SMT-AI: an Abstract Interpreter for a Synchronous Extension of SMT-lib

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 D0178 for avionics [15]. Among those systems, control command software is often written in the well suited synchronous paradigm [2,10], hence a strong interest in proving functional properties on synchronous systems.

The next section will introduce a combination of analysis methods while the remaining of the article focus on the implementation of one of them. Section 3 on the following page describes the input language of our tools. Then the two following sections deal with the analysis it performs. Finally, section 6 on page 11 gives some details about the implementation and section 7 on page 11 concludes and gives an overview of what remains to be done.

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

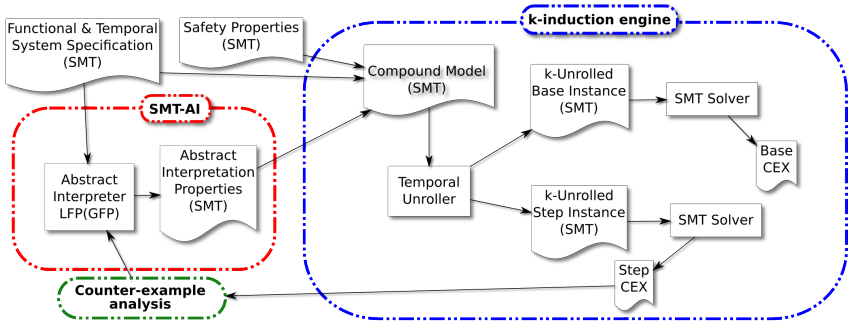

Figure 1. General combination of analyzes k-induction based

2 A combination of analysis

The k-induction method [16] is a SAT/SMT-based model checking technique. It aims at proving by induction a property on the analyzed system, possibly stronger than the one expected. It proved effective [9] but often requires the user to strengthen the description of the analyzed system with additional lemmas.

Abstract interpretation [5] 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].

Our goal is to use an abstract interpreter to automatically infer invariants of the system which can be used as lemmas by a k-induction tool. This combination is sketched in Figure 1. This paper will only deal with the implementation of the abstract interpreter.

Some works such as [3,8] combine k-induction with other methods to strengthen a particular invariant.

3 Input language

3.1 Syntax

Synchronous systems analyzed by our tool are expressed in the QF-UFLIA fragment of SMT-lib [1] extended with a few predicates for temporal aspects:

- `init` is true at initial step then always false, this would be written `true -> false` in Lustre [10] for example;
- `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.

3.2 Semantic

We already describe semantics of temporal predicates `init`, `memi` and `memu` in previous section and we will here focus 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 ρ:

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

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

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

4 Abstract Interpretation

4.1 Synchronous Systems Main Loop

Abstract semantics is computed through classical least fixpoint iterations with widening and narrowing, `memi` and `memu` predicates acting as assignment 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, as explained in next section.

4.2 Analysis of Expressions

This analysis is done by a greatest fixpoint computation.

Since our goal is to compute by abstract interpretation an over-approximation of this semantic, 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 already gave a concrete semantics in section 3.2 on the preceding page but let's define another one, more suitable for abstract interpretation.

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

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

$$[\![\mathit{binop}\ e_1\ e_2]\!](\rho) = [\![e_1]\!](\rho)\ \mathit{binop}\ [\![e_2]\!](\rho)$$
$$\text{for } \mathit{binop} \in \{\text{and}, \text{or}, =, <, >, \leq, \geq, -, +, *\}$$

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

$$[\![\text{let } (v\ e_1)\ e_2]\!](\rho) = [\![e_2]\!](\rho[v \mapsto [\![e_1]\!](\rho)])$$

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

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

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

Lemma 4.1 *For all $\rho : \mathbb{V} \to \mathrm{Val}$, for all expression e and $c \in \mathrm{Val}$, $[\![e]\!](\rho) = c$ if and only if $\rho \vDash e, c$.*

Proof By structural induction on e. □

Theorem 4.2 *Both semantics are equivalent: for all expression e, $[\![e]\!]_1 = [\![e]\!]_2$.*

Proof 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 4.1. Therefore the 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 4.1 which amounts to say that $\rho \in [\![e]\!]_1$. □

4.2.2 A Combined Forward-Backward Abstract Semantic

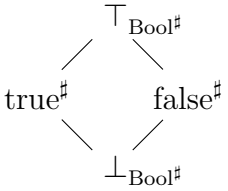

Figure 2: Lattice underlying boolean abstract domain $\text{Bool}^\sharp$.

