Loustrini: A Lustre Model Checker using (H-)Houdini Invariant Learning Algorithm

Or me learning about invariant learning algorithms

Rémi Germe

09.01.2026

Contents

First experiments with SMT solvers

First experiments with SMT solvers

Goal: SMT-solver-agnostic model checker $\Rightarrow$ use a frontend library $\Rightarrow$ in OCaml: `Smt.ml` [1]

Aftermath of these initial experiments:

- discovered unsound behavior in Bitwuzla mappings of `Smt.ml` (see issue #465)
- discovered bug in AltErgo [2] support of `Smt.ml` (see discussion #450)
- more generally, clarification of the current limitations of `Smt.ml` (also discussion #450)
- re-discovered a known issue in AltErgo failing to conclude `SAT` (see issue #1323)

$\Rightarrow$ Loustrini is *not* solver-agnostic (using `Z3` [3] - best would have been `cvc5` [4] for *abducts*).

Translating Lustre to SMT expressions

Encoding *à la* k-induction

Encoding presented in the handout (from [5]).

$\boldsymbol{\Delta(n)}$ encodes the system equations at time n.

Checking a property P is inductive on system Δ:

$$\begin{array}{ll} \text{(initiation)} & \Delta(0) \Rightarrow P(0) \\ \text{(consecution)} & \Delta(n) \wedge \Delta(n+1) \wedge P(n) \Rightarrow P(n+1) \end{array}$$

Wrong semantics for $\rightarrow$

I implemented a wrong semantics for $e \rightarrow e'$:

$1 \rightarrow 2 \rightarrow 3$ generates $\begin{cases} 1 \text{ if } n{=}0 \\ 2 \text{ if } n{=}1 \\ 3 \text{ if } n{\geq}2 \end{cases}$ instead of $\begin{cases} 1 \text{ if } n{=}0 \\ 3 \text{ if } n{\geq}1 \end{cases}.$

Properties refering to n and $n+1$

They require careful thinking:

$$\underbrace{\Delta(n) \land \Delta(n+1)}_{\text{do not constrain } x(n+2)} \land P(n) \Rightarrow \underbrace{P(n+1)}_{\text{refers to } x(n+2)} \; ? \;\; \text{will fail, so still sound}$$

$\Rightarrow$ would require to strenghten our lhs with $\Delta(n+2)$

I did **not** consider those properties.

Encoding as a transition system

Transition system (I, T) using state variables and primed variables, verifying a property P:

$$I \Rightarrow P \text{ and } P \wedge T \Rightarrow P'$$

Handling $\texttt{pre}$ e: introduce a new state variable $S^{\texttt{pre}}_{\text{id}} = \{\texttt{init} = \emptyset; \quad \texttt{next} = [\![e]\!]\}$

Handling of $e \rightarrow e'$: use $\text{ite}(S^{\rightarrow}_i, [\![e]\!], [\![e']\!])$ with:

- $S^{\rightarrow}_0 = \{\texttt{init} = \text{true}; \quad \texttt{next} = \text{false}\}$
- $S^{\rightarrow}_1 = \{\texttt{init} = \text{false}; \quad \texttt{next} = S^{\rightarrow}_0\}$
- $S^{\rightarrow}_2 = \{\texttt{init} = \text{false}; \quad \texttt{next} = S^{\rightarrow}_1\}$
- ...

We also need to make sure we have **at most one** of the $S^{\rightarrow}_i$ to be true (+ harder to obtain positive examples).

⚠️ $$\cancel{\Delta(n) \wedge} \; \Delta(n+1) \wedge P(n) \Rightarrow P(n+1)$$

Even less precise, the consecution might fail because of **spurious counterexamples** (unreachable situations), already the case before but less likely.

Encoding non trivial constructs

Tuples.

Treat a n-tuple as n-expressions and translate each member separately.

Node calls.

Instantiation (inlining) of nodes at call site.

How to learn efficiently invariants for a node (and not just for its instances)?

In real-world projects (see discussion #1256 and [6], [7]):

- compositional reasoning
- modular reasoning
- progressive refinements
- …

This issue will **not** be addressed in the project.

Invariant learning algorithms, Houdini, H-Houdini

Overview of invariant learning algorithms

Goal: prove a safety property P. Let's not use *k-induction* but *invariant learning* instead.

