Abstract Domains for Property Checking Driven Analysis of Temporal Properties

Damien Massé

École Normale Supérieure, Paris, France,
damien.masse@polytechnique.fr,
http://www.stix.polytechnique.fr/~dmasse/

Abstract. Abstract interpretation-based static analysis infers properties from the source code of a program. When the goal is to check a temporal specification on the program, we need the analysis to be as precise as possible to avoid false negatives. In previous work [9], we suggested a method called "property checking driven analysis" to automatically use the specification to check during the analysis in order to refine it. However, this approach requires to abstract domains of lower closure operators, something which was not developed. In this paper, we describe some abstractions on lower closure operators developed for a small analyzer of temporal properties. We examine the need for weak relational abstractions, and show that using our new approach can give more precise results than using a traditional abstract interpretation-based analysis with expensive abstract domains.

1 Introduction

The objective of static program analysis is to automatically infer run-time properties on programs at compile-time. Since the properties are often undecidable, static program analysis uses approximations. The acceptable level of approximation depends on the application of the static analyzer: for example, when the goal is to remove unnecessary run-time checks [4], missing a few possible optimizations is acceptable. However, when the goal is to prove some specifications about the program (e.g. the absence of run-time errors), the analysis must be very precise (to avoid any false negative), and efficient enough to check large programs. To obtain this result, an approach is to design a special-purpose static program analyzer adapted to a restricted class of programs and a restricted class of specifications. This method was proposed and successfully applied in [1], for the verification of real-time embedded software. However, the analyzer so designed handled only one specification (the absence of run-time error).

It may be possible to extend the class of specifications (e.g. to a larger class of temporal properties) if the specification itself is used to refine the analysis. In previous work [9], we proposed to use the specification in a different, "reverse" analysis (e.g. if the abstract semantics is computed with a backward analysis, the "reverse" analysis is a forward one) which computes a "guide" for the computation of the abstract semantics. This approach, which we called "property

checking driven analysis", uses domains of lower closure operators as the concrete description of the "guide" for the main analysis. Thus, implementing this method requires to design efficient and precise abstractions of such domains.

This paper extends [9] by showing how to construct such an abstract domain, and presents a prototype implementation. We show different abstractions of domains of lower closure operators, based on the structure of the underlying domains. Since a non-relational domain appears insufficient to keep the precision of the analysis, we develop a *weak relational domain* for abstract environments, which uses a notion of local dependence between variables related to the temporal specification. Finally we present the results of the analyzer on a few examples, and we discuss the merits of our approach compared with analyses using traditional abstract domains.

2 Concrete semantics

In this section, we introduce briefly the language and the kind of temporal specifications we intend to analyse.

2.1 Language and states

Since the inspiration for the analyzer is Cousot's Marktoberdorf generic analyzer [3], we analyse roughly the same language. It is a very simple imperative language, with only integers and without functions. Integer values range from `min_int` to `max_int`, and there is an arithmetic error Ω. We denote by $\mathbb{I}$ the interval $[\texttt{min\_int}, \texttt{max\_int}]$, and by $\mathbb{I}_\Omega$ the set $\mathbb{I} \cup \{\Omega\}$.

The language uses arithmetic expression Aexp, boolean expression Bexp, command Com and list of commands Seq. The syntax is defined as:

$$\begin{array}{lll} A \in \text{Aexp} & ::= & n \mid \text{x} \mid ?\,\texttt{in}\,[A_1, A_2] \mid \mathit{un}\,A \mid A_1\,\mathit{bin}\,A_2 \\ B \in \text{Bexp} & ::= & \texttt{true} \mid \texttt{false} \mid A_1 = A_2 \mid A_1 < A_2 \\ & & \mid B_1 \,\text{and}\, B_2 \mid B_1 \,\text{or}\, B_2 \\ C \in \text{Com} & ::= & \texttt{skip} \mid \text{x} := A \mid \texttt{if}\,B\,\texttt{then}\,S_1\,\texttt{else}\,S_2\,\texttt{fi} \\ & & \mid \texttt{while}\,B\,\texttt{do}\,S\,\texttt{od} \\ S \in \text{Seq} & ::= & C \mid C\,;\,S \end{array}$$

Here, n are integers, $x \in \mathbb{V}$ variables, $\mathit{un} \in \{+, -\}$ unary operators and $\mathit{bin} \in \{+, -, *, /, \text{mod}\}$ binary operators. The difference with the analyzer developed in [3] is the non-deterministic operation ? `in` $[A_1, A_2]$, which returns an integer between the value v_1 of A_1 and the value v_2 of A_2 if $v_1 \leq v_2$, and Ω if $v_1 > v_2$. This operation intends to model all non-deterministic aspects of the program, e.g. user inputs, sensors or configurations. The exact *meaning* of each non-deterministic operation (i.e. how it relates to the specification) is determined in the temporal specification.

