Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 267 (2010) 55–68
www.elsevier.com/locate/entcs

# SMT-AI: an Abstract Interpreter as Oracle for $k$-induction

Pierre Roux, Rémi Delmas and Pierre-Loïc Garoche [1]

*ONERA – DTIM*
*Toulouse, France*

Abstract

The last decade has seen a major development of verification techniques based on SMT solvers used to prove inductive invariants on systems. This approach allows to prove functional properties and scale up to handle industrial problems. However, it often needs a man in the loop to provide hand-written lemmas on the system in order to help the analysis and complete the proof.
This paper presents a tool that automatically generates lemmas. It takes such systems and over-approximates their collecting semantics, providing a bound on the numerical memories. It is based on the abstract interpretation methodology introduced by Cousot in 1977.

*Keywords:* $k$-induction, abstract interpretation, lemmas generator, SMT

## 1 Context and Motivation

Critical systems have to meet stringent certification requirements such as the D0-178 for avionics [15]. Among those systems, control command software is often written in the well suited synchronous paradigm [2,11], hence a strong interest in proving functional properties on synchronous systems.

The next section will introduce and motivate a combination of analysis methods while the remaining of the article focus on the implementation of one of them. Section 3 describes the input language of our tools. Then the two following sections deal with the analysis it performs. Finally, section 6 gives some details about the implementation, section 7 details an example and section 8 concludes and gives an overview of what remains to be done.

[1] Email: `firstname.lastname@onera.fr`


doi:10.1016/j.entcs.2010.09.018

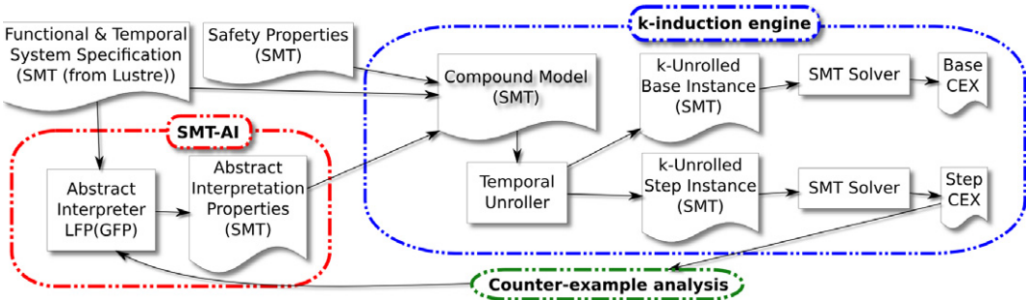


Figure 1. General combination of analyzes, a heuristic analysis of $k$-induction counter-examples could allow to finely tune cost/precision ration of abstract interpretation analysis.

# 2 A combination of analysis

## *2.1 k-induction*

The $k$-induction method [16] is a SMT/SAT-based model checking technique aiming at proving properties by induction on the analyzed system.

If $I$ and $T$ are predicates for initial state and transition relation of a synchronous system and $P$ a property to prove on it, $k$-induction uses formulas:

$$Base(k) := I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)$$

$$Step(k) := (P(s_0) \wedge \cdots \wedge P(s_k)) \wedge (T(s_0, s_1) \wedge \cdots \wedge T(s_k, s_{k+1})) \wedge \neg P(s_{k+1})$$

Starting from 0, $Base(k)$ and $Step(k)$ are fed to solver for increasing values of $k$ until either $Base(k)$ is satisfiable meaning property $P$ does not hold on the system or $Base(k)$ and $Step(k)$ are both unsatisfiable achieving a proof of $P$ by induction. Using SMT solvers allows to handle infinite state systems. It is also interesting to notice that in case of failure, models of satisfiable formulas give counter examples.

This technique proved effective [10] but often requires the user to strengthen the property $P$ with additional lemmas in order to make it inductive for a reasonable value of $k$.

## *2.2 Abstract interpretation*

Abstract interpretation [6] is a theoretical framework to build static analyzers computing sound approximations of the semantics of analyzed programs. Tools based on it achieve a very high level of automation [7].

## *2.3 Combination of k-induction and abstract interpretation*

Our goal is to use an abstract interpreter to automatically infer invariants of the system which can be used, as lemmas coming from an external oracle, by a $k$-induction tool. We hope to take this way advantage of both power of SMT-solvers for deductive reasoning and efficiency of abstract interpretation to infer numerical invariants. General combination is sketched in Figure 1. although this paper will only deal with the abstract interpreter.

### *2.4 Related work*

Other works use spurious counter examples [8] or instantiation of parametrizable patterns [4] to strengthen the invariant in order to make it inductive hence provable by $k$-induction. Those methods are mostly symbolic whereas a more semantic method such as abstract interpretation could infer more easily properties not appearing in the original program.

## 3 Input language

### *3.1 Target: Lustre synchronous language*

To analyze Lustre programs [11], the $k$-induction engine has to compile them to SMT-lib [1] language, used by SMT solvers, during a phase very similar to code generation. Therefore, we chose to use this SMT-lib code, enriched with a few predicates to describe temporal aspects, as input language of our abstract interpreter.

### *3.2 Syntax*

Synchronous systems analyzed by our tool are expressed in SMT-lib [1] extended with following predicates:

- `init` is true at initial step then always false, this would be written `true -> false` in Lustre [11];
- `memu(v, e)` means that variable `v` acts as a memory which takes an undefined value at first step then takes value of expression `e` during previous step, this is equivalent to `v = pre e` in Lustre;
- `memi(v, e1, e2)` is the same as `memu(v, e2)` excepts that value of `v` at first step is defined by expression `e1`, this amounts to `v = e1 -> pre e2`.

