# CoSA

User Manual

**Cristian Mattarei**

July 11, 2019

# Contents

# Introduction

CoSA is a symbolic model checker for hardware design. It incorporates a variety of state-of-the-art techniques to achieve performance, while providing a simple and intuitive interface. This document describes all functionalities provided by CoSA, and the description is supported by a series of running examples.

# 1 Overview

The main inputs to CoSA to define a model checking (§ 2.1) verification task are:

- a list of comma-separated input files (§ 3) describing the hardware;
- a verification problem (§ 5) e.g.; safety (§ 5.3), LTL (§ 5.3), or equivalence checking (§ 5.4); and
- a property (§ 4).

The other parameters can be divided into:

- encoding options (§ 9);
- performance optimizations parameters (§ 10); and
- debugging (§ 11).

The results of the analyses, as counterexample traces, are provided in multiple formats (§ 7).

All these options can be either provided as parameters to CoSA on the command line, or as a single problem file (§ 6).

For more information on the actual parameters, run CoSA with `-h`.

# 2 Background

This section provides an overview of the formal aspects at core of CoSA, and it is intended for the users that are not familiar with symbolic model checking and formal verification.

## 2.1 Model checking

Model checking is a technology that allows for an efficient and exhaustive testing of a system. Model checking can be applied to different domains, and the common problem this technique can solve is to prove that a system - in our case a hardware design - meets a set of predefined and expected behaviors, usually represented as system assertions. A model checking problem is usually denoted as $M \models \varphi$, where $M$ is a mathematical representation of the system, and $\varphi$ is the expected behavior. For instance, given a hardware component *Sum* that computes the sum of its input ports $I_1$ and $I_2$, and that provides the result to the output $O$, a conventional testing procedure for *Sum* would require to check that if $I_1 = 0$ and $I_2 = 0$ then $O = 0$, and that $I_1 = 1$ and $I_2 = 0$ results into $O = 1$, and so on. However, with model checking we can directly evaluate if $Sum \models (O = I_1 + I_2)$, and if this is not the case the model checker (such as CoSA) will provide a counterexample to the expected behavior (i.e., $O = I_1 + I_2$), which is represented as a series of assignments to the ports such as $I_1 = 16, I_2 = 13, O = 32$.

Model checking is a fundamental technology when developing complex systems, thus over the years multiple different techniques have been developed to efficiently solve the problem. A major distinction is between symbolic and explicit state (model checking), and the different resides in the technique used to represent the system.

## 2.2 Symbolic transition system

CoSA, being a symbolic model checker, converts all the input models into an internal representation that is called Symbolic Transition System (STS). An STS has an established semantics, and we report its definition in 2.1. Intuitively, an STS is composed of a set of variables, a formula that defines the initial state of the system, and a transitional relation formula that defines how the system evolves.

**Definition 2.1** (Symbolic Transition System)**.** A *Symbolic Transition System* is a tuple $S = \langle V, I, T \rangle$ where $V$ is a set of (input $V_I$, state $V_S$, and output $V_O$) variables, $I(V)$ is a formula representing the initial states, and $T(V, V')$ is a formula representing the transitions. A *state* of $S$ is an assignment to the variables $V_S$.

**Example 2.1** (STS clock behavior encoding)**.** Given a clock signal that starts from 0, and keeps oscillating at every step, its representation would be a tuple $STS_c := \langle V_c, I_c, T_c \rangle$ where:

- $V_c := \{\text{clk}\}$,
- $I_c := \{\text{clk} = 0\}$,
- $T_c := ((\text{clk} = 0) \rightarrow (\text{clk}' = 1)) \wedge$
  $((\text{clk} = 1) \rightarrow (\text{clk}' = 0))$.

The symbols with $'$ (e.g., $\text{clk}'$) are called primed variable, and they represent the value after the transition. For instance, the formula $(\text{clk} = 0) \rightarrow (\text{clk}' = 1)$ states that if $\text{clk} = 0$ before the transition, then clk should be equal to 1 after the transition.

The STS representation is important in CoSA because it allows the user to define complex behavior interaction via synchronous products with other systems. Its definition is reported in 2.2.

**Definition 2.2** (Synchronous Product of STS)**.** Given two Symbolic Transition Systems $S_1 := \langle V_1, I_1, T_1 \rangle$ and $S_2 := \langle V_2, I_2, T_2 \rangle$ where $V_1 \cap V_2 = \emptyset$, the synchronous product $S$ of $S_1$ and $S_2$, namely $S_1 \times S_2$, is defined as $S := \langle V_1 \cup V_2, I_1 \wedge I_2, T_1 \wedge T_2 \rangle$.

**Example 2.2** (Synchronous product of STSs)**.** This example shows how the synchronous product can be used to integrate multiple systems. We start to introduce a counter that goes from 0 to 3 at every posedge of "clk", defining it with the tuple $STS_3 := \langle V_3, I_3, T_3 \rangle$ where:

- $V_3 := \{\text{output}, \text{clk}\}$,
- $I_3 := \{\text{output} = 0\}$,
- $T_3 := (((\text{clk} = 0) \rightarrow (\text{clk}' = 1)) \rightarrow$
  $(((\text{output} < 3) \rightarrow (\text{output}' = \text{output} + 1)) \wedge$
  $((\text{output} \geq 3) \rightarrow (\text{output}' = 0)))) \wedge$
  $(\neg((\text{clk} = 0) \rightarrow (\text{clk}' = 1)) \rightarrow$
  $(\text{output}' = \text{output}))$.

We can notice that the $STS_3$ does not constraint the behavior of "clk", and it is used only used in the transition relation to defined the behavior of the variable "output", and in principle, the counter can provide the value 0 forever if the clock does not oscillate. The synchronous product between $S_3$ and $S_c$ (defined in the Example 2.1) integrates the formulae from both systems, and it defines a composed STS that imposes the counter to expose its expected behavior.

An important aspect of the synchronous products is that it may leads to have deadlock states, usually from contradictory behaviors. For instance, if $S_1 := \langle \{v\}, v = 0, \text{True} \rangle$ and $S_2 := \langle \{v\}, v = 1, \text{True} \rangle$, the resulting $S_1 \times S_2$ will be $\langle \{v\}, \textbf{False}, \text{True} \rangle$, considering that $(v = 0) \wedge (v = 1)$ is contradictory.