Hence, a state $\sigma \in \Sigma$ is defined as a pair of a program point p (in a set of program points Lab) and an environement in $\mathbb{V} \to \mathbb{I}_\Omega$:

$$\Sigma = \text{Lab} \times (\mathbb{V} \to \mathbb{I}_\Omega)$$

2.2 Temporal specification and semantics

To simplify the abstraction, we choose to analyse only simple specifications which make the distinction between non-deterministic operations as "internal" and "external" non-determinism. These properties are expressed with μ-calculus formulas of the form (we identify atomic predicates and the states they represent):

$$I \models \frac{\mu}{\nu} X.(A \vee (B \wedge \Diamond X) \vee (C \wedge \Box X))$$

with $\frac{\mu}{\nu} \in \{\mu, \nu\}$. We can note that this class of temporal specification is stable by negation (i.e., if ϕ belongs to this class, so does $\neg\phi$).

Here, I are initial states, A final states, B states with "internal" non-determinism (the result of non-deterministic operations there can be chosen in order to satisfy the specification), and C states with "external" non-determinism (the specification must be satisfied for any result of the non-deterministic operations there). States which are not in $A \vee B \vee C$ are error states. For the sake of simplicity we assume that B and C are disjoint. Infinite computations are taken into account by the difference between μ and ν. This class of specifications includes all the **CTL** operators, and some kind of game specifications as well.

In a framework using states, we use the predicate transformers pre ($y \in preX$ iff at least one successor of y is in X) and $\widetilde{pre}$ ($y \in \widetilde{pre}X$ iff all successors of y are in X). Then the specification is (with lgfp being either lfp or gfp):

$$I \subseteq \text{lgfp } X.(A \cup (B \cap preX) \cup (C \cap \widetilde{pre}X)).$$

Since the class of specifications is stable by negation, we can choose to express the negation of the specification. Then it is expressed by the formula:

$$I \cap \text{lgfp } X.(A \cup (B \cap preX) \cup (C \cap \widetilde{pre}X)) = \emptyset \tag{1}$$

Then, the goal of the analyzer is to over-approximate $\mathcal{S} = I \cap \text{lgfp } X.(A \cup (B \cap preX) \cup (C \cap \widetilde{pre}X))$. If we get $\emptyset$, we prove the property. Otherwise, we get initial states which may not satisfy the specification.

2.3 Property checking driven analysis

The goal of our analyzer is to illustrate the technique of property checking driven analysis developed in [9]. In this section we give a summary of this technique. Starting from a concrete semantics $\mathcal{S} = I \cap \text{lgfp } \phi$, this method shows how to construct a lower closure operator[1] ρ such that:

$$I \cap \text{lgfp } (\rho \circ \phi) = I \cap \text{lgfp } \phi.$$

In this formula, ρ reduces the value obtained at each iteration of the fixpoint, while ensuring that the fixpoint will be greater than or equal to $\mathcal{S}$. Hence, ρ focuses the fixpoint computation on parts useful for the computation of $\mathcal{S}$.

[1] Lower closure operators are monotonic, reductive and idempotent. Basic results on lower closure operators are recalled in [9].

In practice, the fixpoints are transfered into the abstract domain using abstract interpretation results, and an over-approximation ρ' of ρ is used. When ρ' is included in the abstract fixpoint computation, it leads to more precise (and still sound) results for $\mathcal{S}$. Thus, the lower closure operator "drives" the analysis towards the verification of the specification. We can notice that this process does not change the abstraction itself, which is defined in the fixpoint transfer: it is not an abstraction refinement, but an analysis refinement.