Our input syntax is then described by the following grammar:

$$
\begin{array}{lll}
e & ::= & v \mid \text{init} \mid const \mid (unop\ e) \mid (binop\ e\ e) \mid (nop\ el) \\
 & & \mid (\text{ite}\ e\ e\ e) \mid (\text{let}\ (v\ e)\ e) \mid (\text{memu}\ v\ e) \mid (\text{memi}\ v\ e\ e) \\
el & ::= & e\ e \mid e\ el \\
v & ::= & \mathbb{V} \\
const & ::= & \mathbb{B} \mid \mathbb{Z} \\
unop & ::= & - \mid \text{not} \\
binop & ::= & < \mid > \mid \leq \mid \geq \mid - \mid + \mid * \mid \text{implies} \\
nop & ::= & = \mid \text{iff} \mid \text{and} \mid \text{or} \mid \text{xor} \mid \text{distinct}
\end{array}
$$

with $\mathbb{V}$ a set of variable names, $\mathbb{B} = \{\text{true}, \text{false}\}$ the set of booleans and $\mathbb{Z}$ the set of integers. $\mathbb{B} \cup \mathbb{Z}$ will be denoted Val.

Two types `Bool` and `Int` along with usual typing rules are used to ensure that only well typed expressions are considered. Moreover nested `memi/memu` are forbidden.

### *3.3 General shape*

Analyzed code coming from compilation of Lustre code takes the shape of nested let definitions for all intermediate variables sorted by topological order. Moreover the compilation ending with a common subexpression elimination phase, the result looks a bit like three address code.

### *3.4 Semantic*

A denotational semantic is given in section 4.2.1.

It is important to notice here that we consider an idealized semantic with arithmetic on unbounded ring $\mathbb{Z}$. For proven properties to remain valid at executable code level, it will be necessary to prove that no overflow can happen in compiled program. Real numbers are not considered here but if they were it would also be necessary to prove that the floating point implementation has an error in some small enough bound from model in $\mathbb{R}$. Efficient tools already exist to prove those properties [7,9].

## 4 Abstract Interpretation

### *4.1 Synchronous Systems Main Loop*

Abstract semantic is computed through usual least fixpoint increasing iterations with a delayed widening with thresholds followed by decreasing steps with narrowing, `memi` and `memu` predicates acting as assignments for variables used as memory of the synchronous system. Most of the work consists in analyzing the expression defining the values of those variables from the values at previous step. The remaining of this section will focus on this second *separate analysis*, conducted by decreasing iterations toward a greatest fixpoint.

### *4.2 Analysis of Expressions*

Since our goal is to compute by abstract interpretation an over-approximation, the `not` operator will require to also compute an under-approximation. To get rid of this, we can put expressions in negative normal form.Therefore we will only encounter `not` in front of variables from now on. We will also forget unary $-$, `implies`, `iff`, `xor` and `distinct` in the following, without restriction since they can be seen as syntactic sugar. Finally `=`, `and` and `or` will be treated as binary operators.

#### *4.2.1 Concrete Semantic*

We define here a semantic suitable for abstract interpretation.

For all expression $e$, $[\![e]\!]$ is a function from $\mathbb{V} \to \mathrm{Val}$ to Val defined by:

$$[\![v]\!](\rho) = \rho(v) \qquad [\![\text{not } v]\!](\rho) = \neg\, \rho(v) \qquad [\![c]\!](\rho) = c \quad \text{for } c \in \mathrm{Val}$$

$$[\![binop\ e_1\ e_2]\!](\rho) = [\![e_1]\!](\rho)\ binop\ [\![e_2]\!](\rho)$$

for $binop \in \{\text{and}, \text{or}, =, <, >, \leq, \geq, -, +, *\}$

$$[\![\text{ite } e_b\ e_t\ e_e]\!](\rho) = \begin{cases} [\![e_t]\!](\rho) & \text{if } [\![e_b]\!](\rho) = \text{true} \\ [\![e_e]\!](\rho) & \text{if } [\![e_b]\!](\rho) = \text{false} \end{cases}$$
$$[\![\text{let } (v\ e_1)\ e_2]\!](\rho) = [\![e_2]\!](\rho[v \mapsto [\![e_1]\!](\rho)])$$

For all expression $e$ of type `Bool`, $[\![e]\!]_{\mathcal{P}}$ is a function from $2^{(\mathbb{V}\to\text{Val})}$ to itself defined by:

$$[\![e]\!]_{\mathcal{P}}R = \{\rho \in R \mid [\![e]\!](\rho) = \text{true}\}$$

We can then define our second semantics of an expression $e$ as the greatest fixpoint of $[\![e]\!]_{\mathcal{P}}$:

$$[\![e]\!]_2 = \text{gfp}[\![e]\!]_{\mathcal{P}}$$

This is well defined according to Knaster-Tarski theorem [17] since $[\![e]\!]_{\mathcal{P}}$ is monotonic on the complete lattice $\left(2^{(\mathbb{V}\to\text{Val})}, \subseteq, \cup, \cap, \emptyset, \mathbb{V} \to \text{Val}\right)$.

The greatest fixpoint seems a bit artificial here but is convenient to define our analysis and prove its soundness.

*4.2.2 A Combined Forward-Backward Abstract Semantic*

We present here a non relational analysis [2]. For this purpose, we assume an abstract domain $\text{Int}^\sharp$ for integers $\mathbb{Z}$ (for example intervals) and an abstract domain $\text{Bool}^\sharp$ based on lattice of figure 2 for booleans. Disjoint sum $\text{Bool}^\sharp \uplus \text{Int}^\sharp$ will be written $\text{Val}^\sharp$ and $\text{Env}^\sharp$ denotes environments of such abstract values: $\text{Env}^\sharp = \mathbb{V} \to \text{Val}^\sharp$. In case one variable is $\bot_{\text{Val}^\sharp}$, the whole environment is written $\bot_{\text{Env}^\sharp}$ meaning that not any valid environment exists.