We present here a non relational abstraction. 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 $\text{Val}^\sharp$, abstract transformers on this abstract domain will be presented in next paragraph. An abstract domain $\text{Env}^\sharp$ is also used as an abstraction of $\mathbb{V} \to \text{Val}$. For $\rho^\sharp \in \text{Env}^\sharp$, we write $\rho^\sharp(v)$ the abstract value attached to variable v and $\rho^\sharp[v \mapsto c^\sharp]$ the environment equal to $\rho^\sharp$ for each variable except for v where it takes abstract value $c^\sharp \in \text{Val}^\sharp$.

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 $c \in \text{Val}$;
- $[\![\text{not}]\!]\uparrow^\sharp_{\text{unop}} : \text{Val}^\sharp \to \text{Val}^\sharp$ and , forward abstract semantic of not;
- $[\![\text{not}]\!]\downarrow^\sharp_{\text{unop}} : \text{Val}^\sharp \times \text{Val}^\sharp \to \text{Val}^\sharp$, backward abstract semantic of not;
- $[\![\mathit{binop}]\!]\uparrow^\sharp_{\text{binop}} : \text{Val}^\sharp \times \text{Val}^\sharp \to \text{Val}^\sharp$ for $\mathit{binop} \in \{\text{and}, \text{or}, =, <, >, \leq, \geq, -, +, *\}$;
- $[\![\mathit{binop}]\!]\downarrow^\sharp_{\text{binop}} : \text{Val}^\sharp \times \text{Val}^\sharp \times \text{Val}^\sharp \to \text{Val}^\sharp \times \text{Val}^\sharp$ for $\mathit{binop} \in \{=, <, >, \leq, \geq, -, +, *\}$;
- $[\![\text{ite}]\!]\uparrow^\sharp_{\text{ite}} : \text{Val}^\sharp \times \text{Val}^\sharp \times \text{Val}^\sharp \to \text{Val}^\sharp$;
- $[\![\text{ite}]\!]\downarrow^\sharp_{\text{ite}} : \text{Val}^\sharp \times \text{Val}^\sharp \times \text{Val}^\sharp \times \text{Val}^\sharp \to \text{Val}^\sharp \times \text{Val}^\sharp \times \text{Val}^\sharp$.

They are formally specified later in section 2 on page 7.

Forward Abstract Semantic

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

$[\![v]\!]\uparrow^\sharp(\rho^\sharp) = \rho^\sharp(v) \quad [\![\text{not } v]\!]\uparrow^\sharp(\rho^\sharp) = [\![\text{not}]\!]\uparrow^\sharp_{\text{unop}}(\rho^\sharp(v)) \quad [\![c]\!]\uparrow^\sharp(\rho^\sharp) = \text{const}^\sharp(c)$ for $c \in \text{Val}$
$[\![\mathit{binop}\ e_1\ e_2]\!]\uparrow^\sharp(\rho^\sharp) = [\![\mathit{binop}]\!]\uparrow^\sharp_{\text{binop}}([\![e_1]\!]\uparrow^\sharp(\rho^\sharp), [\![e_2]\!]\uparrow^\sharp(\rho^\sharp))$
for $\mathit{binop} \in \{\text{and}, \text{or}, =, <, >, \leq, \geq, -, +, *\}$

