolic Transition System. The symbolic transition system for the above circuit is given by:

1. **Initial State:**

$$\texttt{init} := \neg l_0$$

2. **Bad State:**

$$\begin{aligned}\texttt{bad} := a_{16} \\ \wedge (a_8 == i_1 \wedge a_{10}) \\ \wedge (a_{10} == \neg a_{12} \wedge \neg a_{14}) \\ \wedge (a_{12} == i_0 \wedge l_0) \\ \wedge (a_{14} == \neg i_0 \wedge \neg l_0) \\ \wedge (a_{16} == l_0 \wedge \neg l_0) \\ \wedge (l_0' == a_8)\end{aligned}$$

3. **Transition Relation:**

$$\begin{aligned}\texttt{transitions} := (a_8 == i_1 \wedge a_{10}) \\ \wedge (a_{10} == \neg a_{12} \wedge \neg a_{14}) \\ \wedge (a_{12} == i_0 \wedge l_0) \\ \wedge (a_{14} == \neg i_0 \wedge \neg l_0) \\ \wedge (a_{16} == l_0 \wedge \neg l_0) \\ \wedge (l_0' == a_8)\end{aligned}$$

This example illustrates how circuit components and safety conditions are encoded symbolically and how the primed variables represent the next state. Note that $a_{16} = l_0 \wedge \neg l_0$ is a contradiction and will always be false, meaning the circuit is `SAFE`.