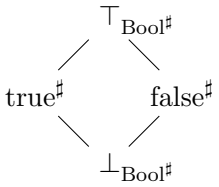


Figure 2: Lattice underlying boolean abstract domain $\text{Bool}^\sharp$.

**Basic Abstract Transformers**

Abstract domain $\text{Val}^\sharp$ comes with following abstract transformers. They are the abstract counterparts of concrete operations in the analyzed language:

- $\text{const}^\sharp : \text{Val} \to \text{Val}^\sharp$, $\text{const}^\sharp(c)$ is an abstract value representing concrete value $c \in \text{Val}$;
- $[\![\text{not}]\!]\uparrow^\sharp_{\text{unop}} : \text{Val}^\sharp \to \text{Val}^\sharp$, forward abstract semantic of not: for any boolean $b$, if abstract value $b^\sharp$ represents $b$, then $[\![\text{not}]\!]\uparrow^\sharp_{\text{unop}}(b^\sharp)$ represents $\neg b$;
- $[\![\text{not}]\!]\downarrow^\sharp_{\text{unop}} : \text{Val}^\sharp \times \text{Val}^\sharp \to \text{Val}^\sharp$, backward abstract semantic of not: for $b^\sharp$ and $r^\sharp$, $[\![\text{not}]\!]\downarrow^\sharp_{\text{unop}}(b^\sharp, r^\sharp)$ is a refinement of $b^\sharp$ (element lower in the lattice) knowing that for any boolean $b$ represented by $b^\sharp$, $\neg b$ is represented by $r^\sharp$;
- similar forward and backward transformers for binary operators and `ite`.

[2] Our tool also offers relational analysis though. This is implemented thanks to the APRON [3] library, which offers relational abstract domains such as the octagons [12].

**Forward Abstract Semantic**

For all expression $e$, $[\![e]\!]\uparrow^\sharp$ is a function from $\mathrm{Env}^\sharp$ to $\mathrm{Val}^\sharp$. $[\![e]\!]\uparrow^\sharp(\rho^\sharp)$ is a classical abstract evaluation of expression $e$ in abstract environment $\rho^\sharp$:

$[\![v]\!]\uparrow^\sharp(\rho^\sharp) = \rho^\sharp(v) \qquad [\![\text{not } v]\!]\uparrow^\sharp(\rho^\sharp) = [\![\text{not}]\!]\uparrow^\sharp_{\text{unop}}(\rho^\sharp(v))$
$[\![c]\!]\uparrow^\sharp(\rho^\sharp) = \text{const}^\sharp(c)$ for $c \in \text{Val}$
$[\![binop\ e_1\ e_2]\!]\uparrow^\sharp(\rho^\sharp) = [\![binop]\!]\uparrow^\sharp_{\text{binop}}([\![e_1]\!]\uparrow^\sharp(\rho^\sharp), [\![e_2]\!]\uparrow^\sharp(\rho^\sharp))$
for $binop \in \{\text{and}, \text{or}, =, <, >, \leq, \geq, -, +, *\}$
$[\![\text{ite } e_b\ e_t\ e_e]\!]\uparrow^\sharp(\rho^\sharp) = [\![\text{ite}]\!]\uparrow^\sharp_{\text{ite}}([\![e_b]\!]\uparrow^\sharp(\rho^\sharp), [\![e_t]\!]\uparrow^\sharp(\rho^\sharp_t), [\![e_e]\!]\uparrow^\sharp(\rho^\sharp_e))$
where $\rho^\sharp_t = [\![e_b]\!]\downarrow^\sharp(\text{true}^\sharp, \rho^\sharp)$ and $\rho^\sharp_e = [\![e_b]\!]\downarrow^\sharp(\text{false}^\sharp, \rho^\sharp)$
$[\![\text{let } (v\ e_1)\ e_2]\!]\uparrow^\sharp(\rho^\sharp) = [\![e_2]\!]\uparrow^\sharp(\rho^\sharp[v \mapsto [\![e_1]\!]\uparrow^\sharp(\rho^\sharp)])$

Most cases are just usual evaluation of the expression. For `ite`, backward semantic is used to refine the environment knowing that the guard evaluates to true in the then branch and to false in the else branch.

**Backward Abstract Semantic**

For all expression $e$, $[\![e]\!]\downarrow^\sharp$ is a function from $\mathrm{Val}^\sharp \times \mathrm{Env}^\sharp$ to $\mathrm{Env}^\sharp$. $[\![e]\!]\downarrow^\sharp(n^\sharp, \rho^\sharp)$ is a refinement of abstract environment $\rho^\sharp$ knowing that $e$ evaluates in something over-approximated by $n^\sharp$:

$[\![v]\!]\downarrow^\sharp(n^\sharp, \rho^\sharp) = \rho^\sharp[v \mapsto \rho^\sharp(v) \sqcap^\sharp_{\text{Val}^\sharp} n^\sharp]$
$[\![\text{not } v]\!]\downarrow^\sharp(n^\sharp, \rho^\sharp) = \rho^\sharp[v \mapsto [\![\text{not}]\!]\downarrow^\sharp_{\text{unop}}(\rho^\sharp(v), n^\sharp)]$

Information about variable $v$ is added to what is already in environment.

$$[\![c]\!]\downarrow^\sharp(n^\sharp, \rho^\sharp) = \begin{cases} \bot_{\text{Env}^\sharp} & \text{if } \text{const}^\sharp(c) \sqcap^\sharp_{\text{Val}^\sharp} n^\sharp = \bot_{\text{Val}^\sharp} \\ \rho^\sharp & \text{otherwise} \end{cases}$$

We can't refine environment except if $c$ is incompatible with value $n^\sharp$.