$[\![\text{ite } e_b\ e_t\ e_e]\!]\uparrow^\sharp(\rho^\sharp) = [\![\text{ite}]\!]\uparrow^\sharp_{\text{ite}}([\![e_b]\!]\uparrow^\sharp(\rho^\sharp), [\![e_t]\!]\uparrow^\sharp(\rho^\sharp_t), [\![e_e]\!]\uparrow^\sharp(\rho^\sharp_e))$
where $\rho^\sharp_t = [\![e_b]\!]\downarrow^\sharp(\text{true}^\sharp, \rho^\sharp)$ and $\rho^\sharp_e = [\![e_b]\!]\downarrow^\sharp(\text{false}^\sharp, \rho^\sharp)$
$[\![\text{let } (v\ e_1)\ e_2]\!]\uparrow^\sharp(\rho^\sharp) = [\![e_2]\!]\uparrow^\sharp(\rho^\sharp[v \mapsto [\![e_1]\!]\uparrow^\sharp(\rho^\sharp)])$

Backward Abstract Semantic

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

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

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

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

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

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

$[\![\text{ite } e_b\ e_t\ e_e]\!]\downarrow^\sharp(n^\sharp, \rho^\sharp) = [\![e_b]\!]\downarrow^\sharp(n^\sharp_b, \rho^\sharp) \sqcap^\sharp_{\text{Env}^\sharp} \left([\![e_t]\!]\downarrow^\sharp(n^\sharp_t, \rho^\sharp_t) \sqcup^\sharp_{\text{Env}^\sharp} [\![e_e]\!]\downarrow^\sharp(n^\sharp_e, \rho^\sharp_e)\right)$
where $\rho^\sharp_t = [\![e_b]\!]\downarrow^\sharp(\text{true}^\sharp, \rho^\sharp)$ and $\rho^\sharp_e = [\![e_b]\!]\downarrow^\sharp(\text{false}^\sharp, \rho^\sharp)$
and $(n^\sharp_b, n^\sharp_t, n^\sharp_e) = [\![\text{ite}]\!]\downarrow^\sharp_{\text{ite}}([\![e_b]\!]\uparrow^\sharp(\rho^\sharp), [\![e_t]\!]\uparrow^\sharp(\rho^\sharp_t), [\![e_e]\!]\uparrow^\sharp(\rho^\sharp_e), n^\sharp)$
$[\![\text{let } (v\ e_1)\ e_2]\!]\downarrow^\sharp(n^\sharp, \rho^\sharp) = [\![e_1]\!]\downarrow^\sharp(\rho^\sharp_1(v), \rho^\sharp_1[v \mapsto \rho^\sharp(v)])$
where $\rho^\sharp_1 = [\![e_2]\!]\downarrow^\sharp(n^\sharp, \rho^\sharp[v \mapsto [\![e_1]\!]\uparrow^\sharp(\rho^\sharp)])$

The abstract semantics is finally computed through decreasing iterations:

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

Such backward semantics is commonly applied throughout the literature [6, §4.3] 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 4.3 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 $\gamma : \mathrm{Val}^\sharp \to 2^{\mathrm{Val}}$. 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 $\gamma_{\mathrm{Env}} : \mathrm{Env}^\sharp \to 2^{\mathbb{V}\to\mathrm{Val}}$ for environments: $\gamma_{\mathrm{Env}} = \rho^\sharp \mapsto \left\{\rho \mid \forall v \in \mathbb{V}.\rho(v) \in \gamma(\rho^\sharp(v))\right\}$. Since we want to compute a sound over-approximation [2] of the concrete semantic, we have to prove that for any expression e:

$$[\![e]\!]_2 \subseteq \gamma_{\mathrm{Env}}\left([\![e]\!]^\sharp\right)$$

Hypotheses

γ is assumed to satisfy the following properties:
$\gamma(\bot_{\mathrm{Val}^\sharp}) = \emptyset$ and $\gamma(\top_{\mathrm{Val}^\sharp}) = \mathrm{Val}$;
γ is monotonous: $\forall x, y \in \mathrm{Val}^\sharp$ 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 γ_{Env} inherits from the three first properties according to its definition.