Some notation is necessary to exhibit the formulas expressing ρ. The lattice of lower closure operators on a domain D is written $(lco\,(D)\,, \sqsubseteq)$. A lower closure operator ρ is characterized by the set of its fixpoints $\rho(D)$, which is an *upper Moore family*: $\rho(D) = \mathcal{M}\,(\rho(D)) = \{\cup A \mid A \subseteq \rho(D)\}$. As usual, ρ will denote either the operator or the set of its fixpoints $\rho(D)$, depending on the context.

With $\phi = \lambda X.(A \cup (B \cap pre X) \cup (C \cap \widetilde{pre} X))$, applying the results of [9] gives:

$$\begin{aligned} \rho_0 &= \lambda X.X \cap I \\ \rho &= \text{lfp } \eta.(\rho_0 \sqcup F_\phi(\eta)) \end{aligned}$$

where F_ϕ is a complex function defined in [9].

Here, ρ is constructed iteratively from ρ_0 (the fixpoints of which are the subsets of I, i.e. all the possible values for $\mathcal{S}$). Intuitively, at each iteration, F_ϕ add as new fixpoints the minimum sets which can lead, by an application of ϕ, to a superset of an existing fixpoint X of ρ. All the fixpoints of ρ will then form a sub-lattice of $\wp\,(\Sigma)$ on which we can restrict the computation of lgfp ϕ, without modifying the intersection of the fixpoint with I.

To create a minimum set Y leading to a superset of X, we first test if X is included in $A \cup B \cup C$. If this is not the case, then we cannot define any Y since $\phi(\Sigma) \not\supseteq X$. But if $X \subseteq A \cup B \cup C$, we construct a possible Y by choosing one successor for each element of $X \setminus B$, and all the successors of $X \setminus C$. Then these sets are added as new fixpoints.

As the approximation of ρ involves only looking for successors of states, we can consider it as our "forward" analysis (it starts from I and goes forward) which will "drive" the "backward" analysis lgfp ϕ. We show in [9] that this process can be iterated (as the result of the backward analysis can be used to get more precise guide) until a fixpoint is obtained. In practice, as was observed previously in other abstract interpretation-based analyzers which use backward-forward combination (e.g. in [2]), this fixpoint is reached after a few iterations.

3 Construction of the abstract domain

Of course, ρ is not computable, we use abstract interpretation to approximate it. Hence, we need to find abstractions on the domain of lower closure operators[2]. Identifying ρ by the set of its fixpoints, an over-approximation of ρ is simply a superset of ρ.

[2] Note that this abstraction is orthogonal to the abstraction of $\wp\,(\Sigma)$ used in the computation of the abstract semantics.

Given a set of states X, we can see the application of ρ on X as the removal of states in X. A state σ may be removed for two reasons:

- The first case is that $\sigma \notin \rho(\Sigma)$. Then, σ is not in any fixpoint of ρ, and σ is totally useless for the computation of $I \cap \text{lgfp } \rho \circ \phi$. This happens if σ never appears in the computation of the "successors" of I.
- There are some fixpoints of ρ which contain σ, but none is included in X. In this case, σ is removed because it is useless *without* other states. Such a state may be removed at some point during the computation of lgfp $\rho \circ \phi$, and reintroduced later in the computation along with other states such that it is not removed. The construction of ρ presented in the previous section guarantees that any "useful" state for the computation of $I \cap \text{lgfp } \phi$ will eventually appear in the computation of lgfp $\rho \circ \phi$.

This distinction gives a starting point for the development of approximations for lower closures. The states appearing in our first case can be obtained through an approximation of $\rho(\Sigma)$. As it is a subset of Σ, we can do that with any existing abstraction of Σ. To remove other states, we need some kind of dependence relations between useful states. Hence, a main issue of this work is to express the dependence (related to the temporal specification) between states.

3.1 "Interval" abstraction

The "interval" abstraction is the only one described in [9]. From an abstraction of P to $P^\sharp$ we derive an abstraction of $lco\,(P)$ where each lower closure ρ is represented by two elements of $P^\sharp$: an "upper" one which abstracts the greatest fixpoint of ρ, and a "lower" one which gives a lower bound on the non-empty fixpoints of ρ.

