Loustrini: a Lustre model checker using the (H-)Houdini algorithm

Rémi Germe

05.01.2026

This short report was written as part of my project for the SYNC class at MPRI. Please refer to the repository[1] for the latest version of the project.

I Introduction

Real-world model checkers such as Kind2 [1] use several invariant learning algorithms [2], and also pair them with other techniques such as k-induction [3]. On the other hand, Loustrini solely relies on invariant learning algorithms.

When trying to prove a property with Loustrini, it will attempt to learn invariants that imply the desired property. If such inductive invariants are discovered, the property holds. Else, Loustrini fails to prove that the property holds.

My work consisted in:
- reading about model checking of Lustre programs,
- reading about invariant learning algorithms such as Houdini [2], [4] and H-Houdini [5],
- implementing a prototype model checker named Loustrini.

The rest of the report is structured as follows: I first present some initial experiments I made with SMT solvers, (re)discovering several bugs. Next, I detail the translation from a Lustre program to SMT expressions. I then describe the invariant learning algorithms powering Loustrini: Houdini and H-Houdini. Finally, I provide a very limited evaluation of Loustrini.

II Initial experiments with SMT solvers

The original skeleton of the project was using Alt-Ergo Zero [6], a solver derived from an old version of Alt-Ergo [7]. I wanted Loustrini to be solver-agnostic and be able to use different SMT solvers interchangeably.

The only OCaml SMT frontend I was able to find is Smt.ml [8]. I discovered some issues/limitations (unsound behavior in Bitwuzla mappings[2], bug in Alt-Ergo support, and limitations of uninterpreted functions[3]) which led me to believe the library was not mature enough yet.

Then, I considered several solvers individually:
- Alt-Ergo, which sometimes answers `UNKNOWN` when the expected answer clearly is `SAT`[4]. This issue could be addressed, as models are still generated, but this discouraged me from using Alt-Ergo.
- cvc5 [9], which whould have been ideal as it natively provides an abduct operation (see Section V.4), but cvc5 unfortunately does not officialy provide OCaml bindings.
- Z3 [10], which was eventually designated as the underlying solver for Loustrini.

III Translating Lustre to SMT expressions

There is two possible encodings for Lustre programs: *à la* k-induction or as a transition system. Both were implemented, but the k-induction encoding was eventually kept for reasons detailed below.

[1] https://github.com/remigerme/loustrini

[2] See https://github.com/formalsec/smtml/issues/465.

[3] See https://github.com/formalsec/smtml/discussions/450.

[4] See https://github.com/OCamlPro/alt-ergo/issues/1323.

À la k-induction [2], [3]. This is the encoding mentioned in the project description: our Lustre program is encoded by a function $\Delta : \mathbb{N} \to \texttt{Expr}$. We can check that a property $P : \mathbb{N} \to \texttt{Expr}$ is inductive with the following queries:

$$\begin{aligned} &(\text{initiation}) && \Delta(0) \Rightarrow P(0) \\ &(\text{consecution}) && \Delta(n) \wedge \Delta(n+1) \wedge P(n) \Rightarrow P(n+1)^{5} \end{aligned}$$

As a transition system [5]. Our program is given by a pair (I, T) where I encodes the initial state of our program, and T encodes the transition relation to go from one step to the next. We do not have an explicit n: $x(n)$ becomes x and $x(n+1)$ becomes x'. Note that, for each delay $\to$ and `pre` operator, we introduce a state variable. With this encoding, the queries for checking inductivity are:

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

Although the transition system encoding might seem conceptually simpler, I found it far less convenient to work with in practice when trying to compute positive examples, *i.e.* a trace of execution of the program (see Section IV.2.2).

On properties refering to multiple steps. Let's consider a property $P(n) = x(n+1) \geq x(n)$. $P(n+1)$ depends on $x(n+2)$. Using our query presented above for checking inductivity, $x(n+2)$ does not appear in $\Delta(n) \wedge \Delta(n+1)$, so the inductivity will fail. If we want such a property to have a chance to pass the test, we also need to consider $\Delta(n+2)$. I chose not to consider such properties for the rest of the project, but this could be a fairly easy extension.

Translating Lustre expressions to SMT expressions using either one of the above encoding is pretty straightforward except for tuples and node calls.

Translating tuples. I chose to flatten all tuples, that is translate and equate each member separately. For example, the Lustre statement $x, y = (e, d)$ is translated as $([\![x]\!] = [\![e]\!]) \wedge ([\![y]\!] = [\![d]\!])$.

Translating node calls. I performed instantiation of a node every time it is called, aka inlining. This raises a scalability problem as, if this inlining is not taken into account for invariant generation (see Section IV.2.1), we need to learn invariants for each instance of the node separately. Kind2 comes up with different mechanisms to ensure modularity and scalability (compositional reasoning, modular reasoning, progressive refinements, ...)[6] [1], [2] which I chose not to implement.