Moreover, the basic abstract transformers are expected to fulfill following soundness specifications [3]:
$\forall c \in \mathrm{Val}$, $c \in \gamma\left(const^\sharp(c)\right)$;
$\forall x^\sharp \in \mathrm{Val}^\sharp, \forall x \in \mathbb{B}, x \in \gamma\left(x^\sharp\right) \Rightarrow \neg x \in \gamma\left([\![\mathrm{not}]\!]\uparrow^\sharp_{\mathrm{unop}}\left(x^\sharp\right)\right)$;
$\forall x^\sharp, r^\sharp \in \mathrm{Val}^\sharp, \forall x \in \mathbb{B}, x \in \gamma\left(x^\sharp\right) \wedge \neg x \in \gamma\left(r^\sharp\right) \Rightarrow x \in \gamma\left([\![\mathrm{not}]\!]\downarrow^\sharp_{\mathrm{unop}}\left(x^\sharp, r^\sharp\right)\right)$;
$\forall x^\sharp, y^\sharp \in \mathrm{Val}^\sharp, \forall x, y \in \mathrm{Val}, x \in \gamma\left(x^\sharp\right) \wedge y \in \gamma\left(y^\sharp\right) \Rightarrow x\ binop\ y \in \gamma\left([\![binop]\!]\uparrow^\sharp_{\mathrm{binop}}\left(x^\sharp, y^\sharp\right)\right)$;

[2] In particular if the abstract semantics computed is $\bot_{\mathrm{Env}^\sharp}$, 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 synchronous systems.
[3] Those assumptions have to be proved against the actual implementation of abstract domain $\mathrm{Val}^\sharp$ and its abstract transformers.

$\forall x^\sharp, y^\sharp, r^\sharp, x'^\sharp, y'^\sharp \in \mathrm{Val}^\sharp, \forall x, y \in \mathrm{Val}, x \in \gamma\left(x^\sharp\right) \wedge y \in \gamma\left(y^\sharp\right) \wedge x \; binop \; y \in \gamma\left(r^\sharp\right) \wedge \left(x'^\sharp, y'^\sharp\right) = [\![binop]\!]\downarrow^\sharp_{\mathrm{binop}}\left(x^\sharp, y^\sharp, r^\sharp\right) \Rightarrow x \in \gamma\left(x'^\sharp\right) \wedge y \in \gamma\left(y'^\sharp\right);$
$\forall b^\sharp, x^\sharp, y^\sharp \in \mathrm{Val}^\sharp, \forall x \in \mathrm{Val}, \mathrm{true} \in \gamma\left(b^\sharp\right) \wedge x \in \gamma\left(x^\sharp\right) \Rightarrow x \in \gamma\left([\![\mathrm{ite}]\!]\uparrow^\sharp_{\mathrm{ite}}\left(b^\sharp, x^\sharp, y^\sharp\right)\right);$
$\forall b^\sharp, x^\sharp, y^\sharp \in \mathrm{Val}^\sharp, \forall y \in \mathrm{Val}, \mathrm{false} \in \gamma\left(b^\sharp\right) \wedge y \in \gamma\left(y^\sharp\right) \Rightarrow y \in \gamma\left([\![\mathrm{ite}]\!]\uparrow^\sharp_{\mathrm{ite}}\left(b^\sharp, x^\sharp, y^\sharp\right)\right);$
$\forall b^\sharp, x^\sharp, y^\sharp, r^\sharp, b'^\sharp, x'^\sharp \in \mathrm{Val}^\sharp, \forall x \in \mathrm{Val}, \mathrm{true} \in \gamma\left(b^\sharp\right) \wedge x \in \gamma\left(x^\sharp\right) \wedge x \in \gamma\left(r^\sharp\right) \wedge \left(b'^\sharp, x'^\sharp, \_\right) = [\![\mathrm{ite}]\!]\downarrow^\sharp_{\mathrm{ite}}\left(b^\sharp, x^\sharp, y^\sharp, r^\sharp\right) \Rightarrow \mathrm{true} \in \gamma\left(b'^\sharp\right) \wedge x \in \gamma\left(x'^\sharp\right);$
$\forall b^\sharp, x^\sharp, y^\sharp, r^\sharp, b'^\sharp, y'^\sharp \in \mathrm{Val}^\sharp, \forall y \in \mathrm{Val}, \mathrm{false} \in \gamma\left(b^\sharp\right) \wedge y \in \gamma\left(y^\sharp\right) \wedge y \in \gamma\left(r^\sharp\right) \wedge \left(b'^\sharp, \_, y'^\sharp\right) = [\![\mathrm{ite}]\!]\downarrow^\sharp_{\mathrm{ite}}\left(b^\sharp, x^\sharp, y^\sharp, r^\sharp\right) \Rightarrow \mathrm{false} \in \gamma\left(b'^\sharp\right) \wedge y \in \gamma\left(y'^\sharp\right).$