$[\![binop\ e_1\ e_2]\!]\downarrow^\sharp(n^\sharp, \rho^\sharp) = [\![e_1]\!]\downarrow^\sharp(n^\sharp_1, \rho^\sharp) \sqcap^\sharp_{\text{Env}^\sharp} [\![e_2]\!]\downarrow^\sharp(n^\sharp_2, \rho^\sharp)$
for $binop \in \{=, <, >, \leq, \geq, -, +, *\}$
where $(n^\sharp_1, n^\sharp_2) = [\![binop]\!]\downarrow^\sharp_{\text{binop}}([\![e_1]\!]\uparrow^\sharp(\rho^\sharp), [\![e_2]\!]\uparrow^\sharp(\rho^\sharp), n^\sharp)$

We evaluate $e_1$ and $e_2$ forward before refining this values, since $binop\ e_1\ e_2$ evaluates in $n^\sharp$, giving $n^\sharp_1$ and $n^\sharp_2$ from which $e_1$ and $e_2$ are backward evaluated.

$$[\![\text{and } e_1\ e_2]\!]\downarrow^\sharp(n^\sharp, \rho^\sharp) = \begin{cases} \bot_{\text{Env}^\sharp} \text{ if } ([\![\text{and}]\!]\uparrow^\sharp_{\text{binop}}([\![e_1]\!]\uparrow^\sharp(\rho^\sharp), [\![e_2]\!]\uparrow^\sharp(\rho^\sharp))) \sqcap^\sharp_{\text{Val}^\sharp} n^\sharp = \bot_{\text{Val}^\sharp} \\ [\![e_1]\!]\downarrow^\sharp(\text{true}^\sharp, \rho^\sharp) \sqcap^\sharp_{\text{Env}^\sharp} [\![e_2]\!]\downarrow^\sharp(\text{true}^\sharp, \rho^\sharp) \quad \text{if } n^\sharp = \text{true}^\sharp \\ [\![e_1]\!]\downarrow^\sharp(\text{false}^\sharp, \rho^\sharp) \sqcup^\sharp_{\text{Env}^\sharp} [\![e_2]\!]\downarrow^\sharp(\text{false}^\sharp, \rho^\sharp) \quad \text{if } n^\sharp = \text{false}^\sharp \\ \rho^\sharp \quad \text{otherwise} \end{cases}$$

When the result $n^\sharp$ is $\text{false}^\sharp$, one of the subexpressions $e_1$ or $e_2$ can force the value of the conjunction. We can thus only compute a join of backward evaluations to $\text{false}^\sharp$ of subexpressions. Disjunction is similar, replacing join with meet and vice versa.

$$[\![\text{ite } e_b\ e_t\ e_e]\!]\downarrow^\sharp(n^\sharp,\rho^\sharp) = [\![e_b]\!]\downarrow^\sharp(n_b^\sharp,\rho^\sharp) \sqcap^\sharp_{\text{Env}^\sharp} \left([\![e_t]\!]\downarrow^\sharp(n_t^\sharp,\rho_t^\sharp) \sqcup^\sharp_{\text{Env}^\sharp} [\![e_e]\!]\downarrow^\sharp(n_e^\sharp,\rho_e^\sharp)\right)$$
where $\rho_t^\sharp = [\![e_b]\!]\downarrow^\sharp(\text{true}^\sharp,\rho^\sharp)$ and $\rho_e^\sharp = [\![e_b]\!]\downarrow^\sharp(\text{false}^\sharp,\rho^\sharp)$
and $(n_b^\sharp, n_t^\sharp, n_e^\sharp) = [\![\text{ite}]\!]\downarrow^\sharp_{\text{ite}}([\![e_b]\!]\uparrow^\sharp(\rho^\sharp),[\![e_t]\!]\uparrow^\sharp(\rho_t^\sharp),[\![e_e]\!]\uparrow^\sharp(\rho_e^\sharp),n^\sharp)$

Backward `ite` can return $\text{true}^\sharp$ for $n_b^\sharp$ for example if $n_e^\sharp$ and $n^\sharp$ are incompatible.

$$[\![\text{let } (v\ e_1)\ e_2]\!]\downarrow^\sharp(n^\sharp,\rho^\sharp) = [\![e_1]\!]\downarrow^\sharp(\rho_1^\sharp(v),\rho_1^\sharp[v\mapsto\rho^\sharp(v)])$$
where $\rho_1^\sharp = [\![e_2]\!]\downarrow^\sharp(n^\sharp,\rho^\sharp[v\mapsto[\![e_1]\!]\uparrow^\sharp(\rho^\sharp)])$

Reassignment of value $\rho^\sharp(v)$ to variable $v$ is here only to avoid captures when same variable name $v$ is used multiple times. When implementing, a simpler solution can be to guarantee uniqueness of all variable names

The abstract semantics is finally computed through *decreasing iterations*:

$$[\![e]\!]^\sharp = \prod_{n}^{\sharp_{\text{Env}^\sharp}} \left(\lambda X.[\![e]\!]\downarrow^\sharp(\text{true}^\sharp,X)\right)^n (\top_{\text{Env}^\sharp})$$

Such backward semantics is commonly applied on guards of if ... then ... else or while loop constructs of imperative languages like C. Iterating can be needed in our case to gain precision, when some information learn in one part of an expression enables to get more precise results in another part.

