Mikino: Induction for Dummies

Adrien Champion, Steven de Oliveira, and Keryan Didier

OCamlPro,
`name.last-name@ocamlpro.com`

Abstract

Mikino is a simple induction engine over transition systems. It is written in Rust, with a strong focus on ergonomics and user-friendliness. It is accompanied by a detailed tutorial which introduces basic notions such as logical operators and Smt solvers. Mikino and its companion tutorial target developers with little to no experience with verification in general and induction in particular. This work is an attempt to educate developers on what a *proof by induction* is, why it is relevant for program verification, why it can fail to (dis)prove a candidate invariant, and what to do in this case.

1 Introduction

The ambition of this work is to present Smt-based induction to developers with no background in formal verification. We hope to achieve this ambition thanks to our hands-on, novice-friendly tutorial and the efforts invested in making mikino as user-friendly and ergonomic as possible. Note that mikino is available both as a binary[1] and a library[2], which we think can motivate enthusiastic readers to try to build their own analyses.

Mikino is released under MIT/Apache 2.0 (library and binary), while the tutorial itself is under *CC BY-SA* (Creative Commons Attribution-ShareAlike). We hope this encourages teachers/trainers to use this work in relevant classes/training sessions.

The following sections discuss mikino's internals and ambitions. Appendix A goes over the process of installing mikino, and Appendix B illustrates mikino's input and output on three examples. We highlight perspectives for mikino in Appendix C.

2 Declarative Transition Systems

Mikino is essentially a much simpler version of the internal proof engine found, for instance, in the Kind 2 [2] model-checker. It analyzes *declarative transition systems*; such systems have a vector of typed state variables $\bar{v}$ that encapsulate the whole state of the system, *i.e.* no information relevant to the system exists outside of $\bar{v}$. For instance, $\bar{v}_c = [\texttt{count} : \texttt{int}, \texttt{reset} : \texttt{bool}]$. A state $\bar{s}$ is a valuation of the state variables, such as $\bar{s} = [7, \texttt{false}]$.

The *initial state(s)* of the system are specified by a *state predicate* $\texttt{init}(\bar{v})$: a formula over $\bar{v}$. Any state $\bar{s}$ such that $\texttt{init}(\bar{s})$ evaluates to $\texttt{true}$ is a legal initial state for the system. For instance, $\texttt{init}(\bar{v}_c) \equiv \texttt{count} \geq 0 \wedge (\texttt{reset} \Rightarrow \texttt{count} = 0)$. The *transition relation* $\texttt{trans}(\bar{v}, \bar{v}')$ is a formula over unprimed (*current*) and primed (*next*) state variables. State $\bar{s}'$ is a legal successor of $\bar{s}$ for the system if and only if $\texttt{trans}(\bar{s}', \bar{s})$ evaluates to $\texttt{true}$. For instance,

$$\texttt{trans}(\bar{v}_c, \bar{v}_c') \quad \equiv \quad \texttt{count}' \; = \; \textbf{if}\ \texttt{reset}'\ \{0\}\ \textbf{else}\ \{\texttt{count} + 1\}$$

where the *next* value of $\texttt{count}$ is 0 if $\texttt{reset}'$, and its old value plus one otherwise. The next value of $\texttt{reset}$ is not constrained at all.

[1] `https://github.com/OCamlPro/mikino_bin`
[2] `https://github.com/OCamlPro/mikino`

Last, mikino expects some *candidate properties* or *candidate invariants* which are just called *candidates*. A candidate is a predicate over $\bar{v}$. These candidates are named and mikino uses the name provided when producing feedback. For instance,$\texttt{candidate}_1(\bar{v}_c) \equiv \texttt{cnt} \geq 0$ could be a candidate for our running example.

3 Analysis

Mikino's analysis relies heavily on SMT solvers [4], and more precisely on Z3 [3]. A *proof by induction* over a transition system $(\bar{v}, \texttt{init}, \texttt{trans})$ of a candidate `candidate` consists in showing two things:

- $\texttt{init}(\bar{v}) \land \neg\texttt{candidate}(\bar{v})$ is unsat,
  thus proving that $\texttt{init}(\bar{v}) \Rightarrow \texttt{candidate}(\bar{v})$;
- $\texttt{candidate}(\bar{v}) \land \texttt{trans}(\bar{v}, \bar{v}') \land \neg\texttt{candidate}(\bar{v}')$ is unsat,
  thus proving that $\texttt{candidate}(\bar{v}) \land \texttt{trans}(\bar{v}, \bar{v}') \Rightarrow \texttt{candidate}(\bar{v}')$.