Formally, from a Galois connection $P \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (P^\sharp, \sqsubseteq^\sharp)$, we construct a new Galois connection $lco\,(P) \underset{\alpha^\bullet}{\overset{\gamma^\bullet}{\leftrightarrows}} (P^\sharp, \sqsupseteq^\sharp) \times (P^\sharp, \sqsubseteq^\sharp)$ defined as:

$$\alpha^\bullet(\rho) = (\sqcap^\sharp\{\alpha(Y) \mid Y \in \rho \setminus \{\emptyset\}\},\ \alpha(\rho(\Sigma)))$$
$$\gamma^\bullet(l, u) = \{X \mid l \sqsubseteq^\sharp \alpha(X) \sqsubseteq^\sharp u\}$$

With the functional view of lower closures, $\gamma^\bullet$ becomes:

$$\gamma^\bullet(l, u) = \lambda X. \begin{cases} X \cap \gamma(u) & \text{if } \alpha(X) \sqsupseteq^\sharp l \\ \emptyset & \text{otherwise} \end{cases}$$

Hence the lower bound gives the relations between the elements of P. The precision of these relations relies on the precision of the abstract intersection: $\{X \in P \mid \alpha(X) \sqsubseteq^\sharp \sqcap^\sharp E\}$ should be the smallest possible for a given E.

Example 1. Applying this construction directly on a whole existing abstraction of $\wp(\Sigma)$ is difficult and gives imprecise results (because the abstract intersection is not precise enough). Rather, we use it on abstractions of basic elements, and we use the result as a base for the construction of our abstract domain.

An example is given in [9] for integers, using the interval domain. We use the same approach to abstract $lco\,(\wp\,(\mathbb{I}))$: the initial abstract domain is $(\mathbb{I}\times\mathbb{I}, \sqsubseteq)$, with $(a_1, b_1) \sqsubseteq (a_2, b_2) \iff (a_2 \leq a_1) \wedge (b_1 \leq b_2)$ (this is an extension of the interval domain, as we do not restrict a_i to be less than b_i), and $\alpha(X) = \min\ X, \max\ X$.

Then a lower closure ρ is abstracted by two intervals $((\mathrm{Mm}, \mathrm{mM}), (\mathrm{mm}, \mathrm{MM}))$[3], such that:

$$\forall X \subseteq \mathbb{I},\ \rho(X) \subseteq \begin{cases} \emptyset & \text{if } \min X > \mathrm{Mm} \text{ or } \max X < \mathrm{mM} \\ X \cap [\mathrm{mm}, \mathrm{MM}] & \text{otherwise} \end{cases}$$

3.2 Abstraction of $lco\,(\wp\,(\mathbb{I}_\Omega))$

The abstraction of $lco\,(\wp\,(\mathbb{I}_\Omega))$ can be deduced from an abstraction $lco\,(\wp\,(\mathbb{I}))$, using the fact that $\mathbb{I}_\Omega = \mathbb{I} \cup \{\Omega\}$. More generally, we study possible abstractions of $lco\,(\wp\,(A \cup B))$ (with A and B disjoint) using $lco\,(\wp\,(A))$ and $lco\,(\wp\,(B))$. A very simple abstraction is the non-relational abstraction:

Proposition 1. *Let A, B be two sets with $A \cap B = \emptyset$. Then $lco\,(\wp\,(A \cup B)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} lco\,(\wp\,(A)) \times lco\,(\wp\,(B))$ defined as:*

$$\alpha(\rho) = (\lambda X.\rho(X \cup B) \cap A), (\lambda Y.\rho(A \cup Y) \cap B)$$
$$\gamma(\rho_A, \rho_B) = (\lambda Z.\rho_A(Z \cap A) \cup \rho_B(Z \cap B))$$

is a Galois connection.

This abstraction is non-relational because it does not keep any constraints between the two sets. Applied for our domain of values, it would abstract $lco\,(\wp\,(\mathbb{I}_\Omega))$ to $lco\,(\wp\,(\mathbb{I})) \times \{\rho_\emptyset, \rho_\Omega\}$, such that $\rho_\emptyset = \lambda x.\emptyset$ means that no error appears in any fixpoint, whereas $\rho_\Omega = \lambda x.x$ means that some fixpoints have arithmetic errors.