**Example 4.1** On the following expression, a first iteration ensures that `b1` is true which allows a second iteration to discover that `b2` must also be true and `x` must be negative which finally appears impossible during a third iteration:

```
(and (ite b1 (and b2 (< x 0)) true)
     (<= (ite b1 (ite b2 (-x) x) 1) 0))
```

### *4.2.3 Partial correctness*

Concrete and abstract values are linked together by a concretization function $\gamma : \text{Val}^\sharp \to 2^{\text{Val}}$. For any abstract value $n^\sharp$, $\gamma(n^\sharp)$ is the set of (concrete) values represented by $n^\sharp$. There is a similar concretization function $\gamma_{\text{Env}} : \text{Env}^\sharp \to 2^{\mathbb{V}\to\text{Val}}$ for environments: $\gamma_{\text{Env}} = \rho^\sharp \mapsto \left\{\rho \;\middle|\; \forall v \in \mathbb{V}.\rho(v) \in \gamma(\rho^\sharp(v))\right\}$.

Assuming that basic abstract transformers fulfill usual soundness hypotheses, following theorem can be proved.

**Theorem 4.2** *For all expression e,* $[\![e]\!]_2 \subseteq \gamma_{\text{Env}}\left([\![e]\!]^\sharp\right)$.

This shows that the computed abstract semantic $[\![e]\!]^\sharp$ is a sound over-approximation of the concrete semantic [3].

[3] In particular if the abstract semantics computed is $\bot_{\text{Env}^\sharp}$, it proves that the formula is unsatisfiable. This did happen when running the tool on SMT-lib benchmarks but this is not our goal and an SMT-solver can do it much more efficiently. Moreover it cannot happen on formula describing deterministic synchronous systems.

#### *4.2.4 Termination Issue*

For the computation of the abstract semantics $[\![e]\!]^\sharp$ to terminate, we need the decreasing sequence $\left(\left(\lambda X.[\![e]\!]\downarrow^\sharp (true, X)\right)^n\right)_{n\in\mathbb{N}}$ to be ultimately stationary. This could not be the case if the lattice underlying the abstract domain $\mathrm{Val}^\sharp$ does not meet the condition of absence of infinite decreasing chains (DCC). This property does not hold for the commonly used interval lattice for example which accepts the infinite decreasing chain $([n, +\infty))_{n\in\mathbb{N}}$. This lead the analysis of expressions such as `(and (> x 0) (> x y) (> y x))` not to terminate with this abstract domain.

We can not make use of a widening operator (or more precisely its dual) to accelerate convergence since this would lead to an under-approximation of the greatest fixpoint whereas we want to compute an over-approximation of it. Only solution to enforce convergence in reasonable time is to make use of a narrowing operator which basically amounts to bound the number of iterations. This would lead to far too much coarse results if iterations were commonly stopped by narrowing. However convergence seems to be reached after only a few iterations on the formula fed to our tool (c.f. section 3.3).

#### *4.2.5 Precision Issue*

There are a few well located points were loss of precision usually happens: when computing join operators $\sqcup^\sharp$ under `or` and `ite` constructs and when removing of environments local variables bounded by `let` constructs. This can lead to forget information which can be useful at next iteration to improve precision of the result.

There are two ways to address the problem:

(i) computing local fixpoints before any operation which could lead to a loss of information or

(ii) caching potentially lost information to reuse it a next iteration.

The first solution present the major drawback of an exponential complexity in the number of nested points where such local fixpoints are computed.

Therefore we chose to adopt the second solution for let constructs which are usually deeply nested.

#### *4.2.6 Efficiency Issue*

Looking at the abstract semantic, it appears that during each iteration, forward semantics of leafs of syntax tree of an expression will be recomputed again and again when computing forward semantics of each node above. Use of common dynamic programming techniques addresses this problem.

Abstract meet of two backward semantics $[\![e_1]\!]\downarrow^\sharp \left(n_1^\sharp, \rho^\sharp\right) \sqcap^\sharp_{\mathrm{Env}^\sharp} [\![e_2]\!]\downarrow^\sharp \left(n_2^\sharp, \rho^\sharp\right)$ can also be replaced by the computation of one of them from the result of the other $[\![e_2]\!]\downarrow^\sharp \left(n_2^\sharp, [\![e_1]\!]\downarrow^\sharp \left(n_1^\sharp, \rho^\sharp\right)\right)$. Doing this for the `and` allows to save an iteration on example 4.1.

# 5 Inlining and Partitioning

The analyzed code having lot of expressions defined under let constructs (c.f. section 3.3), they tend to be analyzed in a less precise environment than the one they are actually used in (e.g. in a branch of an if).

A first easy solution is to inline all let definitions. But this can, among other things, lead to coarser results by loosing correlation between multiple instances as shown in following example.

**Example 5.1** The following expression:

```
(let (b (< x y)) (and b (not b)))
```

gives after inlining: `(and (< x y) (<= y x))` which does not allow to conclude to unsatisfiability without using a relational domain with `x` and `y` whereas it was obvious before inlining.

A better solution is to use a symbolic abstract domain [13] keeping trace of the expression attached by a let to a variable to be able to analyze it when this variable is then encountered during the analysis.

However keeping lot of local variables in abstract environment makes computation of abstract meet $\sqcap^\sharp$ and join $\sqcup^\sharp$ operators slower hence some interest for inlining. We have in particular no reason not to inline variables used only once (which is common in the expressions we analyze).

Sometime, all this is not sufficient to have some code analyzed in a precise enough context.

**Example 5.2** If $\texttt{x} \leq \texttt{N}$, The following expression evaluates to something not greater than `N`:

```
(+ x (ite (< x N) 1 0))
```

but we need to analyze the whole sum and not only the branches of the `ite` in both contexts $\texttt{x} < \texttt{N}$ and $\texttt{x} \geq \texttt{N}$ to discover it.

A good way to do it is to use trace partitioning [14], keeping two set of traces for the `ite` and merging them after the sum.