### 2.3 Linear Temporal Logic (LTL)

Differently from safety verification (e.g., assertions), the Linear Temporal Logic defines a behavior over system executions. The syntax of LTL is defined as follows:
$\varphi ::= p \mid \text{True} \mid \text{False} \mid \neg\varphi \mid \varphi \wedge \gamma \mid \varphi \vee \gamma \mid \varphi \rightarrow \gamma \mid \mathbf{X}(\varphi) \mid \mathbf{F}(\varphi) \mid \mathbf{G}(\varphi) \mid \varphi\mathbf{U}\gamma \mid \varphi\mathbf{R}\gamma$
where:

- $p$ is an atomic proposition,
- $\wedge$, $\vee$, $\rightarrow$, and $\neg$ are Boolean operators respectively for and, or, implies, and negation, and
- **X**, **F**, **G**, **U**, and **R** are temporal operators:
  - **G**lobally, e.g., $\mathbf{G}(\varphi)$ means that $\varphi$ has to hold for every states,
  - **F**inally or Eventually, e.g., $\mathbf{F}(\varphi)$ means that $\varphi$ has to eventually hold,
  - ne**X**t, e.g., $\mathbf{X}(\varphi)$ means that $\varphi$ has to hold at the next state,
  - **U**ntil, e.g., $\gamma\mathbf{U}\varphi$ means that $\gamma$ has to hold until $\varphi$ becomes *True*,
  - **R**elease, e.g., $\gamma\mathbf{R}\varphi$ means that $\varphi$ is *True* until $\gamma$ becomes *True*.

**Example 2.3** (Common LTL formalizations)**.** Following we report some common formalization examples:

- "the output value is always equal to 3":
  $\mathbf{G}(\text{output} = 3)$,
- "the output value will eventually reach 3":
  $\mathbf{F}(\text{output} = 3)$,
- "whenever there is a posedge reset, output should be equal to 0":
  $\mathbf{G}(((\text{rst} = 0) \wedge \mathbf{X}(\text{rst} = 1)) \rightarrow \mathbf{X}(\text{output} = 0))$,
- "at the first posedge reset, output should be equal to 0":
  $\mathbf{F}(((\text{rst} = 0) \wedge \mathbf{X}(\text{rst} = 1)) \rightarrow \mathbf{X}(\text{output} = 0))$,
- "the ready signal should always eventually be equal to 1":
  $\mathbf{G}(\mathbf{F}(\text{ready} = 1))$,
- "every time that there is a send signal, eventually receive should hold":
  $\mathbf{G}(\text{send} \rightarrow \mathbf{F}(\text{receive}))$.

# 3 Input formats

CoSA supports multiple input formats, and while being equivalent from an expressivity point of view, they are tailored and design to accomplish different purposes. CoSA distinguish between different formats by relying on their file extension. Running `CoSA -h` shows the list of accepted input formats, and make sure that the configuration is correct.

CoSA accepts a comma-separated list of files, and the resulting model is **synchronous product** (§ 2.2) between all of them. This allows for a clear and explicit definition of **reset procedures**, as well as environmental assumptions.

An input file can be provided with additional model flags listed in squared brackets, which instruct the encoder of how to process the file.

**Example 3.1** (Model flags)**.** `CoSA -i input_file.extension[model_flag]` provides `input_file.extension` as input file, with `model_flag` as a model flag.

Following, we cover all the formats supported by CoSA.

## 3.1 Verilog

This format is natively supported by relying on PyVerilog, and it requires a top-level model flag.

**Example 3.2** (Verilog input)**.** `CoSA -i examples/counters_4/counters_4.v[Counters_4]` provides `examples/counters_4/ counters_4.v` as input model, with `Counters_4` as a top module.

## 3.2 SystemVerilog

SystemVerilog is supported using Verific, which is an industrial tool that can translate SystemVerilog to Verilog. After translating the file using Verific, CoSA relies on the internal Verilog encoder to process the model. The SystemVerilog encoder also requires to provide the top module as model flag.

**Example 3.3** (SystemVerilog input)**.** `CoSA -i examples/counters_4/counters_4.sv[Counters_4]` provides `examples/counters_4/ counters_4.sv` as input model, with `Counters_4` as a top module.

## 3.3 Verilog/SystemVerilog with Yosys

CoSA can also use the open-source synthesis tool, Yosys, to read Verilog and (some) SystemVerilog. This approach can use Yosys to simplify the circuit and can result in faster verification. If yosys is installed and available at the command line, then this is the default Verilog/SystemVerilog parser. To read in multiple files, use the `*.vlist` extension with the top module as a model flag.

## 3.4 CoreIR

CoreIR [1] is supported by relying on PyCoreIR. Its file extension is `.json`. The format accepts model flags to instruct the encoder to extract additional information from the model such as lemmas from CoreIR optimization passes [3].

Currently, CoSA supports the following CoreIR model flags:

- `FC-LEMMAS`: it automatically extracts the lemmas from the fold constants CoreIR pass.

**Example 3.4** (CoreIR input with fold-constants lemmas)**.** `CoSA -i examples/counters/counters.json` to load the `examples/counters/counters.json` design, or `CoSA -i examples/fold-constants/ mpe_fc.json[FC-LEMMAS]` to load `examples/fold-constants/ mpe_fc.json` with `FC-LEMMAS` as model flag.

## 3.5 BTOR2

BTOR2 [4] is a very concise format to represent SMT-based hardware designs. Its extension is `.btor` or `.btor2`, and it is mainly used to interface with other tools such as Yosys [5], since it can produce such format.

## 3.6 Symbolic Transition System (STS)

This format allows for the definition of a component-based Symbolic Transition System, which is characterized by:

- system variables, divided into `STATE`, `INPUT`, `OUTPUT`, and `VAR`
- initial states formula, i.e., `INIT`
- transition relation formula, i.e., `TRANS`
- invariant formula, i.e., `INVAR`, which constraints every states of the system, including the initial ones

The language supports also a typed modules definition and instantiation. A module can be defined using the keyword `DEF`, followed by a list of parameters, while its instantiation should be defined in the `VAR` section.

Simple definition of an 8-bit counter with clock and reset is reported in Figures 1, and 2. The latter is defined using sub-modules instantiation, and the lines that starts with `#` are comments.