If both these formulas are unsat, then `candidate` is an invariant for the system by induction.

Now, if the `init` check fails, we can extract a model from the solver. This model, by construction, falsifies `candidate`: the candidate can be falsified by at least one of the initial states. This is a concrete counterexamples as discussed previously.

If the `trans` check fails however, it only means that `candidate` is not preserved by the transition relation. This result says nothing on whether `trans` can reach a state falsifying `candidate`. The best we can do is to extract a model, which in this case corresponds to a pair of states $(\bar{s}, \bar{s}')$ such that *i)* $\bar{s}$ verifies the candidate, *ii)* $\bar{s}'$ is a legal successor of $\bar{s}$, and *iii)* $\bar{s}'$ falsifies the candidate.

One of the main goals of the mikino tutorial is to bring its intended audience of verification-agnostic developers to fully understand what it means for a `trans` check to fail, and what the model extracted corresponds to. Mikino itself goes to great length to have as readable and user-friendly an output as possible, hopefully making things easier for readers going through the tutorial.

The second main goal of the tutorial is to teach readers how to react to a failed `trans` check. *Property strengthening* —or *candidate strengthening*, here— consists in adding *information* to a candidate when it is not inductive. This consists in adding new candidates that act as lemmas so that the original candidate *in conjunction with the lemmas* is inductive. The pair of succeeding states returned by failed `trans` checks is helpful in deciding what lemma should be added, as long as users have an understanding of what these states represent.

Strengthening can only succeed if the candidate is indeed an invariant (although a non-inductive one) for the system. Bugs happen however, and mikino can find them thanks to its Bounded Model-Checking (BMC) feature. We do not discuss BMC further here, but note that one of the mikino runs shown in Appendix B discusses BMC and uses it to find counterexamples.

4 Conclusion

Mikino was created for verification novices, typically average developers, and is thus focused on user-friendliness and ergonomics. Its companion tutorial introduces SMT solvers, declarative transition systems, BMC, induction, and candidate strengthening using simple terms and examples that readers can run locally, modify, and rewrite. Mikino is open source and available both as a binary and a library, which encourages experimentation beyond the relatively small scope of its companion tutorial.

References

[1] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[2] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The kind 2 model checker. In *CAV (2)*, volume 9780 of *Lecture Notes in Computer Science*, pages 510–517. Springer, 2016.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[4] Cesare Tinelli. Foundations of satisfiability modulo theories. In *WoLLIC*, volume 6188 of *Lecture Notes in Computer Science*, page 58. Springer, 2010.

A Installing Mikino

Mikino requires the Z3 Smt solver to run. In practice, this means retrieving a Z3 binary as discussed in the tutorial:

`https://ocamlpro.github.io/verification_for_dummies/smt/index.html#z3`

The easiest way to do this is on Z3's release page:

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

By default, mikino assumes the Z3 binary is in your path and called `z3`. You can use mikino's `--z3_cmd` to specify a different name or path. Refer to `mikino help` for details.
We recommend you refer to tutorial's instructions for installing mikino:

`https://ocamlpro.github.io/verification_for_dummies/mikino_bmc/index.html`

which, unlike what follows, are kept updated. The easiest way to obtain the latest mikino binary is to go to its release page:

`https://github.com/OCamlPro/mikino_bin/releases`

At the time of writing, the latest version is `0.5.2`.

Alternatively, if Rust (`https://www.rust-lang.org`) is installed on your machine, you can run `cargo install mikino` which will compile mikino and put it in you path. In case you want to update to the latest version, run `cargo install --force mikino`.