**Example 5.3** This technique allows to infer non trivial invariants such as $\texttt{check} \geq 0$ on the following program:

```
(let (check (ite b (-x) 1))
(let (neg_abs_x (ite (< x 0) x (-x)))
(and (memi b true (= check 1))
     (memi x (-2) (ite (not b) neg_abs_x
               (ite input_0 (+ (+ x check) (-2)) (- check x)))))))
```

by partitioning against the value of `b` at current and previous step.

# 6 Implementation

We implemented the previously described analysis in a prototype. The OCaml code is available under GPL license at `http://cavale.gforge.enseeiht.fr/smt-ai/`.

Input language is, as described in section 3.2, raw SMT-lib [1] plus a few extra predicates:

- `init`, `memu` and `memi` to enable description of synchronous systems (c.f. section 3.2);
- `trace_partitioning` and `trace_merge` semantically equivalent to identities and used as pragmas for trace partitioning (c.f. section 5, a partitioning id enables not well-parenthesized partitioning).

The analysis proceeds in three consecutive phases:

(i) parsing and type checking the input, consistency of partitioning ids is also checked;

(ii) inlining (c.f. section 5), normalization (negation normal form) and extraction of two formulas respectively describing initial state and transition relation of the system;

(iii) the analysis itself (c.f. section 4).

The tool outputs the result of the analysis, in particular bounds for values of the variables in reachable states.

# 7 Example

In the following Lustre code, $k$-induction cannot prove that variable $OK$ is always true[4]. Our abstract interpreter is also unable to complete the proof but infers the invariant $x + y = 42$ along with bounds for numerical variables, which allows $k$-induction to conclude at depth $k = 0$, using bitblasting to handle the non linear operation.

**node** example(a : bool) **returns** (OK : bool);
**var** pre_x : int; pre_y : int; x : int; y : int; z : int; n : int;
**let**
  n = 42;
  pre_x = fby(0, 1, x); pre_y = fby(0, 1, y);
  x = 0 → **if** pre_x < 2 ∗ n **then** pre_x + 1 **else** pre_x;
  y = 2 ∗ n → **if** pre_y > 0 **then** pre_y − 1 **else** pre_y;
  z = x ∗ y;
  OK = z ≤ n ∗ n;
**tel**

[4] Although bounds on variables $x$ and $y$, even very coarse, and a so called path compression constraint seem to allow a proof at a depth $k$ related to the parameter $n$.

# 8 Conclusion and Future Work

We have presented an implementation of an abstract interpreter intended to work in collaboration with a $k$-induction procedure to prove functional properties on synchronous systems.

The analysis presented here, even used with simple abstractions such as intervals, already gives interesting results. Computing non relational properties like bounds on variables allows to optimize the analysis for the SMT solver, for example relying on bit-blasting techniques [5]. On some simple systems with integer counters, for instance, the $k$-induction analysis could necessitate to increase the induction depth up to unreasonable values, while our abstract interpretation tool using widening with thresholds will infer bounds in a few steps of computation.

This is still work in progress and a lot remains to do. The abstract interpreter being intended to be called many times with various parameters such as trace partitioning points and packing for relational domains, it could be interesting to be able to reuse previously computed results to speed up subsequent analysis. Finally, we have to test our approach on industrial case studies.

# References

[1] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. In *Proceedings of The IEEE*, pages 64–83, 2003.

[3] J. Bertrand and A. Miné. Apron: A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[4] A. Bradley and Z. Manna. Property-directed incremental invariant generation. *Formal Asp. Comput.*, 20(4-5):379–405, 2008.

[5] R. Bryant, D. Kroening, J. Ouaknine, S. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. *Software Tools for Technology Transfer (STTT)*, 11:95–104, 2009.

[6] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[7] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In S. Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[8] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification (extended abstract, category a). In W. Hunt Jr. and F. Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2003.

[9] E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2007.

[10] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In A. Cimatti and R. Jones, editors, *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (Portland, Oregon)*, pages 109–117. IEEE, 2008.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[13] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In E. Emerson and K. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2006.

[14] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[15] RTCA. *Software Considerations in Airborne systems and Equipment Certification*, 1992.

[16] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In W. Hunt Jr. and S. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[17] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

# A Equivalent operational semantic

We already describe semantics of temporal predicates `init`, `memi` and `memu` and we will focus here on the semantics of other constructs.

For an expression $e$, $c \in \text{Val}$ and a valuation $\rho : \mathbb{V} \to \text{Val}$ assigning a value to each variable appearing in e, we define $\rho \vDash e, c$ if $e$ evaluates to $c$ in $\rho$:

| | |
|---|---|
| $\rho \vDash v, c$ | if $\rho(v) = c$ |
| $\rho \vDash c, c'$ | if $c = c'$ where $c \in \text{Val}$ |
| $\rho \vDash (unop\ e), c$ | if $\exists\, c'.\ \rho \vDash e, c' \wedge unop\ c' = c$ where $unop \in \{-, \text{not}\}$ |
| $\rho \vDash (binop\ e_1\ e_2), c$ | if $\exists\, c_1\ c_2.\ \rho \vDash e_1, c_1 \wedge \rho \vDash e_2, c_2 \wedge c_1\ binop\ c_2 = c$ where $binop \in \{<, >, \leq, \geq, -, +, *, \text{implies}\}$ |
| $\rho \vDash (nop\ e_1\ \ldots\ e_n), c$ | if $\exists\, c_1\ \ldots\ c_n.\ \bigwedge_{i=1}^{n} \rho \vDash e_i, c_i \wedge nop_{i=1}^{n}\ c_i = c$ where $nop \in \{=, \text{iff}, \text{and}, \text{or}, \text{xor}, \text{distinct}\}$ |
| $\rho \vDash (\text{ite}\ e_b\ e_t\ e_e), c$ | if $(\rho \vDash e_b, \text{true} \wedge \rho \vDash e_t, c) \vee (\rho \vDash e_b, \text{false} \wedge \rho \vDash e_e, c)$ |
| $\rho \vDash (\text{let}\ \ (v\ e_1)\ \ e_2), c$ | if $\exists\, c'.\ \rho \vDash e_1, c' \wedge \rho[v \mapsto c'] \vDash e_2, c$ |

The semantics of an expression $e$ is then the set of valuations $\rho$ such that $\rho \vDash e, \text{true}$:

$$[\![e]\!]_1 = \{\rho : \mathbb{V} \to \text{Val} \mid \rho \vDash e, \text{true}\}$$

This semantic is equivalent to the one given in section 4.2.1.