Figure 1: Counter example

```
VAR
clk: BV(1);
rst: BV(1);

OUTPUT
out: BV(8);

INIT
out = 0_8;
clk = 0_1;

TRANS
# Clock behavior definition
(clk = 0_1) <-> (next(clk) = 1_1);
# When posedge and not reset we increase out by 1
(posedge(clk) & ! posedge(rst)) -> (next(out) = (out + 1_8));
# When not posedge and not reset we keep the value of the out
(! posedge(clk) & ! posedge(rst)) -> (next(out) = (out));
# When reset we set out to 0
posedge(rst) -> (next(out) = 0);
```

Figure 2: Counter example (hierarchical)

```
VAR
  clk: BV(1);
  rst: BV(1);
  counter_1: Counter(clk, rst);

OUTPUT
  out: BV(8);

TRANS
  # Clock behavior definition
  (clk = 0_1) <-> (next(clk) = 1_1);

INVAR
  # Enforcement of the equivalence between local output value
  # and the output of the sub-module
  out = counter_1.out;

DEF Counter(clk: BV(1), rst: BV(1)):
  VAR
  out: BV(8);

  INIT
  out = 0_8;

  TRANS
  # When posedge and not reset we increase out by 1
  (posedge(clk) & ! posedge(rst)) -> (next(out) = (out + 1_8));
  # When not posedge and not reset we keep the value of the out
  (! posedge(clk) & ! posedge(rst)) -> (next(out) = (out));
  # When reset we set out to 0
  posedge(rst) -> (next(out) = 0);
```

### 3.7 Explicit state Transition System (ETS)

This format allows for the definition of an Explicit States Transition System, which is characterized by two sections:

- states definition, expressed as values assignments to system variables, e.g., `I: clk = 0_1` for the initial state or `S1: output = 4_8` for the state S1, and so on
- transitions definition, e.g., `I -> S1` defines a transition from the state I to the state S1

The language does not require types definitions, because they are inferred by the values assignment. Figure 3 shows a an example of a 2-bit counter, and the lines that starts with # are comments.

Figure 3: 2-bit Counter

```
# States definition
I: output = 0_2
S1: output = 1_2
S2: output = 2_2
S3: output = 3_2

# Transitions
I -> S1
S1 -> S2
S2 -> S3
S3 -> I
```

The ETS format is particularly suited for the definition of sequential behaviors such as the **reset procedures**. In fact, most hardware definitions require to be properly initialized before performing any analysis. In the following example, the `reset_done` variable is used to keep track of the reset status, and it is used to specify a pre-condition for the verification properties.

Figure 4: Reset procedure example

```
I: rst = 0_1
I: reset_done = False

S1: rst = 1_1
S1: reset_done = False

SE: rst = 0_1
SE: reset_done = True

I -> S1
S1 -> SE
# the reset_done signal remains up forever, defined as a self-loop on the SE state
SE -> SE
```

**Example 3.5** (ETS input and synchronous product)**.** `CoSA -i examples/counters_4.v[Counters_4],examples/counters_4/rst_beh.ets` to load the `examples/counters_4.v` design and the ETS `examples/counters_4/rst_beh.ets` and perform a synchronous product between them.

## 3.8 Initial State Constraints (INIT)

When performing formal verification, it can often be useful to start the model in a custom initial state. In fact, this can be necessary for reaching deep bugs or avoiding long reset sequences.

CoSA supports a special format using the `.init` extension to set the initial values of state elements. A `*.init` file can be passed to CoSA through the `--init` command line flag or problem file option. Using `ETS` or `STS`, a user can add to the initial state constraints, whereas the `INIT` format replaces any existing initial state constraints with a concrete state. Typically this concrete state was obtained from a simulation.

The syntax resembles that of the `ETS` format without state labels, but ignores previously undefined or non-state symbols. This is necessary because the initial state is often obtained from a VCD trace which might include non-state values. Note, "reg" signals in Verilog are not necessarily state elements, so VCD traces don't contain enough information to distinguish state elements from combinational logic. Please see the `scripts/vcd2init.py` script for producing a `.init` file from the final state of a VCD trace.

*Current limitation:* Due to the communication format between Yosys and CoSA for Verilog input, the `vcd2init.py` will not behave as expected unless you run CoSA with `--abstract-clock`, or an option that implies an abstract clock, such as `--synchronize`. This is because the BTOR format for an explicit clock encoding does not preserve user-defined state variable names. The name still exists in the system, but it becomes a wire instead of a state element.

Figure 5: Initial state example

```
A = 0_4
B = 3_4
mem[0_4] = 0_4
memacc(mem, 1) = 1_4 # memacc let's you access a memory with an integer instead of a bit-vector
```

# 4 Properties

CoSA supports the definition of invariant and Linear Temporal Logic (LTL) properties. Moreover, the tool allows for the definition of syntactic sugar and parametric generators. CoSA relies on PySMT [2] for the formulae management, thus the syntax is going to related to this tool.

## 4.1 Invariant

An invariant property is a propositional formula over the system variables that has to hold at any time.

**Example 4.1** (Invariant property)**.** Given the `examples/counters_4.v` model, the invariant property `out < 10_16` states that the value of `out` should be always less than the decimal encoding of `10` as a bit-vector of size 16.

An invariant property can predicate also over the `next` variables, which are the primed version of each variable.

**Example 4.2** (Invariant property with next)**.** Given the `examples/counters_4.v` model, the invariant property `(rst = 1_1) -> (next(out) = 0_16)` states that when the reset signal `rst` is equal to 1 (encoded as a bit-vector of size 1), the primed value of `out` should be `0` (encoded as a bit-vector of size 16).

## 4.2 Linear Temporal Logic

The support for LTL extends propositional logic with additional temporal operators:

- `G`: for globally,
- `F`: for finally/eventually,
- `next`: for next,
- `U`: for until,
- `R`: for release.

**Example 4.3** (LTL property with Finally)**.** Given the `examples/counters.json` model, the LTL property `F(self.out = 4_16)` states that the signal `self.out` must reach the value `4` at some time in the future.

**Example 4.4** (LTL property with Finally Globally)**.** Given the `examples/counters.json` model, the LTL property `F(G(self.out = 4_16))` states that the signal `self.out` must reach the value `4` at some time in the future, and keep the value forever.