B Examples: Proof Failure/Success and Bmc

Let us go back to the example system used in the main body of this article. It is defined by its state variables $\bar{v}$, its initial predicate `init`, and its transition relation `trans`. We also define two candidates $\texttt{candidate}_1$ and $\texttt{candidate}_2$:

$$\begin{aligned} \bar{v} &\equiv [\texttt{count} : \texttt{int}, \texttt{reset} : \texttt{bool}] \\ \texttt{init}(\bar{v}) &\equiv \texttt{count} \geq 0 \wedge (\texttt{reset} \Rightarrow \texttt{count} = 0) \\ \texttt{trans}(\bar{v}, \bar{v}') &\equiv \texttt{count}' \ = \ \textbf{if } \texttt{reset}' \ \{0\} \textbf{ else } \{\texttt{count} + 1\} \\ \texttt{candidate}_1(\bar{v}) &\equiv \neg(\texttt{count} = -7) \\ \texttt{candidate}_2(\bar{v}) &\equiv \texttt{reset} \Rightarrow \texttt{count} = 0 \end{aligned}$$

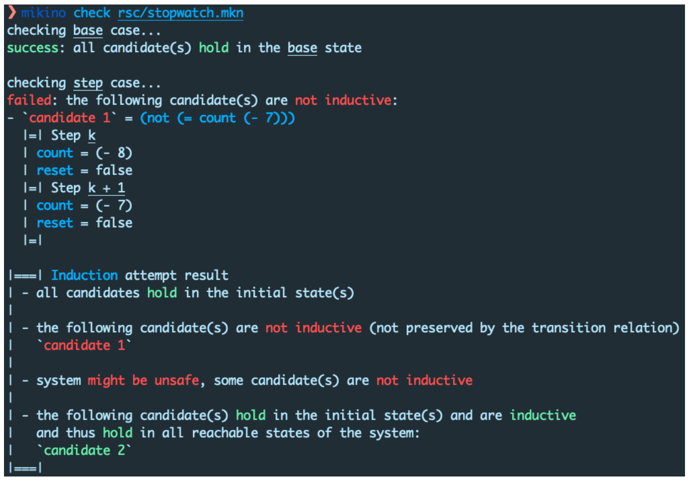
The equivalent in mikino's input format is

```
svars {
    count: int,
    reset: bool,
}
init {
    count ≥ 0,
    reset ⇒ (count = 0),
}
trans {
    'count = if 'reset { 0 } else { count + 1 },
}
candidates {
    "candidate 1": ¬(count = -7),
    "candidate 2": reset ⇒ (count = 0),
}
```

Note that:

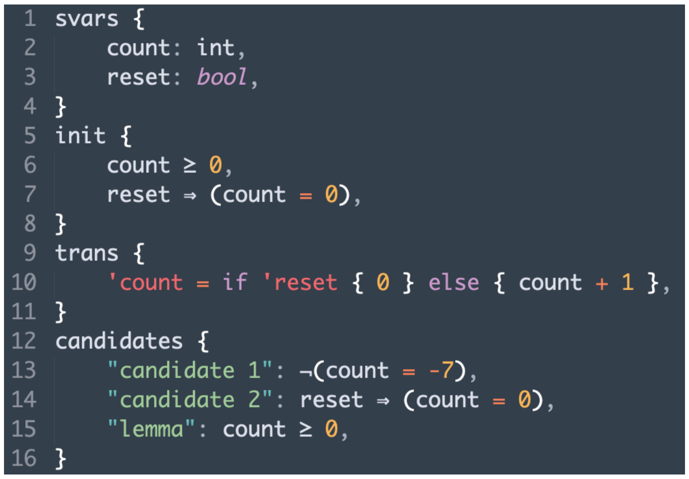
- *primed* variables have their prime *before* the identifier, not after. This is because we want to use Rust syntax highlighting: in Rust, `'<ident>` is the syntax for *lifetimes* which make `'count` and `'reset` pop out. On the other hand, `<ident>'` would be interpreted by syntax highlighting as an identifier followed by the start of a character literal.
- mikino's Rust expressions are more readable than SMT-LIB's S-expressions.
- mikino supports various UTF-8 operators in addition to their usual ASCII equivalent(s) `>=`, `=>`, `&&`, `||`, `!`, *etc.*
- the `init` and `trans` sections take a list of comma-separated expressions (with optional trailing comma), understood as a conjunction.

Running mikino on this systems yields the following. (Run `mikino help` for details on mikino's CLI.)