**Theorem A.1** *Semantics are equivalent: for all expression e,* $[\![e]\!]_1 = [\![e]\!]_2$.

**Lemma A.2** *For all* $\rho : \mathbb{V} \to \text{Val}$*, for all expression e and* $c \in \text{Val}$*,* $[\![e]\!](\rho) = c$ *if and only if* $\rho \vDash e, c$.

**Proof** By structural induction on $e$. □

**Proof** [theorem A.1] For any $\rho : \mathbb{V} \to \text{Val}$, let us prove that $\rho \in e_1$ implies $\rho \in [\![e]\!]_2$. By definition of $[\![e]\!]_1$, $\rho \vDash e, \text{true}$ hence $[\![e]\!](\rho) = \text{true}$ by lemma A.2. Therefore singleton $\{\rho\}$ is a fixpoint of $[\![e]\!]$ and is then included in $\text{gfp}[\![e]\!] = [\![e]\!]_2$.

Conversely, if $\rho \in [\![e]\!]_2$, $[\![e]\!](\rho) = \text{true}$ hence $\rho \vDash e, \text{true}$ by lemma A.2 which amounts to say that $\rho \in [\![e]\!]_1$. □

# B Proof of analysis soundness (Theorem 4.2)

**Hypotheses**

$\gamma$ is assumed to satisfy the following properties:
$\gamma(\bot_{\mathrm{Val}^\sharp}) = \emptyset$ and $\gamma(\top_{\mathrm{Val}^\sharp}) = \mathrm{Val}$;
$\gamma$ is monotonic: $\forall x, y \in \mathrm{Val}^\sharp$ if $x \sqsubseteq_{\mathrm{Val}^\sharp} y$ then $\gamma(x) \subseteq \gamma(y)$;
$\sqcap^\sharp_{\mathrm{Val}^\sharp}$ is sound wrt $\cap_{\mathrm{Val}}$: $\forall x, y \in \mathrm{Val}^\sharp$, $\gamma(x) \cap_{\mathrm{Val}} \gamma(y) \subseteq \gamma\left(x \sqcap^\sharp_{\mathrm{Val}^\sharp} y\right)$;
$\gamma(\mathrm{true}^\sharp) = \{\mathrm{true}\}$ and $\gamma(\mathrm{false}^\sharp) = \{false\}$.
It can be noticed that $\gamma_{\mathrm{Env}}$ inherits from the three first properties according to its definition.

Moreover, the basic abstract transformers are expected to fulfill following soundness specifications [5]:

$$\forall c \in \mathrm{Val},\ c \in \gamma\left(const^\sharp(c)\right);$$
$$\forall x^\sharp \in \mathrm{Val}^\sharp, \forall x \in \mathbb{B}, x \in \gamma\left(x^\sharp\right) \Rightarrow \neg x \in \gamma\left(\llbracket \mathrm{not} \rrbracket^\sharp{}_{\mathrm{unop}}\left(x^\sharp\right)\right);$$
$$\forall x^\sharp, r^\sharp \in \mathrm{Val}^\sharp, \forall x \in \mathbb{B}, x \in \gamma\left(x^\sharp\right) \wedge \neg x \in \gamma\left(r^\sharp\right) \Rightarrow x \in \gamma\left(\llbracket \mathrm{not} \rrbracket\!\downarrow^\sharp{}_{\mathrm{unop}}\left(x^\sharp, r^\sharp\right)\right);$$
$$\forall x^\sharp, y^\sharp \in \mathrm{Val}^\sharp, \forall x, y \in \mathrm{Val}, x \in \gamma\left(x^\sharp\right) \wedge y \in \gamma\left(y^\sharp\right) \Rightarrow x\ binop\ y \in \gamma\left(\llbracket binop \rrbracket^\sharp{}_{\mathrm{binop}}\left(x^\sharp, y^\sharp\right)\right);$$
$$\forall x^\sharp, y^\sharp, r^\sharp, x'^\sharp, y'^\sharp \in \mathrm{Val}^\sharp, \forall x, y \in \mathrm{Val}, x \in \gamma\left(x^\sharp\right) \wedge y \in \gamma\left(y^\sharp\right) \wedge x\ binop\ y \in \gamma\left(r^\sharp\right) \wedge \left(x'^\sharp, y'^\sharp\right) = \llbracket binop \rrbracket\!\downarrow^\sharp{}_{\mathrm{binop}}\left(x^\sharp, y^\sharp, r^\sharp\right) \Rightarrow x \in \gamma\left(x'^\sharp\right) \wedge y \in \gamma\left(y'^\sharp\right);$$
$$\forall b^\sharp, x^\sharp, y^\sharp \in \mathrm{Val}^\sharp, \forall x \in \mathrm{Val}, \mathrm{true} \in \gamma\left(b^\sharp\right) \wedge x \in \gamma\left(x^\sharp\right) \Rightarrow x \in \gamma\left(\llbracket \mathrm{ite} \rrbracket^\sharp{}_{\mathrm{ite}}\left(b^\sharp, x^\sharp, y^\sharp\right)\right);$$
$$\forall b^\sharp, x^\sharp, y^\sharp \in \mathrm{Val}^\sharp, \forall y \in \mathrm{Val}, \mathrm{false} \in \gamma\left(b^\sharp\right) \wedge y \in \gamma\left(y^\sharp\right) \Rightarrow y \in \gamma\left(\llbracket \mathrm{ite} \rrbracket^\sharp{}_{\mathrm{ite}}\left(b^\sharp, x^\sharp, y^\sharp\right)\right);$$
$$\forall b^\sharp, x^\sharp, y^\sharp, r^\sharp, b'^\sharp, x'^\sharp \in \mathrm{Val}^\sharp, \forall x \in \mathrm{Val}, \mathrm{true} \in \gamma\left(b^\sharp\right) \wedge x \in \gamma\left(x^\sharp\right) \wedge x \in \gamma\left(r^\sharp\right) \wedge \left(b'^\sharp, x'^\sharp, \_\right) = \llbracket \mathrm{ite} \rrbracket\!\downarrow^\sharp{}_{\mathrm{ite}}\left(b^\sharp, x^\sharp, y^\sharp, r^\sharp\right) \Rightarrow \mathrm{true} \in \gamma\left(b'^\sharp\right) \wedge x \in \gamma\left(x'^\sharp\right);$$
$$\forall b^\sharp, x^\sharp, y^\sharp, r^\sharp, b'^\sharp, y'^\sharp \in \mathrm{Val}^\sharp, \forall y \in \mathrm{Val}, \mathrm{false} \in \gamma\left(b^\sharp\right) \wedge y \in \gamma\left(y^\sharp\right) \wedge y \in \gamma\left(r^\sharp\right) \wedge \left(b'^\sharp, \_, y'^\sharp\right) = \llbracket \mathrm{ite} \rrbracket\!\downarrow^\sharp{}_{\mathrm{ite}}\left(b^\sharp, x^\sharp, y^\sharp, r^\sharp\right) \Rightarrow \mathrm{false} \in \gamma\left(b'^\sharp\right) \wedge y \in \gamma\left(y'^\sharp\right).$$