**Example 4.5** (LTL property with Globally Finally)**.** Given the `examples/counters.json` model, the LTL property `G(F(self.out = 4_16))` states that the signal `self.out` must reach the value `4` infinitely many times.

### 4.3 Syntactic sugar

In order to simplify the definition of hardware properties, CoSA integrates an expandable support for syntactic sugar. Additional syntactic sugars should be defined in `cosa/encoders/sugar.py`, and registered in `cosa/encoders/factory.py`. The one that are registered can be seen in the `special operators` section of the CoSA helper (parameter `-h`). Currently, CoSA supports the following syntactic sugar:

- `posedge`, with a single parameter. `posedge(variable)` is equivalent to `(variable = 0_1) & next(variable = 1_1)`, if `variable` is a bit-vector of size 1, while `(variable) & next(! variable)` if `variable` is of Boolean type,
- `negedge`, which is implemented as `posedge` with inverted polarity,
- `change`, with a single parameter. `change(variable)` is equivalent to `(variable != next(variable)`,
- `nochange`, with a single parameter. `nochange(variable)` is equivalent to `(variable = next(variable)`,
- `ones`, with a single parameter. `ones(variable)` provides the highest possible value of `variable`,
- `zero`, with a single parameter. `zero(variable)` provides the bit-vector encoding of all zeros, given the size of `variable`,
- `dec2bv`, with a two parameters. `dec2bv(value, variable)` provides the bit-vector encoding of value, given the size of `variable`.

### 4.4 Generators

More complex behaviors sometimes require the definition of parametric modules to support the verification. The generators help on simplifying those properties, by providing a parametric way to defined commonly used symbolic transition systems. A common example of a generator is the definition of a *scoreboard* to verify the behavior of a *FIFO*. The purpose of a *scoreboard* is to keep track of a packet that goes into the *FIFO*, and tells when it is supposed to come out, according with the number of pushes and pops that have been performed. An approach to define such behavior would be to define an STS that models the *scoreboard*, which in synchronous product with the original system would allow the user to specify the appropriate property. However, with the CoSA generators this verification is much simpler.

**Example 4.6** (Generators usage)**.** Given the FIFO implementation in `examples/fifo/fifo.sts`, the property that verify its behavior is defined as `sb.end -> (sb.packet = output)` by instantiating the generator `sb=FixedScoreboard (input, 6, posedge(clk))`. The meaning of the property is that when signal `end` from the *scoreboard* `sb` is *True*, then the `packet` should be equivalent to the `output` of the *FIFO*. While the instantiation of the *scoreboard* takes as input the `input` of the *FIFO*, its length (in this case is 6), and the signal that triggers the coming of a new packet.

The generators can be defined in `cosa/encoders/generators.py` and should be registered in `cosa/encoders/factory.py`. CoSA currently provides this set of generators:

- `FixedScoreboard`: *scoreboard* for a FIFO with no pop. The parameters are (input_port, max_value, push_signal), and its values are (end, tracking),
- `Scoreboard`: *scoreboard* for a FIFO with push and pop. The parameters are (input_port, max_value, push_signal, pop_signal), and its values are (end, tracking),
- `Random`: provides a non-deterministic value. The parameter is (size) and its value is (value), and it can be either a variable (it would consider its size), or a decimal value.

## 5 Verification definition

This section covers the definition of a verification task, including the environmental behavior where the system design is supposed to operate.

## 5.1 Environmental assumptions

A circuit usually requires to be analyzed assuming a specific environmental behavior. This might include clock behaviors, reset procedures, and valid inputs configurations. This set of assumptions can be set relying on additional models defined using STSs or ETSs and include them in the list of input files. However, CoSA provides also a set of automated procedures to add environmental behaviors. Currently, the tool supports automated clock definition.

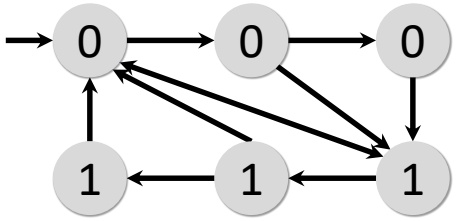


Figure 1: Non-deterministic clock behavior of period = 3

The parameter `--add-clock` links a clock behavior to each input signals whose name contains the "clk", and their behavior is defined by the parameter `--clock-behaviors`. Clock behaviors can be defined in the file `cosa/encoders/clock.py` and should be registered in the file `cosa/encoders/factory.py`. Currently, CoSA supports the following clock behaviors:

- `DetClock`: a deterministic behavior with fixed period. The parameters are (clock_variable, period), and it will generate a behavior that oscillates from 0 to 1, keeping its value constant for period steps,
- `ConstClock`: a constant behavior with fixed value. The parameters are (clock_variable, value), and it will keep the value forever. This is usually used with clock abstraction (see Section 10),
- `NondetClock`: a non-deterministic behavior with maximum period. The parameters are (clock_variable, period), and it will generate a behavior that oscillates from 0 to 1, keeping its value constant for at most period steps. Figure 1 shows the transition system of a non-deterministic clock with period = 3.

If `--clock-behaviors` is not defined, the default configuration is reported in Table 1.

| # of clock signals | clock abstraction | clock behaviors |
|---|---|---|
| 1 | No | `DetClock`, period = 1 |
| $> 1$ | No | `NondetClock`, period = 3 |
| 1 | Yes | `ConstClock`, value = 0 or 1 * |
| $> 1$ | Yes | `ConstClock`, value = 0 or 1 * |

*: 1 if its behavior is defined only on posedge, 0 otherwise

Table 1: Default clock behaviors

## 5.2 Simulation

The simulation generates a system execution of a provided depth. This verification does not require a property, in that case it will be set to *True*. If a property is provided the shortest execution reaching a state that satisfies the property is provided. If such a state does not exist, the verification fails. Table 2 summarizes the possible results of the analysis, given `bmc_length` parameter.