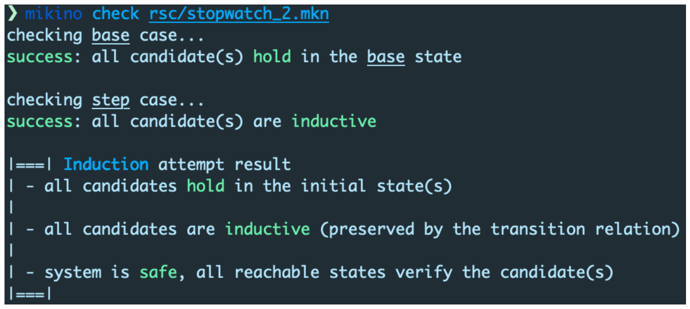
```
❯ mikino check rsc/stopwatch.mkn
checking base case...
success: all candidate(s) hold in the base state

checking step case...
failed: the following candidate(s) are not inductive:
- `candidate 1` = (not (= count (- 7)))
  |=| Step k
  | count = (- 8)
  | reset = false
  |=| Step k + 1
  | count = (- 7)
  | reset = false
  |=|

|===| Induction attempt result
| - all candidates hold in the initial state(s)
|
| - the following candidate(s) are not inductive (not preserved by the transition relation)
|   `candidate 1`
|
| - system might be unsafe, some candidate(s) are not inductive
|
| - the following candidate(s) hold in the initial state(s) and are inductive
|   and thus hold in all reachable states of the system:
|   `candidate 2`
|===|
```

The first candidate is not inductive and needs some strengthening. Let us just add the very natural lemma `count` ≥ 0.



```
svars {
    count: int,
    reset: bool,
}
init {
    count ≥ 0,
    reset ⇒ (count = 0),
}
trans {
    'count = if 'reset { 0 } else { count + 1 },
}
candidates {
    "candidate 1": ¬(count = -7),
    "candidate 2": reset ⇒ (count = 0),
    "lemma": count ≥ 0,
}
```

Running mikino again, we see that the strengthening was successful and mikino is able to prove all three candidates.

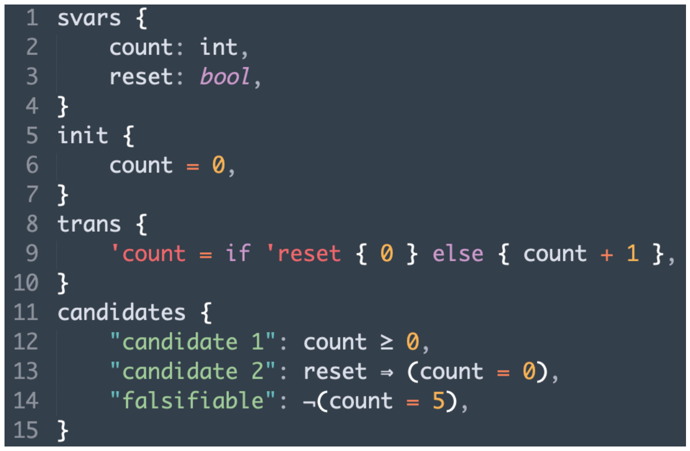


```
❯ mikino check rsc/stopwatch_2.mkn
checking base case...
success: all candidate(s) hold in the base state

checking step case...
success: all candidate(s) are inductive

|===| Induction attempt result
| - all candidates hold in the initial state(s)
|
| - all candidates are inductive (preserved by the transition relation)
|
| - system is safe, all reachable states verify the candidate(s)
|===|
```

Last, here is an example of asking mikino to check a falsifiable candidate by BMC. Roughly, BMC is an iterative process that starts by looking for a falsification of the candidate(s) in the initial state, exactly like induction's `init` check. If none exists, BMC asks the same questions about the successors of the initial states. More precisely, is