However, it may be useful to know facts like "all non-empty fixpoints have arithmetic errors" (in this case, we can stop the analysis), so we will use three "possible error values" instead of two. To get this abstraction, we define the set $T = \{\mathrm{INI}, \mathrm{ERR}, \mathrm{TOP}\}$, with the following intuitive meaning:

- INI means that no error is possible. The lower closure associated ρ satisfies $\rho(X) = \rho(X \setminus \{\Omega\})$ for all $X \subseteq \mathbb{I}_\Omega$.
- ERR means that all elements of ρ, except $\emptyset$, contains Ω. Hence this is the "error" case.
- TOP is the "do not know" answer.

Then we construct an abstraction α from $lco\,(\wp\,(\mathbb{I}_\Omega))$ to $lco\,(\wp\,(\mathbb{I})) \times T$. Noting $\alpha_1(\rho), \alpha_2(\rho)$ the two components of the abstraction, we define:

$$\alpha_1(\rho) = \lambda X.\rho(X \cup \{\Omega\}) \cap \mathbb{I}$$
$$\alpha_2(\rho) = \begin{cases} \mathrm{INI} & \text{if } \forall X \subseteq \mathbb{I}_\Omega, \rho(X) = \rho(X \setminus \{\Omega\}) \\ \mathrm{ERR} & \text{if } \rho \neq \lambda X.\emptyset\ \wedge \forall X \subseteq \mathbb{I}, \rho(X) = \emptyset \\ \mathrm{TOP} & \text{otherwise} \end{cases}$$

[3] Which stands for ((max min, min max), (min min, max max)), as these are in fact the bounds of the bounds of the non-empty fixpoints of ρ.

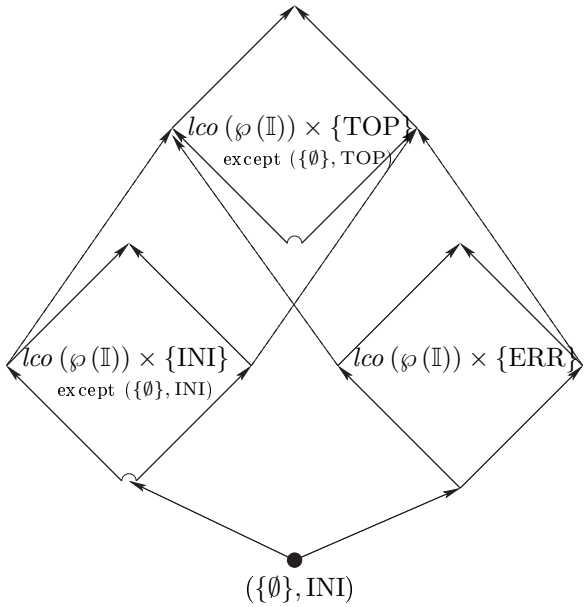

Fig. 1. Lattice $lco\,(\wp\,(\mathbb{I})) \times T$, divided in three parts, one for each value of T. Note that $(\{\emptyset\}, \mathrm{INI})$ is the global bottom, and that $(\{\emptyset\}, \mathrm{TOP})$ is collapsed with $(\{\emptyset\}, \mathrm{ERR})$.

Then $lco\,(\wp\,(\mathbb{I})) \times T$ can be defined as a lattice with the structure defined Fig. 1, and α is the abstraction of a Galois connection.

Our abstract domain of values is then $\mathbb{I}_{\mathrm{int}} \times T$. This loses the relation between the Ω and the non-error values in a same set in ρ. However, we keep the option "error everywhere", which enables to know whether the error is not avoidable.

3.3 Abstract environment

To abstract an environment, we need to abstract lower closures on powerset of Cartesian products. There is an intuitive non-relational abstraction (i.e. which abstracts each variable separately), but we will see that it is not sufficient in general, as it loses dependency information between source of non-determinism. Hence we will describe a *weak relational abstraction* which expresses the dependence between the possible values of several variables.

Proposition 2 (Non-relational abstraction). *Let A, B be two sets, and $\pi_A : \wp\,(A \times B) \to \wp\,(A)$, $\pi_B : \wp\,(A \times B) \to \wp\,(B)$ the projections of subsets of $A \times B$ on their components. Then $lco\,(\wp\,(A \times B)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} lco\,(\wp\,(A)) \times lco\,(\wp\,(B))$ defined as:*

$$\alpha(\rho) = (\lambda X.\pi_A(\rho(X \times B))), (\lambda Y.\pi_B(\rho(A \times Y)))$$
$$\gamma(\rho_A, \rho_B) = (\lambda Z.\rho_A(\pi_A(Z)) \times \rho_B(\pi_B(Z)))$$

is a Galois connection.