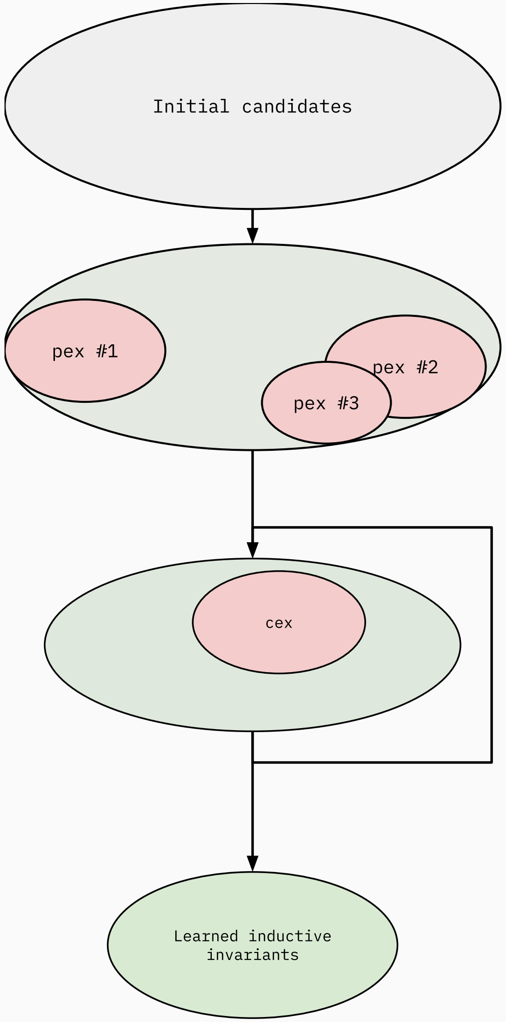
Challenge: find a property H such that:

$$\begin{aligned} &\text{(initiation)} & I &\Rightarrow H \\ &\text{(consecution)} & H \wedge T &\Rightarrow H' \\ &\text{(implies desired property)} & H &\Rightarrow P \end{aligned}$$

How to find such an H?

Bottom-up approaches	**Top-down** approaches (property-directed)
Learn as many invariants as possible	Learn relevant invariants to prove a given desired property
• Houdini [8] • Instantiation-based invariant discovery [7]	• IC3 [9] • H-Houdini [10]

Houdini: overview [7], [8]



- Generating a lot of candidates using templates (see next slide)
- Initial sift: removing all candidates that do no satisfy a trace of execution (see next next slide)
- Inductivity sift: removing all candidates that break inductivity, and iterate until reaching an inductive set (see next next slide)
- Final learned inductive invariants

Houdini: generating invariants

Variables below are all evaluated at n (n and $n-1$ (+ init to true) would have been possible).

Booleans. $\mathcal{I}_b ::= b = \text{true} \mid b = \text{false} \mid b_1 = b_2 \mid b_1 = \neg b_2$

Integers. $\mathcal{I}_i ::= i \diamond \text{cst} \mid i_1 \diamond i_2$

Reals. Same as for integers.

Where:

- $\text{cst} \in \{0, 1, -1\} \cup \{\text{constants of interest (hardcoded in the program)}\}$
- $\diamond ::= \ \geq \mid > \mid \leq \mid < \mid = \mid \neq$

We obtain a set of candidates $H = \bigwedge_{i=1}^{d} h_i$.

SOUND but absolutely not **COMPLETE**

but Houdini is complete *relative* to the templates

Houdini: sifting candidates

Inductivity check.

$$\Delta(n) \wedge \Delta(n+1) \wedge H(n) \wedge \neg H(n+1) \ ?$$

$$\texttt{UNSAT} : H \text{ is inductive OR } H(n) \text{ contradicts } \Delta(n) \wedge \Delta(n+1)$$

$$\texttt{SAT} : \exists \text{ cex}, \begin{cases} \text{cex} \vDash H(n) \\ \text{cex} \vDash \neg H(n+1) \end{cases} i.e.\ \exists i \in \{1, \ldots, d\}, \underbrace{\textbf{cex} \vDash \neg \boldsymbol{h_i(n+1)}}_{\text{remove all these candidates}}$$

Better than checking each h_i separately.

Initial sift.

1. Prune the search space by removing many candidates that do not hold.
2. Ensure H is non-contradictory (vacuously false).

$$\Delta(0) \wedge \ldots \wedge \Delta(k) \wedge \neg H(k)$$

$$\texttt{UNSAT} : H \text{ is consistent with step } k$$

$$\texttt{SAT} : \exists \text{ cex}, \ldots \text{ (similar as above, and we keep iterating for this } k\text{)}$$

Houdini: demo

Extra-prunning of “obvious” invariants.

ic3.lus

```
x = 1 -> pre x + 1;
y = 1 -> pre (x + y);
ok = y >= 0;
```

Not k-inductive!

ic_more.lus

Would fail with naive validation of candidates.

fib.lus

With/without explicit state variables.