**Lemma B.1** *For all expression $e$, $\llbracket e \rrbracket_{\mathcal{P}} \circ \gamma_{\mathrm{Env}} \dot{\subseteq} \gamma_{\mathrm{Env}} \circ \left(\lambda X.\, \llbracket e \rrbracket\!\downarrow^\sharp\left(\mathrm{true}^\sharp, X\right)\right)$.*

**Proof** This follows from the following property, which can be proved by structural induction on $e$:

$$\forall e, \forall n^\sharp, \left(\lambda X.\left\{\rho \in X \;\middle|\; \llbracket e \rrbracket(\rho) \in \gamma\left(n^\sharp\right)\right\}\right) \circ \gamma_{\mathrm{Env}} \dot{\subseteq} \gamma_{\mathrm{Env}} \circ \left(\lambda X.\, \llbracket e \rrbracket\!\downarrow^\sharp\left(n^\sharp, X\right)\right) \wedge \left(\lambda X.\{\llbracket e \rrbracket(\rho) \mid \rho \in X\}\right) \circ \gamma_{\mathrm{Env}} \dot{\subseteq} \gamma \circ \llbracket e \rrbracket^\sharp$$

□

**Lemma B.2** *If $f \in \mathrm{Env} \to \mathrm{Env}$ is monotonic (i.e. for all $x, y \in \mathrm{Env}$, if $x \subseteq y$ then $f(x) \subseteq f(y)$) and $f^\sharp : \mathrm{Env}^\sharp \to \mathrm{Env}^\sharp$ is a sound abstraction of $f$ (i.e. $f \circ$*

[5] Those assumptions have to be proved against the actual implementation of abstract domain $\mathrm{Val}^\sharp$ and its abstract transformers.

$\gamma_{\text{Env}} \dot{\subseteq} \gamma_{\text{Env}} \circ f^\sharp$), *then* $\text{gfp} f \subseteq \gamma_{\text{Env}} \left( \prod^\sharp_{n\ \text{Env}^\sharp} f^{\sharp n}(\top_{\text{Env}^\sharp}) \right)$.

**Proof** By induction on $n$ we have: $\forall n \in \mathbb{N}, f^n \left(\gamma_{\text{Env}} \left(\top_{\text{Env}^\sharp}\right)\right) \subseteq \gamma_{\text{Env}} \left(f^{\sharp n} \left(\top_{\text{Env}^\sharp}\right)\right)$. $f$ being monotonic on a complete lattice, $\text{gfp} f$ exists and $\text{gfp} f \subseteq \bigcap_n f^n(\top_{\text{Env}})$ hence $\text{gfp} f \subseteq \bigcap_n \gamma_{\text{Env}} \left(f^{\sharp n}(\top_{\text{Env}^\sharp})\right)$ by the previous property and knowing that $\gamma_{\text{Env}}(\top_{\text{Env}^\sharp}) = \top_{\text{Env}}$. Finally $\text{gfp} f \subseteq \gamma_{\text{Env}} \left( \prod^\sharp_{n\ \text{Env}^\sharp} f^{\sharp n}(\top_{\text{Env}^\sharp}) \right)$ since $\sqcap^\sharp_{\text{Env}^\sharp}$ is a sound abstraction of $\cap_{\text{Env}}$. □

**Proof** [Theorem 4.2] For any expression $e$, $[\![e]\!]_{\mathcal{P}}$ was proved monotonic in section 4.2.1 and from lemma B.1 $\lambda X.[\![e]\!]\downarrow^\sharp \left(\text{true}^\sharp, X\right)$ is a sound abstraction of the former, hence the result by lemma B.2. □