Hence, we can produce a non-relational abstraction of the whole environment, but this abstraction is not sufficient. Let us look at the following program:

```
x := ? in [0,1] ;
y := ? in [0,1] ;
z := x + y
```

If we want to prove that, by choosing the value for x and y, we can satisfy $z = 2$ at the end of the program, we must know that x and y are independent before computing z (e.g. we do not have $x = 1 - y$). However, with a completely non-relational abstraction, we know that x and y can satisfy $x = 1$ and $y = 1$, but we do not know that these properties can be true simultaneously. The forward analysis would give the same result if we replace the second line by $y := 1 - x$.

This limitation makes the analysis much less useful. Thus, we need to keep a kind of independence relation between variables, which is used to know that the constraints expressed on each variable are effective simultaneously. We do this by a weak relational abstraction, keeping only a relation between each variables.

Weak relational abstraction: two variables case First, we give a weak abstraction of $lco\,(\wp\,(\mathbb{I}_\Omega)) \times lco\,(\wp\,(\mathbb{I}_\Omega))$ (like with two variables). Like the non-relational abstraction, we keep an abstract value for each component, but we add a boolean expressing the dependence between the components (*true* means that the components may depend on each other). Saying that x and y are independent in a lower closure ρ means that ρ is a Moore family generated by Cartesian products of sets of values (i.e., all the fixpoints of ρ are union of the Cartesian products generated by the fixpoints of the abstract values of each components).

Hence, the abstract domain is $lco\,(\wp\,(\mathbb{I}_\Omega)) \times lco\,(\wp\,(\mathbb{I}_\Omega)) \times \mathbb{B}$. The concretization of $(\rho_x, \rho_y, \mathit{false})$, expressed as a Moore family, is $\rho = \mathcal{M}\,(X \times Y \mid X \in \rho_x \wedge Y \in \rho_y)$: ρ is generated by Cartesian products. The concretization of $(\rho_x, \rho_y, \mathit{true})$ is $\rho = \{Z \in \wp\,(\mathbb{I}_\Omega) \times \wp\,(\mathbb{I}_\Omega) \mid \pi_x(Z) \in \rho_x \wedge \pi_y(Z) \in \rho_y\}$ with $\pi_x(Z)$ (resp. $\pi_y(Z)$) the projection of Z to its first (resp. second) component: this is the non-relational concretization of (ρ_x, ρ_y).

Example 2. To illustrate our abstraction, we restrict the values to $S = \{0, 1\}$. We want to study an abstraction of $lco\,(\wp\,(S \times S))$ to $lco\,(\wp\,(S)) \times lco\,(\wp\,(S)) \times \mathbb{B}$, or, more precisely, we want to describe the meaning of the abstract values. To simplify, we suppose that the first component of the abstract value is $\{\emptyset, \{0, 1\}\}$, and that the second satisfies $\rho(\{0, 1\}) = \{0, 1\}$. Still, we have eight possible cases (four values for the second lower closure, and two for the boolean), all described in Fig. 2, with the order between them. When the boolean is *false*, the values are independent, which means that all the generators of the lower closure are Cartesian products, whereas when the boolean is *true* we can keep any generator.

The relation is very weak, and its meaning is restricted to the "lower" part of the abstraction, but it is sufficient to keep the independence of the random generators, which was our goal. In particular, the abstract functions will be easier to compute than with stronger relations like octagons[10].