Limitation: we cannot learn an invariant not present in the templates.

H-Houdini: overview [10]

Motivation.

- candidates generation is subject to combinational explosion,
- sifting may require *many*, *large* SMT queries.

H-Houdini ("Hierarchical Houdini") aims at solving these by making Houdini **property-directed**.

```python
def h-houdini(p_target):
  V = SLICE(p_target) # extract relevant variables
  P = MINE(p_target, V) # generate candidates
  while # flag to keep iterating:
    A = ABDUCT(p_target, P) # extract a proposition that fixes (makes inductive) p_target
    if A is None: return None
    H = p_target
    for p in A:
      h_sol = h-houdini(p)
      if h_sol is None: # break, keep iterating the while loop
      H = H ^ h_sol
    return H
```

H-Houdini: slicing and mining

Slicing.

- `SLICE(p_target)` extracts the variables that influence the inductivity of p_{target}
- implemented by instrumenting the Lustre $\rightarrow$ SMT translation

Mining.

- `MINE(p_target, V)` generates invariants according to some templates
- implemented by reusing functions for Houdini - not ideal as they do not take p_{target} as an input and so subsequent sifting is performed

H-Houdini: abducting

`ABDUCT(p_target, P)` returns `None` or a list of predicates A derived from P that fixes p_{target}: $A \land p_{\text{target}} \Rightarrow {p_{\text{target}}}'$. If called multiple times, it returns a different abduct each time.

The paper proposes the following method:

$$\bigwedge_i P_i \land p_{\text{target}} \land \neg {p_{\text{target}}}'$$
$$\texttt{UNSAT} : \text{we extract a minimal unsat core } A$$
$$\texttt{SAT} : \text{no possible abduct using } P$$

But I fail to see how this method can generate several *different* abducts.

Using SMT solvers:

- ▷ to the best of my knowledge, Z3 does not provide any builtin method
- ▷ cvc5 has a builtin `getAbductNext()` method but no official OCaml bindings

Also: Z3 `get_unsat_core` returns a list, but in practice it is only one element which is a big conjunct (not handy).

The implementation is currently bugged.

Future work & conclusion

Future work & conclusion

Future work

“Minor” improvements for H-Houdini:

- mining according to p_{target}
- topological ordering of equations for slicing
- memoization

Major issue: abducting.

Key insights

- Houdini works!
- bottom-up vs top-down (property-directed) approaches
- limited to templates (see IC3)
- in real world: used as complement to other techniques
- in real world: modular reasoning

References

[1] J. M. Pereira *et al.*, "Smt.ml: A Multi-Backend Frontend for SMT Solvers in OCaml," 2024.

[2] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout, "Alt-Ergo 2.2," in *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018.

[3] L. De Moura and N. Bjrner, "Z3: an efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, in TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340.

[4] H. Barbosa *et al.*, "cvc5: A Versatile and Industrial-Strength SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, D. Fisman and G. Rosu, Eds., in Lecture Notes in Computer Science, vol. 13243. Springer, 2022, pp. 415–442. doi: 10.1007/978-3-030-99524-9_24.

[5] G. Hagen and C. Tinelli, "Scaling up the formal verification of Lustre programs with SMT-based techniques," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, in FMCAD '08. Portland, Oregon: IEEE Press, 2008.

[6] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, "The Kind 2 model checker," in *International Conference on Computer Aided Verification*, 2016, pp. 510–517.

[7] T. Kahsai, Y. Ge, and C. Tinelli, "Instantiation-Based Invariant Discovery," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 192–206.

References

[8] C. Flanagan and K. R. M. Leino, "Houdini, an Annotation Assistant for ESC/Java," in *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, in FME '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 500–517.

[9] A. R. Bradley, "SAT-based model checking without unrolling," in *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, in VMCAI'11. Austin, TX, USA: Springer-Verlag, 2011, pp. 70–87.

[10] S. Dinesh, Y. Zhu, and C. W. Fletcher, "H-Houdini: Scalable Invariant Learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 603–618.