**Example 5.1** (Simulation)**.** Running `CoSA -i examples/counters/counters.sts --verification simulation` will generate a system execution of length equal to the default setting for the parameter `-k`, which is 10.

| Property | Trace exists? | Result | Trace length |
|---|---|---|---|
| *True* | Yes | `TRUE` | k+1 |
| *True* | No | `FALSE` | NA |
| $\varphi$ | Yes | `TRUE` | min(k) : $\varphi$ |
| $\varphi$ | No | `FALSE` | NA |

Table 2: Simulation results

### 5.3 Safety and LTL verification

Safety (parameter `--verification safety`) and LTL (parameter `--verification ltl`) verifications require to provide a property using the parameter `-p`. An additional parameter `--prove` requires the tool to try to prove that the property holds, independently from the depth of the analysis (i.e., `--bmc_length`/`-k` parameter). The result of the analysis can be either `FALSE`, `TRUE`, or `UNKNOWN`. The latter is provided when only a bounded proof exists, and the probability to find an unbounded proof is dependent from the depth of the analysis. Some input files can contain also property definitions, and the usage of `--verification safety` or `--verification ltl` parameter includes also those properties in the verification task. Table 3 summarizes the possible results according with the parameter `--prove`, and the existence of a proof.

| Property | Prove | Trace exists? | Proof found? | Result | Trace length |
|---|---|---|---|---|---|
| $\varphi$ | Yes | Yes | NA | `FALSE` | min(k) : $\varphi$ |
| $\varphi$ | Yes | No | Yes | `TRUE` | NA |
| $\varphi$ | Yes | No | No | `UNKNOWN` | NA |
| $\varphi$ | No | Yes | NA | `FALSE` | min(k) : $\varphi$ |
| $\varphi$ | No | No | NA | `UNKNOWN` | NA |

Table 3: Safety and LTL results

**Example 5.2** (Prove property)**.** Running `CoSA -i examples/counters/counters.json --add-clock --verification safety -p "count1.r.reg0.out < 19_16"` performs a safety analysis of the property `count1.r.reg0.out < 19_16`, and the result is shown in Figure 6. While adding the parameter `--prove` to the command, CoSA is able to prove the property and it provides the result shown in Figure 7.

Figure 6: Safety example (UNKNOWN)

```
1  ** Problem safety **
2  Result: UNKNOWN
3  BMC depth: 10
```

Figure 7: Safety example (TRUE)

```
1  ** Problem safety **
2  Result: TRUE
```

#### 5.3.1 Formula Syntax

For properties, assumptions and lemmas, we rely on the PySMT parser to interpret logical formulas. Please see the `HRLexer` rules here for a list of the supported operators.

Useful hint: Note that these logical formulas are "strongly typed". In particular, there is a difference between a boolean variable and a bit-vector size one and they cannot be used interchangeably. By default, variables obtained from input models that do not distinguish between the two (e.g. Verilog, BTOR2) are encoded as bit-vectors. Thus it is common to convert bit-vectors to booleans by comparing to a bit-vector constant for use in boolean formulae.

Example:

```
(rst = 1_1) -> (next(config_reg) = 0_16)
```

where bit-vector constants are written `<value>_<bitwidth>`, e.g. a bit-vector of size one with value zero is `0_1`.

## 5.4 Equivalence checking

The equivalence checking takes two systems and it verifies if they start from the same initial state, and if they always receive the same inputs they will always provide the same output. This verification requires that the two systems have the same interface. The results of this analysis is similar to the safety and LTL verification, and are reported in Table 4.

| Prove | Trace exists? | Proof found? | Result | Trace length |
|---|---|---|---|---|
| Yes | Yes | NA | `FALSE` | min(k) : $sys_1 \neq sys_2$ |
| Yes | No | Yes | `TRUE` | NA |
| Yes | No | No | `UNKNOWN` | NA |
| No | Yes | NA | `FALSE` | min(k) : $sys_1 \neq sys_2$ |
| No | No | NA | `UNKNOWN` | NA |

Table 4: Equivalence results

**Example 5.3** (Equivalence checking)**.** Running `CoSA -i examples/mul_2/mul_2.json --equivalence examples/mul_2/mul_2_pe.json --prove -k 15` performs an equivalence checking between `examples/mul_2/mul_2.json` and `examples/mul_2/ mul_2_pe.json`, and the result is shown in Figure 8.

Figure 8: Equivalence example (UNKNOWN)

```
1  ** Problem equivalence **
2  Result: UNKNOWN
3  BMC depth: 15
```

## 5.5 Parametric model checking

CoSA supports also parametric model checking analysis on extended models. At the current stage, CoSA supports only to extend a model with the internal model extension, which requires to select one of the possible faulty behaviors using the parameter `--model-extension`. The possible extension can be shown in the CoSA helper (`CoSA -h`) under the section `model modifiers`. Additional modifiers can be defined in the `cosa/modifiers/model_extension.py` and should be registered in `cosa/modifiers/factory.py`. At the current stage, CoSA supports the following variable modifiers:

- `NonDeterministic`: allows for any possible value,
- `Zero`: fixes the value to 0,
- `High`: fixes the value to the highest possible value,
- `Inverted`: inverts the correct value.

The result of a parametric model checking analysis is the region of parameters (in this case the fault variables) that allow the system to violate the property. As a precondition to this analysis, such property should hold in the original model, otherwise the region of parameters will be equal to *True*. For instance, the property in the Example 5.4 holds for the given model, while performing the parametric analysis on the extended model, as in the Example 5.5, it provides a region required to be negated in order to satisfy the property. The parametric analysis is also dependent to the parameter `--cardinality`, which provides the upper bound of the number of failure that can occur in the system.

**Example 5.4** (Parametric verification (nominal analysis)). Running `CoSA -i examples/counters_4/counters_4.v[Counters_4],examples/counters_4/ rst_beh.ets --add-clock --verification safety -p "(reset_performed & ! posedge(rst) & posedge(clk)) -> ((next(out) < (out + 2_16)))" -a "reset_perfor -> (rst = 0_1);out < 240_16" --prove --prefix trace` performs a safety analysis over the given model, property, and assumption, and the result is shown in Figure 9.

Figure 9: Safety analysis example

```
1  ** Problem safety **
2  Result: True
```

**Example 5.5** (Parametric verification (faults analysis)). Running `CoSA -i examples/counters_4/counters_4.v[Counters_4],examples/counters_4/ rst_beh.ets --add-clock --model-extension High --parametric -p "(reset_performed & ! posedge(rst) & posedge(clk)) -> ((next(out) < (out + 2_16)))" -a "reset_performed -> (rst = 0_1);out < 240_16" --prefix trace` performs a parametric model checking analysis extending the model with a `high` faulty behavior. The result is shown in Figure 10, and the region three possible (minimal) assignments of cardinality 2.