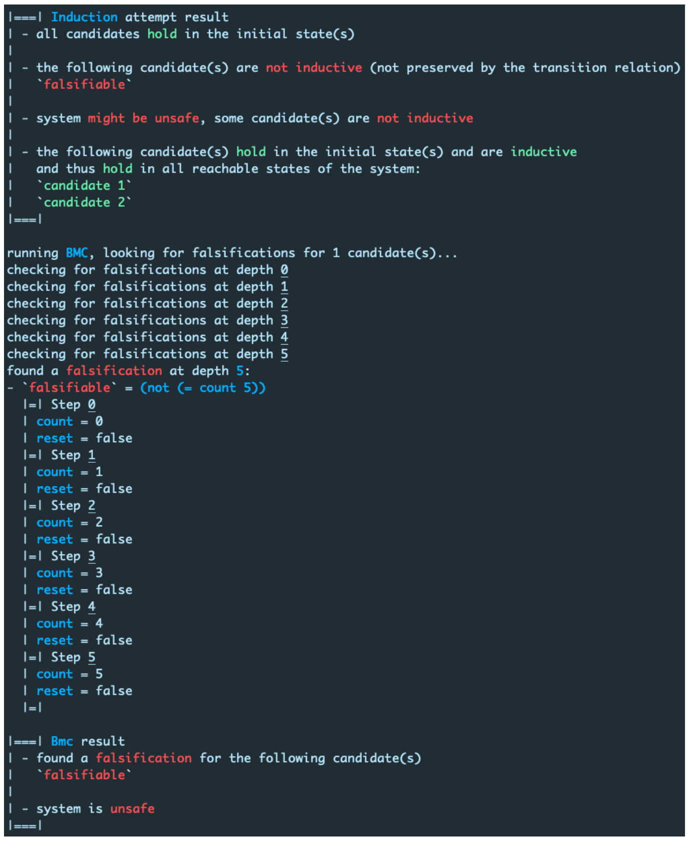
$$\texttt{init}(\bar{v}_0) \land \texttt{trans}(\bar{v}_0, \bar{v}_1) \land \neg\texttt{candidate}(\bar{v}_1)$$

satisfiable? If it is, BMC can extract a model corresponding to two succeeding states leading to a falsification of the candidate. If not, BMC *unrolls* the transition relation again to check the successors of the successors of the initial states.
To showcase BMC in mikino, let us modify `init` slightly and give ourselves a falsifiable candidate.



```
svars {
    count: int,
    reset: bool,
}
init {
    count = 0,
}
trans {
    'count = if 'reset { 0 } else { count + 1 },
}
candidates {
    "candidate 1": count ≥ 0,
    "candidate 2": reset ⇒ (count = 0),
    "falsifiable": ¬(count = 5),
}
```

In `check` mode, BMC is activated by passing the `--bmc` option to mikino: `mikino check --bmc <file>`. This option makes mikino run BMC on all non-inductive candidates. The second image only shows the relevant part of the output, after the `trans` check counterexample is reported.



```
|===| Induction attempt result
| - all candidates hold in the initial state(s)
|
| - the following candidate(s) are not inductive (not preserved by the transition relation)
|   `falsifiable`
|
| - system might be unsafe, some candidate(s) are not inductive
|
| - the following candidate(s) hold in the initial state(s) and are inductive
|   and thus hold in all reachable states of the system:
|   `candidate 1`
|   `candidate 2`
|===|

running BMC, looking for falsifications for 1 candidate(s)...
checking for falsifications at depth 0
checking for falsifications at depth 1
checking for falsifications at depth 2
checking for falsifications at depth 3
checking for falsifications at depth 4
checking for falsifications at depth 5
found a falsification at depth 5:
- `falsifiable` = (not (= count 5))
  |=| Step 0
  | count = 0
  | reset = false
  |=| Step 1
  | count = 1
  | reset = false
  |=| Step 2
  | count = 2
  | reset = false
  |=| Step 3
  | count = 3
  | reset = false
  |=| Step 4
  | count = 4
  | reset = false
  |=| Step 5
  | count = 5
  | reset = false
  |=|

|===| Bmc result
| - found a falsification for the following candidate(s)
|   `falsifiable`
|
| - system is unsafe
|===|
```

C Perspectives

Since the first chapters of the tutorial go over SMT checks, we rely on the SMT-LIB standard [1] to write the examples. SMT-LIB is based on S-expressions as a middle ground between human-readability and ease of parsing. While very appropriate as a common input language for SMT solvers and for writing/sharing benchmarks (its main purposes), it can be deterring for novices just starting out: $x + 1$ for instance would be written `(+ x 1)`. Mikino's syntax for transition systems is based on Rust and is much more natural to uninitiated readers.

Depending on the feedback on mikino and its tutorial, we would like mikino to be able to act as a (thin, potentially interactive) frontend for SMT solvers by accepting Rust-style expressions. This would remove the need for showing SMT-LIB code at all and, we think, make it easier for novices to dive into formal verification.