Lemma 4.4 *For all expression e, $[\![e]\!]_{\mathcal{P}} \circ \gamma_{\mathrm{Env}} \dot{\subseteq} \gamma_{\mathrm{Env}} \circ \left(\lambda X.\, [\![e]\!]\downarrow^\sharp\left(\mathrm{true}^\sharp, X\right)\right)$.*

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

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

□

Lemma 4.5 *If $f \in \mathrm{Env} \to \mathrm{Env}$ is monotonous (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. $f \circ \gamma_{\mathrm{Env}} \dot{\subseteq} \gamma_{\mathrm{Env}} \circ f^\sharp$), then* $\mathrm{gfp} f \subseteq \gamma_{\mathrm{Env}}\left(\prod^\sharp_{n\ \mathrm{Env}^\sharp} f^{\sharp n}(\top_{\mathrm{Env}^\sharp})\right)$.

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

Theorem 4.6 *Our abstract semantics is sound with respect to the concrete semantic: for all expression e, $[\![e]\!]_2 \subseteq \gamma_{\mathrm{Env}}\left([\![e]\!]^\sharp\right)$.*

Proof For any expression e, $[\![e]\!]_{\mathcal{P}}$ was proved monotonous in section 4.2.1 on page 4 and from lemma 4.4 $\lambda X.[\![e]\!]\downarrow^\sharp\left(\mathrm{true}^\sharp, X\right)$ is a sound abstraction of the former, hence the result by lemma 4.5. □

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 $\text{Val}^\sharp$ 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 is usually reached after only a few iterations in practical cases.

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 $[\![e_1]\!]\downarrow^\sharp \left(n_1^\sharp, \rho^\sharp\right) \sqcap^\sharp_{\text{Env}^\sharp} [\![e_2]\!]\downarrow^\sharp \left(n_2^\sharp, \rho^\sharp\right)$ can also be obtained by computing one of them from the result of the other $[\![e_1]\!]\downarrow^\sharp \left(n_1^\sharp, [\![e_2]\!]\downarrow^\sharp \left(n_2^\sharp, \rho^\sharp\right)\right)$. Doing this for the `and` allows to save an iteration on example 4.3 on page 7.

5 Inlining and Partitioning

The code we analyze comes out from a common subexpression elimination phase and looks a bit like three address code. This makes lot of expression defined under let constructs 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.

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 3.1 on page 2, raw QF-UFLIA fragment of SMT-lib [1] plus a few extra predicates:

- `init`, `memu` and `memi` to enable description of synchronous systems (c.f. section 3.1 on page 2);
- `trace_partitioning` and `trace_merge` semantically equivalent to identities and used as pragmas for trace partitioning (c.f. section 5 on the previous page, 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 on the preceding page), 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 on page 3).

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

7 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 [4]. 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. In particular, we intend to use the APRON library [11] which offers relational abstract domains such as the octagons [12]. 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 also 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] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[2] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, Robert, and De Simone. The synchronous languages 12 years later. In *Proceedings of The IEEE*, pages 64–83, 2003.

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

[4] 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.

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

[6] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[7] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[8] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification (extended abstract, category a). In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2003.

[9] Bruno Dutertre and Maria Sorea. Timed systems in sal. Technical report, Computer Science Laboratory, 2004.

[10] 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.

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

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

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

[14] Xavier Rival and Laurent 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] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

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