Figure 10: Parametric analysis example

```
1  ** Problem parametric **
2  Result: UNKNOWN
3  BMC depth: 10
4  Region:
5   - (counter_clk.out$FAILURE$ & counter_2.out$FAILURE$) or
6   - (counter_clk.out$FAILURE$ & counter_4.out$FAILURE$) or
7   - (counter_clk.out$FAILURE$ & counter_3.out$FAILURE$)
8  Executions: [1], [2], [3]
9  Traces (max) length: 4
10
11 *** TRACES ***
12
13 [1]:    trace[1]-parametric.txt
14 [2]:    trace[2]-parametric.txt
15 [3]:    trace[3]-parametric.txt
```

# 6 Problem Files

In order to facilitate the usage of the tool in case of complex and comprehensive analysis of a model, CoSA supports the definition of problem files. In practice, a problem file links a model with a set of formal analyses that have to be performed, and it is provided to the tool with the parameter `--problems`.

Figure 11 shows an example of a problem file, which is composed of the following sections:

- `[GENERAL]`: it is required, and it defines the system model under analysis. A problem file accepts only one model, thus all encoding options have to be defined in this section. The files paths are local to the problem file, unless they start with `/` symbol,
- `[DEFAULT]`: this section is optional, and it defines the default values for all the problems in this problem file.
- a list of verification tasks (problems): each of them starts with a unique identifier in square brackets, and they set all the possible parameters that define the verification task.

Table 5 lists the possible parameters for the `[GENERAL]` section, while Table 6 lists the parameters for the `[DEFAULT]` section and each verification. For more information about each parameter, see the help message, `CoSA --help`.

| Parameter | Default | Parameter | Default |
|---|---|---|---|
| abstract_clock | False | add_clock | False |
| assume_if_true | False | boolean | None |
| clock_behaviors | None | equivalence | None |
| init | None | model_extension | None |
| model_files | None | run_coreir_passes | True |
| synchronize | False | symbolic_init | False |
| vcd | False | verbosity | None |
| zero_init | None | | |

Table 5: `[GENERAL]` section options

| Parameter | Default | Parameter | Default |
|---|---|---|---|
| assumptions | None | bmc_length | 10 |
| bmc_length_min | 0 | cache_files | False |
| cardinality | -1 | coi | False |
| description | None | expected | None |
| properties | None | full_trace | False |
| generators | None | incremental | None |
| lemmas | None | precondition | None |
| prove | False | smt2_tracing | None |
| solver_name | None | strategy | None |
| time | False | trace_all_vars | False |
| trace_prefix | None | trace_values_base | None |
| trace_vars_change | False | vcd | False |

Table 6: `[DEFAULT]` and verification sections options

## 6.1 Useful Hints

- properties, assumptions and lemmas can have multiple entries separated by ';'. Note: it is best practice to create a new problem for each property though, so that each property has a description
- option values can span multiple lines, provided the subsequent lines are indented. Example:

```
assumptions: <assumption 1>;
             <assumption 2>
```

## 6.2 Examples

Figure 11: Problem file example

```
1  [GENERAL]
2  model_files: model.v[main_module]
3  add_clock: True
4
5  [DEFAULT]
6  bmc_length: 40
7
8  [safety_verification_1_id]
9  description: "Description of the safety verification 1"
10 properties: <property to be verified>
11 assumptions: <assumptions for the verification>
12 verification: safety
13 prove: True
14 expected: False
```

```
15
16  [safety_verification_2_id]
17  description: "Description of the safety verification 2"
18  ...
19
20  [ltl_verification_1_id]
21  description: "Description of the LTL verification 1"
22  ...
```

**Example 6.1** (Problem file definition)**.** The Figure 12 shows the problem file in `examples/counter/problem.txt`, and it can be run with the command `CoSA --problems examples/counter/problem.txt --prefix trace`.

Figure 12: Problem file in `examples/counter/problem.txt`

```
1   [GENERAL]
2   model_files: counter.json,counter_live.sts
3   add_clock: True
4
5   [DEFAULT]
6   bmc_length: 40
7
8   [Globally]
9   description: "Globally Check"
10  properties: self.out < 4_16
11  assumptions: en_clr = 0_1 -> self.clr = 0_1
12  verification: safety
13  prove: True
14  expected: False
15
16  [Finally]
17  description: "Finally Check"
18  properties: F(self.out = 4_16)
19  assumptions: en_clr = 0_1 -> self.clr = 0_1
20  verification: ltl
21  prove: True
22  expected: True
23
24  [Liveness]
25  description: "Liveness Check"
26  properties: F(G(self.out = 4_16))
27  assumptions: en_clr = 0_1 -> self.clr = 0_1
28  verification: ltl
29  prove: True
30  expected: False
```

Figure 13 shows the execution of the problem file in Figure 12. After performing the analysis, CoSA provides a summary of the results, and it grabs the attention to the analyses that have an unexpected result.

Figure 13: Analysis for the problem in Figure 12

```
1   Parsing file "examples/counter/counter.json"... DONE
2   Parsing file "examples/counter/counter_live.sts"... DONE
3   Solving "Globally" ... TRUE
4   Solving "Finally" ........ TRUE
5   Solving "Liveness" .............. FALSE
6
7   *** SUMMARY ***
8
9   ** Problem Globally **
10  Description: "Globally Check"
11  Result: TRUE
```

```
Expected: FALSE
TRUE != FALSE <<<---------| ERROR

** Problem Finally **
Description: "Finally Check"
Result: TRUE
Expected: TRUE

** Problem Liveness **
Description: "Liveness Check"
Result: FALSE
Expected: FALSE
Counterexample: [1]
Trace length: 16

*** TRACES ***

[1]:    trace[1]-Liveness.txt

WARNING: Verifications with unexpected result
```

**Example 6.2** (Problem file definition (equivalence))**.** The problem file reported in Figure 14 shows the definition of an equivalence checking. In this case, the model under analysis is the composition between `mul_2.json`, and `mul_2_pe.json`, thus all verifications are related to it.