IV An invariant learning algorithm: Houdini

I first provide a very brief overview of invariant learning algorithms, and then focus on Houdini.

IV.1 Overview of invariant learning algorithms

The context is the following: we want to prove a property P *by induction* on a given program. P might not be inductive by itself, so we want to strenghten it with inductive lemmas (*.i.e.* invariants) to make the resulting set of properties inductive. Invariant learning algorithms can be classified in two classes: bottom-up or top-down (property-directed) algorithms.

Bottom-up. These algorithms try to learn as many inductive invariants on the program as possible. This learning process is executed without any specific desired property. Then, we can check if the learned invariants make the desired property inductive. We might have learned many invariants

[5] We could technically consider $\Delta(n+1) \wedge P(n) \Rightarrow P(n+1)$ but this would lead to more spurious counterexamples, *i.e.* states that contradict the inductivity check are in fact not reachable anyway.

[6] See https://github.com/kind2-mc/kind2/discussions/1256.

which appear to be useless when looking at the desired property. Houdini [4] and instantiation-based invariant discovery [2][7] are instances of bottom-up algorithms.

Top-down. These algorithms try to strenghten the desired property until it is inductive. IC3 [11] and (unlike Houdini) H-Houdini are property-directed algorithms.

As mentioned earlier, real-world projects such as Kind2 use both kinds of algorithms simultaneously to perform an efficient and modular analysis. In practice, bottom-up algorithms (at least the ones studied here) are simpler than top-down ones.

IV.2 Houdini

Houdini consists in three steps:

- generate (a huge number of) candidate invariants according to some templates,
- remove candidates invalidated by positive examples, that is a trace of execution of the program,
- remove predicates that break inductivity; iterate this step until the overall property is inductive.

Also, an additional step consists in filtering "obvious" invariants. This leads to a less polluted output of Loustrini when printing learned invariants.

IV.2.1 Generating candidate invariants

I used the following templates for boolean and numeric (integer and real) variables:

$$\begin{aligned} \mathcal{I}_b &::= b = \text{true} \mid b = \text{false} \mid b_1 = b_2 \mid b_1 = \neg b_2 \\ \mathcal{I}_{\text{num}} &::= x \diamond \text{cst} \mid x_1 \diamond x_2 \\ \text{where} \quad \diamond &::= = \mid \neq \mid \geq \mid > \mid \leq \mid < \\ \text{cst} &::= -1 \mid 0 \mid 1 \mid \{\text{any numeric value hardcoded in the program}\} \end{aligned}$$

Above, b, b_1, and b_2 are boolean variables, and x, x_1, and x_2 are integer (or real) variables. All possible candidates are generated using these templates. This unfortunately leads to contradictory candidates, which we need to be cautious about when trying to remove invalid candidates.

IV.2.2 Sifting candidate invariants

Removing candidates breaking inductivity. Let's take things backwards and start with the third step. We consider a set of candidates $\{h_1, ..., h_d\}$ and $H = \bigwedge_{i=1}^{d} h_i$. Is H inductive? We use the following query:

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

$$\begin{aligned} &\texttt{UNSAT} : H \text{ is inductive OR } H(n) \text{ contradicts } \Delta(n) \wedge \Delta(n+1) \\ &\texttt{SAT} : \exists \text{ cex}, \text{cex} \vDash H(n) \wedge \text{cex} \vDash \neg H(n+1)) \ \textit{i.e.}\ \exists i \in \{1, ..., d\}, \text{cex} \vDash \neg h_i(n+1) \end{aligned}$$

Using the counterexample cex, we remove all candidates that are invalidated by it, and we iterate until the remaining H is inductive (the empty set in the worst case).

This method is complete relatively to the templates, that is, if an invariant exists as a conjunction of candidates, it will be found. That would not work at all with a naive algorithm consisting in checking every candidate separately.

Note that to be able to use this method, we need to ensure that $\Delta(n) \wedge \Delta(n+1) \wedge H(n)$ is not vacuously false, else we are not able to conclude when the query for inductivity is `UNSAT`.

[7] Morally, the algorithm described as instantiation-based invariant discovery can be seen as an instance of Houdini.

Initial sift using positive examples. The role of this initial sift is twofold: first, it allows to make H consistent with our system equations, then, it allows to greatly shrink the set of candidates, accelerating the third step of the algorithm. We can obtain positive examples (*i.e.* examples that the invariants must satisfy) by simulating a trace of execution of length d, for all $k \leq d$:

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

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

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

IV.2.3 Comments on Houdini

Houdini is conceptually very simple, but this comes at a cost:

1. candidates generation is subject to combinational explosion,
2. we might need to perform many *large* SMT queries before reaching an inductive set,
3. we can only learn invariants which are conjuncts of candidates given by the templates (compared to k-induction or IC3 which do not suffer from this restriction).