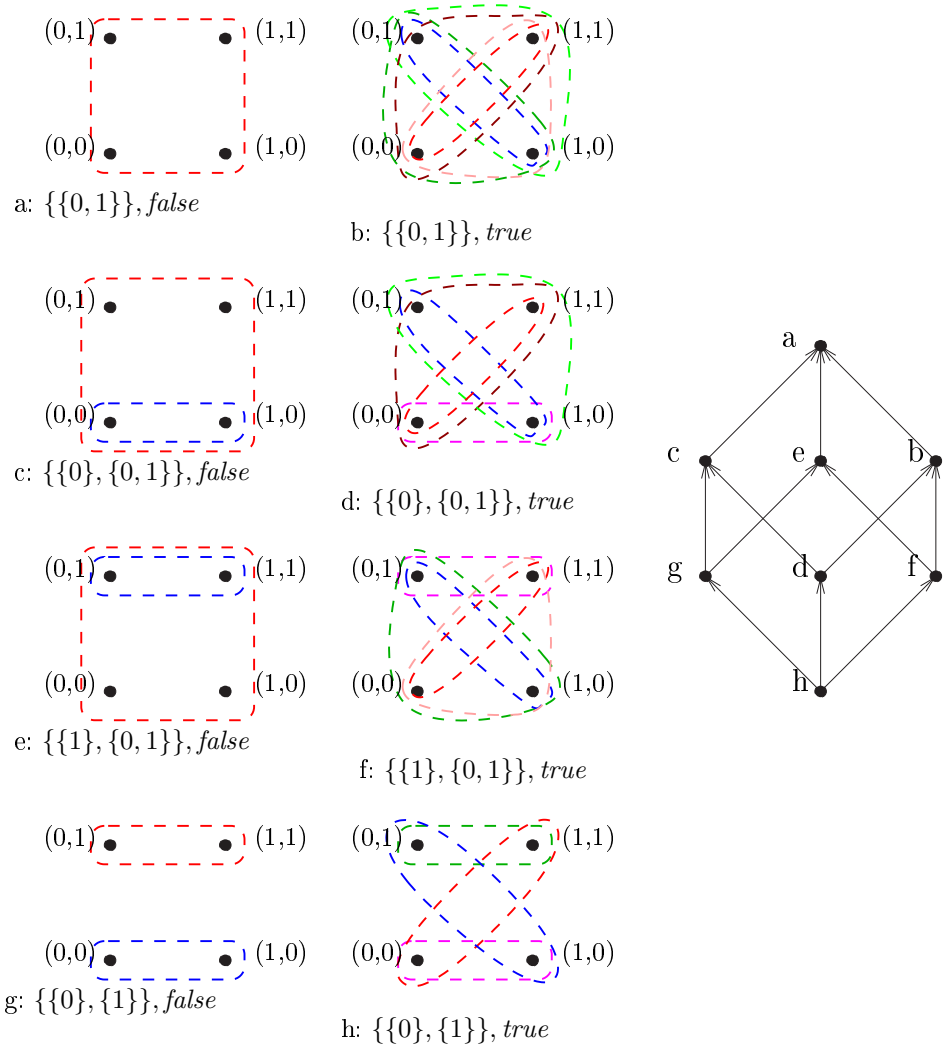

Fig. 2. The eight cases of example 2. Each case is described by the value of the boolean, and the generators of the second lower closure operator in the abstract value. For each case, we give the generators of the concretized lower closure. We give also the order between them. We can see that adding the independence boolean (case *false*) restrict greatly the concrete lower closure.

Weak relational abstraction: general case Our first generic abstract domain is $(\mathbb{V} \to \mathit{lco}\,(\wp\,(\mathbb{I}_\Omega))) \times \wp\,(\wp\,(\mathbb{V}))$, with the concretization:

$$\gamma(f, B) = \{Z \mid \forall x \in \mathbb{V}, \pi_x(Z) \in f(x) \wedge \forall V \notin B, \pi_V(Z) = \times_{x \in V} \pi_x(Z)\}$$

The principle is to associate a boolean to any subset V of $\mathbb{V}$, describing the idea that these variables are "globally" dependent. Note that three variables may be "independent" when taken pairwise without being "globally" independent (a set of points in dimension 3 may look like a Cartesian product when projected in any direction, but not be itself a Cartesian product, cf. Fig 3 for an example). Thus we cannot deduce exactly the "global" dependence function from the dependence relations between pairs of variables.