Figure 14: Problem file in `examples/mul_2/problem_1.txt`

```
[GENERAL]
model_files: mul_2.json
add_clock: True

[DEFAULT]
bmc_length: 5

[Equivalence Checking]
description: "Mul2 is equivalent to Mul2 PE"
verification: equivalence
equal_to: mul_2_pe.json
prove: True
expected: Unknown
```

# 7 Results analysis

This section describes the information that CoSA produces after a verification is performed.

## 7.1 Counterexample traces

The result of the verification can in some cases provide a counterexample trace, which is an execution of the system that supports the result of the analysis. CoSA produces finite-state traces, but in case of LTL verification the trace can be infinite with a lasso-shaped loop. CoSA can generate traces either in a human readable textual format or in VCD format.

### 7.1.1 Finite traces

The format used by CoSA to represent traces is similar to the ETS format (as shown in the Figure 15), and it provides a series of states with assignments to the variables. The default configuration shows only the top level `INPUT` and `OUTPUT` ports, and all variables can be shown by using the parameter `--trace-all-vars`. Moreover, a trace shows only the variables that do not change their value, and this option can be disabled with the parameter `--trace-vars-change`.

**Example 7.1** (Safety counterexample trace)**.** Running `CoSA -i examples/counters_4/counters_4.v[Counters_4] --verification simulation -k5` will generate a system execution of length equal to 6 (5 steps + initial state), as shown in Figure 15.

Figure 15: Simulation Counter_4

```
** Problem simulation **
Result: TRUE
Execution:
---> INIT <---
  I: clk = 1_1
  I: out = 8_16
  I: rst = 1_1

---> STATE 1 <---

---> STATE 2 <---

---> STATE 3 <---

---> STATE 4 <---
  S4: clk = 0_1
  S4: rst = 0_1

---> STATE 5 <---
  S5: out = 0_16
  S5: rst = 1_1
```

**Example 7.2** (Counterexample with all variables)**.** Running `CoSA -i examples/counters_4/counters_4.v[Counters_4] --verification simulation -k5 --trace-vars-change` will generate a system execution of length equal to 6 (5 steps + initial state), showing also the variables that change value, as shown in Figure 16.

Figure 16: Simulation Counter_4 (with changing values)

```
** Problem simulation **
Result: TRUE
Execution:
---> INIT <---
  I: clk = 1_1
  I: out = 8_16
  I: rst = 1_1

---> STATE 1 <---
  S1: clk = 1_1
  S1: out = 8_16
  S1: rst = 1_1

---> STATE 2 <---
  S2: clk = 1_1
  S2: out = 8_16
  S2: rst = 1_1

---> STATE 3 <---
  S3: clk = 1_1
  S3: out = 8_16
  S3: rst = 1_1

---> STATE 4 <---
  S4: clk = 0_1
  S4: out = 8_16
  S4: rst = 0_1

---> STATE 5 <---
  S5: clk = 1_1
  S5: out = 9_16
  S5: rst = 0_1
```

We can notice that state 5 differs between traces in Figures 15 and 16. In fact CoSA does not guarantee a deterministic result in terms because the internal solver can produce different models according with the state space exploration that has been performed.

The parameter `--prefix` allows the user to save the trace to file.

**Example 7.3** (VCD trace generation)**.** Running `CoSA -i examples/counters_4/counters_4.v[Counters_4] --verification simulation -k5 --vcd --prefix trace` will generate a system execution of length equal to 6, and save it to a file with prefix "trace", as shown in Figure 17. A VCD trace is also generated when `--vcd` is set.

Figure 17: Simulation Counter_4 (with prefix)

```
1  ** Problem simulation **
2  Result: TRUE
3  Executions: [1], [2]
4  Traces (max) length: 6
5
6  *** TRACES ***
7
8  [1]:    trace[1]-simulation.txt
9  [2]:    trace[2]-simulation.vcd
```

### 7.1.2 Infinite traces

The counterexample showing the violation of an LTL property usually requires an infinite trace. For instance, if the property $F(\varphi)$ does not hold it means that the system can start in $\neg\varphi$, and never reach a state where $\varphi$ holds. A trace that will show this behavior is usually represented with a lasso-shaped loop. An example is shown in Figure 18, where the system can loop forever from `STATE 1` to `INIT` to `STATE 1`, and so on.

**Example 7.4** (Infinite trace (LTL))**.** Running `CoSA -i examples/counter/counter.json,examples/counter/counter_live.sts --verification ltl -a "en_clr = 0_1 -> self.clr = 0_1" -p "F(self.out = 4_16)" --trace-vars-change` will generate a counterexample trace for the property `F(self.out = 4_16)`, as shown in Figure 18.

Figure 18: Simulation Counter_4 (with changing values)

```
1  ** Problem ltl **
2  Result: FALSE
3  Counterexample:
4  ---> INIT <---
5    I: self.clk = 1_1
6    I: self.clr = 0_1
7    I: self.out = 0_16
8
9  ---> STATE 1 <---
10   S1: self.clk = 1_1
11   S1: self.clr = 0_1
12   S1: self.out = 0_16
13
14 ---> INIT (Loop) <---
```

# 8 Good practice

This section provides an high-level guidance for the users that approach for the first time model checking and formal verification.

- *Make sure that the formalization correctly represents what you want to verify.*

  This might seem a trivial statement, but is the most common mistake when verifying a system. An example could be that the translation of "if the output value is $> 10$, the reset can be triggered" to $(output > 10) \rightarrow reset$. However, this forces the reset to happen when output $= 11$, but the actual translation should be $reset \rightarrow (output > 10)$.

- *Be aware of property semantics*

  This is related to the previous point, but it assumes that the property correctly represents the intended behavior. An example of possible misleading results for not considering the property semantics could be that $\alpha \rightarrow \gamma$ holds because $\alpha$ never happens.

  For certain properties, such as implications, make sure that the pre-conditions can happen, for instance checking that the invariant $\neg\alpha$ is *False*.

- *Be aware of verification semantics*

  Most formal verifications have a "hidden" quantification in the problem that they solve. In case of safety, the check verifies if the property $\varphi$ holds for every reachable states of the system. However, the set of reachable states can be reduced if the system reaches a deadlock state, or it does not have all the behaviors that you expect.