[11] N. Halbwachs and P. Raymond, "A tutorial of lustre," *Lustre V4, January*, 2002, [Online]. Available: https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v4/distrib/lustre_tutorial.pdf

Initial fiddling with SMT solvers: encoding programs

Two encodings of the same Lustre program:

```
x = 1 -> pre x + 1;
y = 1 -> pre (x + y);
ok = y >= 0;
```

```smt2
(define-fun-rec x_naive ((n Int)) Int
    (ite (= n 0) 1 (+ (x_naive (- n 1)) 1)))
(define-fun-rec y_naive ((n Int)) Int
    (ite (= n 0) 1 (+ (y_naive (- n 1)) (x_naive (- n 1)))))
(define-fun ok_naive ((n Int)) Bool
    (>= (y_naive n) 0))

(declare-const n Int)

(echo "consecution: unsat iff ok(n) => ok(n+1) is true:")
(assert (and (>= n 0) (ok n) (not (ok (+ n 1)))))
(check-sat)
```

Z3 **timeouts** without providing a counterexample.

```smt2
(define-fun init ((x Int) (y Int)) Bool (and (= x 1) (= y 1)))

(define-fun trans ((x Int) (y Int) (nx Int) (ny Int)) Bool
    (and (= nx (+ x 1))
         (= ny (+ y x))))

(define-fun ok ((y Int)) Bool (>= y 0))

(declare-const x Int)
(declare-const y Int)
(declare-const nx Int)
(declare-const ny Int)

(echo "consecution: unsat iff ok is inductive:")
(assert (and (ok y)
             (trans x y nx ny)
             (not (ok ny))))
(check-sat)
```

Z3 **immediately answers** `SAT`, providing a counterexample.

Lustre ⇒ SMT: tuples

Two possible solutions:

- define tuple sorts directly in `Z3`'s logic
- **treat a n-tuple as n expressions**: return a list of expr defining each member of the tuple.

```ocaml
let rec compile_expr
  (ctx   : context   )
  (env   : z3_env_t )
  (n     : Expr.expr)
  (n_pre : int       )
  (n_arr : int       ) : Expr.expr list = ...
```

Remarks:

- nested tuples are flattened
- we considered **`if`** to be the only polymorphic operator [11]
  (other operators such as =, +, … do not support tuples)

Lustre ⇒ SMT: function calls

Problem:

```lustre
node incr(x: int) returns (y: int);
var l: int;
let
    l = x + 1;
    y = x + l;
tel

node compute(a, b: int) returns (c: int);
var z, t: int;
let
    z = incr(a);
    t = incr(b);
    b = z + t;
tel
```

We can't just have

$$\text{(incr def)} \quad \begin{cases} l(n) = x(n) + 1 \\ y(n) = x(n) + l(n) \end{cases}$$

$$\text{(incr calls)} \quad \begin{cases} x(n) = a(n) \\ z(n) = y(n) \\ \color{red}{x(n) = b(n)} \\ \color{red}{t(n) = y(n)} \end{cases}$$

We need to resort to **instantiating** the node at each call site (*i.e.* perform inlining).

(And that was a very painful rabbit hole, because I wanted definitions to be functions of n: `expr -> expr`, instead: use substitutions.)

Lustre ⇒ SMT: function calls

$$\text{(incr call 1)} \begin{cases} x_a(n) = a(n) \\ l_a(n) \ = x_a(n) \\ y_a(n) = x_a(n) + l_a(n) \\ z(n) \ \ = y_a(n) \end{cases} \qquad \text{(incr call 2)} \begin{cases} x_b(n) = b(n) \\ l_b(n) \ = x_b(n) \\ y_b(n) = x_b(n) + l_b(n) \\ t(n) \ \ = y_b(n) \end{cases}$$

Problem: we now have to learn invariant for **each instance** (separately in the worst case).

How to learn efficiently invariants for a node (and not just for its instances)?

In real-world projects (see discussion #1256 and [6], [7]):

- compositional reasoning
- modular reasoning
- progressive refinements
- …

This issue will **not** be addressed in the project.

Positive examples for transition system

Strategy: $(C \land T \Rightarrow C') \iff C \land T \land \neg C'$ to see if C is inductive (iff query `UNSAT`).

Problem: **if C is `UNSAT`** alone (e.g. $C = x \geq 0 \land x < 0$), we **cannot refine** C any further.

$\Rightarrow$ [7], [10]: use **positive examples** (concretely: some traces of the program) to first sift C

Problem: initial state alone is not sufficient: `pre` variables are not initialized

$x = 0 \to 1 \to \text{pre pre } x$

- $x = \text{ite}(S_0^{\to}, 0, \text{ite}(S_1^{\to}, 1, S_1^{\text{pre}}))$
- $S_1^{\text{pre}\prime} = S_0^{\text{pre}}$
- $S_0^{\text{pre}\prime} = x$

t	x	$S_0^{\to}$	$S_1^{\to}$	S_1^{pre}	S_0^{pre}
0	0	true	false	$\varnothing$	$\varnothing$
1	1	false	true	$\varnothing$	0
2	0	false	false	0	1

$\Rightarrow$ simulate the program for k steps (where k denotes the max depth of `pre` statements): *difficult* with this encoding