V H-Houdini

H-Houdini [5], which stands for Hierarchical Houdini aims at solving point 2. from above (and also point 1.), by making Houdini top-down/property-directed.

V.1 Overview

Before explaining the algorithm, let's introduce the concept of abducts. An abduct A for a property p is a property that makes p inductive. That is, we have $A \wedge p \Rightarrow p'$.

A very simplified version of the algorithm is provided in Algorithm 1. Intuitively, H-Houdini tries to divide-and-conquer the inductivity of p using abducts. If an abduct fails, we need to backtrack.

Algorithm 1: H-Houdini

```
1:  procedure learn(p_target)
2:      ▷ Returns $H$ an invariant that proves p_target or none
3:      $V \leftarrow \mathcal{O}_{\text{slice}}(\text{p\_target})$
4:      $P \leftarrow \mathcal{O}_{\text{mine}}(\text{p\_target}, V)$
5:      solution ← false
6:      while not solution do
7:          $H \leftarrow$ p_target
8:          $A \leftarrow \mathcal{O}_{\text{abduct}}(\text{p\_target}, P)$
9:          if $A$ is none then
10:             return none
11:         end
12:         solution ← true
13:         for $p \in A$ do
14:             $H_{\text{sol}} \leftarrow$ learn(p)
15:             if $H_{\text{sol}}$ is none then
16:                 solution ← false
17:                 break
18:             end
19:             $H \leftarrow H \wedge H_{\text{sol}}$
20:         end
21:     end
22:     return $H$
23: end
```

To find inductive invariants to prove p_target, H-Houdini works as follows:

1. First, we extract a set V of variables on which p_target depends on, using a slice operator $\mathcal{O}_{\text{slice}}$. V is the set of all variables which can influence p_target in one step of execution.
2. Then, we generate a set of candidate invariants P using an operator $\mathcal{O}_{\text{mine}}$.
3. Then, until we found a solution or none is returned, we generate an abduct A from P using $\mathcal{O}_{\text{abduct}}$. We require each call to $\mathcal{O}_{\text{abduct}}$ to generate a new abduct A or none if no new abduct is possible.

V.2 Slicing

$\mathcal{O}_{\text{slice}}$ was implemented by instrumenting the translation from Lustre to SMT. Note that the current implementation requires to handle equations of a node in (combinatorial) topological order, which was not implemented yet, so for now, equations should be written in topological order.

V.3 Mining

$\mathcal{O}_{\text{mine}}$ is very similar to the generation of candidates for Houdini, but we need to make sure that the generated invariants are consistent with p_target. As of submitting the project, the implementation could be more straightforward: I generate candidates without knowledge of p_target and then sift these candidates. It would be better to generate candidates with respect to p_target.

V.4 Abducting

This is the magical part, and so, where the issues are.

A method described in the paper is to emit the following query:

$$P \land \text{p\_target} \land \neg\ \text{p\_target'}$$

If it is `SAT`, we return none. Else, it is `UNSAT` and we fetch and return an unsat core using the SMT solver. Ideally, this unsat core should be “minimal” (because each assertion of the unsat core will (potentially) lead to a H-Houdini call).

I fail to see how this method allows to generate more than one abduct for given p_target and P. An other way would be to rely on the SMT solver directly, but to the best of my knowledge Z3 does not provide an API to compute abducts. cvc5 does but does not provide OCaml bindings.

Moreover, in practice, the unsat core returned by Z3 suffers from two issues:
- instead of being a list of simple assertions, it is a list containing only one expression: a large conjunction,
- and this conjunction seems very far from being minimal, for any reasonable definition of minimal, which really doesn't help as it would emit more proof obligations for H-Houdini.

As of submitting the project, this part contains known and also, for sure, yet unknown bugs.

VI Evaluation

Proving with Houdini. Loustrini can successfully prove non k-inductive properties (see `ic3.lus`). Also, Houdini successfully finds invariants that would fail if tested separately (see remark on completeness in Section IV.2.2, and `ic_more.lus`).

However, Houdini (and H-Houdini) can easily fail: we generate candidates only for existing Lustre variables. It could be beneficial to introduce state variables (see `fib.lus`) and generate candidates for them too, at the risk of combinational explosion.

Proving with H-Houdini. This part is clearly not ready yet.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] “Alt-Ergo Zero.” Accessed: Jan. 05, 2025. [Online]. Available: https://usr.lmf.cnrs.fr/cubicle/alt-ergo-zero/

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

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

[9] 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.

[10] L. De Moura and N. Bj́rner, “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.

[11] 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.