  As an example, we might want to verify that a counter never reaches the value 10, and we expect it to count from 0 to 9 for each posedge and then it resets to 0. The safety verification of the property $output < 10$ is *True*. However, the model checker can return this result also if the counter always resets at the value 2, or if it deadlocks before reaching 10.

  In this case, a simulation of depth 30 can rule out some unwanted behaviors, especially deadlocks.

- *Gradually increase the verification complexity.*

  Sometimes the formal verification might be an expensive analysis, and a solving a model checking problem might take hours, and can cause frustration when no results are provided. Moreover, even if a counterexample is produced, its complexity might be very hard to decipher.

  Proceed incrementally by reducing the size of the parameters in the model, analyzing trivial property first, and proceeding bottom up in the hierarchy is always a good practice to increase the confidence that the model is correct.

  Moreover, model checkers like CoSA can take advantage of the properties that have proven to be *True*, and this can improve the performance of the successive analyses.

# 9 Encodings

CoSA supports different encodings of the model and are oriented to improve the verification performance. Following the list of possible encoding flags:

- `boolean`: encodes single bits as Boolean instead of bit-vectors of size 1,
- `clock-abstraction`: performs a clock abstraction encoding, imposing to perform the operations on every transition instead of on either of the clock edge. This encoding might be unsound when the system has both posedge and negedge behaviors, meaning that *True* results might be spurious. This option has an effect only on CoreIR, Verilog, and SystemVerilog models,
- `symbolic-init`: it relaxes the constraints on the initial states, while keeping the in variants,
- `zero-init`: it forces every unassigned variables to 0 on the initial states.

# 10 Performance optimizations

This section lists the parameters that enable performance optimizations in CoSA. All optimizations are disabled by default, in order to provide the simplest possible behavior when no explicit options are provided.

## 10.1 Lemmas

Each verification accepts a list of lemmas with the parameters that are added as invariant to the system before performing the analysis, if the pass the induction check.

## 10.2 Strategy

CoSA supports different verification algorithms. The possible algorithms are listed in the CoSA helper under the `strategy` section. For not experienced users, we suggest to select one of the following:

- `AUTO`: automatic selection, this is the default option,
- `ALL`: it tries all equivalent algorithms in series,
- `MULTI`: used in combination with the parameter `-j`, it runs multiple algorithms in parallel.

Table 7 shows the algorithm selection with `AUTO` strategy, and the different algorithms use the following naming convention:

- BMC-FWD: Bounded Model Checking (BMC) with forward unrolling,
- BMC-BWD: Bounded Model Checking (BMC) with backward unrolling,
- BMC: general LTL-based Bounded Model Checking,
- K-IND: K-Induction,
- INT: Interpolation,
- K-LIVE: K-Liveness.

| Prove | Core Verification | Technique |
|---|---|---|
| Yes | Safety | BMC-FWD+K-IND |
| Yes | LTL | BMC+K-LIVE |
| No | Safety | BMC-FWD |
| No | LTL | BMC |

Table 7: `AUTO` configuration

Table 8 shows the CoSA behavior when using the `MULTI` strategy. In this case multiple engine instances are run according with the number of processes. This option can be set with the parameter `-j` (default configuration is the number of cores), and it requires to run the first `j` algorithms in the "techniques list" column.

| Prove | Core verification | Techniques list |
|---|---|---|
| Yes | Safety | [BMC-FWD+K-IND, INT, BMC-BWD, BMC-FWD] |
| Yes | LTL | [BMC+K-LIVE] |
| No | Safety | [BMC-FWD, BMC-BWD] |
| No | LTL | [BMC] |

Table 8: `MULTI` configuration

## 10.3 Assume if true

Each safety property (if the assumption is *True*) that hold in the model can be seen as a lemma, thus it can be safely added as an invariant to the model. This option is enabled with the parameter `--assume-if-true`. A (full) safety analysis is different from the induction performed for the lemmas, in fact this analysis is "cheaper" and it does not take into account reachable states.

### 10.4 Cone of Influence

The verification of a system usually covers different parts of the system description, and it might the case that a property does not require the entire model to be analyzed. The cone of influence (COI) is a technique that extracts only the part of the model that is relevant for a given property. This technique relies on the structure of system, to extract the relevant information regarding the variables dependencies. At the current stage, CoSA supports COI (with the parameter `--coi`) only for Verilog and SystemVerilog models.

### 10.5 Circuit Optimizing

A very similar option to Cone of Influence reduction is to use Yosys to optimize the circuit. This calls a sequence of carefully selected passes which can propagate constants, remove redundant or unnecessary signals, and overall shrink the model. Although this can drastically speed up verification, it can be confusing as parts of the circuit might disappear. In particular, any part of the circuit that is not within the fan-in of a primary output, or a marked signal, will be removed. Signals and modules can be marked with a Verilog attribute, `(* keep *)` to prevent removal. This optimization can be invoked with the `--opt-circuit` option at the command line, or `opt_circuit: True` in a problem file.

### 10.6 Files Caching

CoSA translates each given model into an internal representation that is similar to the STS. When managing complex models, such as Verilog descriptions, this translation requires a not trivial computation. For this reason, CoSA implements a caching mechanism to avoid to re-compute the models that have been previously encoded. The parameter `-c` enables the files caching feature, which stores in the directory `.CoSA` (located in the same directory as the input file) a cached file according with its md5 and its encoding options.

## 11 Debugging

The options shown by the CoSA helper do not expose the configurations that are usually used by a developer. For this reason, the parameter `--devel` enables a series of additional behaviors, including also a more detailed error printing with stack trace information. For instance, enabling the developer mode allows the user to produce smt2 tracing files for each call to the solver.

# References


[1] R. Daly. CoreIR: A simple LLVM-style hardware compiler. `https://github.com/rdaly525/coreir`, 2017.

[2] M. Gario and A. Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT)*, pages 373–384, 2015.

[3] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan. CoSA: Integrated Verification for Agile Hardware Design. In *Formal Methods in Computer-Aided Design, FMCAD 2018, Austin, Texas, USA, October 30 - November 2, 2018*. IEEE, 2018.

[4] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. BTOR2, BtorMC and Boolector 3.0. In *Computer Aided Verification - 30th International Conference, CAV 2018, Oxford, UK, July 14-17*, Lecture Notes in Computer Science. Springer, 2018.

[5] C. Wolf, J. Glaser, and J. Kepler. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.