However, keeping an element of $\wp\,(\wp\,(\mathbb{V})))$ is too costly and, in practice, it is not useful[4]. Rather, we keep only a symmetric relation between variables (i.e. a subset of $\wp\,(\mathbb{V} \times \mathbb{V})$) expressing the dependence of the variables, but such that all sets of variables completely unrelated must be globally independent (thus, this is a subset of the real dependence relations between each couple of variables, but from it we can reconstruct the whole set of dependencies). This construction can be viewed as an abstraction (as we construct a sound, but not complete, "global" dependence function), where the concretization function $\gamma_{\mathcal{V}}$ between $\wp\,(\mathbb{V} \times \mathbb{V})$ and $\wp\,(\wp\,(\mathbb{V})))$ is:

$$\gamma_{\mathcal{V}}(R) = \{\{v_i\}_{i \in \Delta} \mid \exists (i, j) \in \Delta^2, (v_i, v_j) \in R\}.$$

Hence, the domain of abstract environments is $(\mathbb{V} \to \mathbb{I}_{\text{int}} \times T) \times \wp\,(\mathbb{V} \times \mathbb{V})$. Due to this relational abstraction, we do not have a best abstraction function α, but we still have the concretization function γ. Though we cannot use the Galois connection framework, we can use the frameworks developed in [5].

3.4 Abstract domain

With the abstract domain constructed above, we can use the non-relational abstraction developed for powerset of unions:

$$\mathit{lco}\,(\wp\,(\text{Lab} \times (\mathbb{V} \to \mathbb{I}_\Omega))) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \text{Lab} \to \mathit{lco}\,(\wp\,(\mathbb{V} \to \mathbb{I}_\Omega))$$

This abstraction forgets information on conditionals and loops. This may be a problem, for example, with the program:

```
x:= ? in [0,1] ;
if (x=0) then x:=0 else x:=1 ;
```

Before the test, there is only one non-empty fixpoint for x ($\{0, 1\}$), but after the test, we get two fixpoints ($\{0\}, \{1\}$). Hence we lose information. In practice, we think that it is possible to deal with this issue without modifying the abstract domain, by a good implementation of the abstract test during the backward analysis. Here, it means that we should be able to see that both branches of the test are taken. Hence, if one branch can not satisfy the specification, the backward analysis must propagate that the specification is not satisfied.

[4] It appears to be difficult to infer the dependencies with this level of precision.

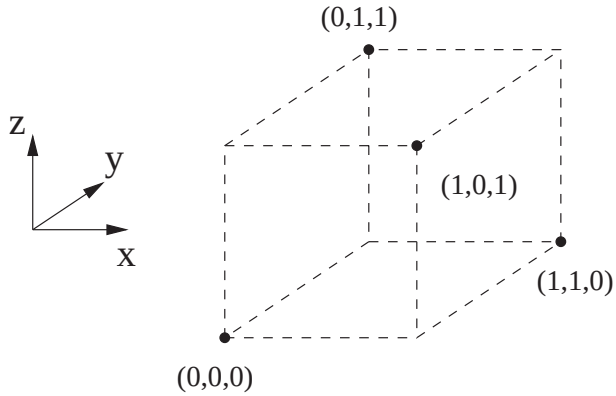

Fig. 3. An example of a set of 3-dimensional points which gives Cartesian products when projected in every direction, but is not a Cartesian product.

```
(1)                 Initial states: I
(2)  x:= ? in {0,1}
(3)  y:= ? in {0,1}
(4)  x:= x+y        Final states: F :  x = 1
```

1	{ x:[[ini: (-oo,+oo,-oo,+oo)]]; y:[[ini: (-oo,+oo,-oo,+oo)],x] }	1	{ BOT }
2	{ x:[[ini: (0,1,0,1)]]; y:[[ini: (-oo,+oo,-oo,+oo)],x] }	2	{ BOT }
3	{ x:[[ini: (0,1,0,1)]]; y:[[ini: (0,0,1,1)]] }	3	{ BOT }
4	{ x:[[ini: (0,1,1,2)]]; y:[[ini: (0,0,1,1)],x] }	4	{ x:[[ini: (1,1,1,1)]]; y:[[ini: (0,0,1,1)],x] }

Result after a forward analysis.

Result after a forward analysis followed by a backward analysis.

Fig. 4. Example described in section 4.1: program and results.

4 Examples of analyses

We wrote a prototype analyzer in OCaml with a graphical interface, from Cousot's Marktoberdorf generic analyzer [3]. The abstract domain for $\wp(\Sigma)$ is the domain of intervals, and the abstract domain for lower closures is the domain defined in the previous section. As a simple prototype, we did not try to optimize the abstract operations. Thus, its complexity is exponential in the depth of the nested loops, cubic in the number of variables and linear in the size of the program.

4.1 First example

We analyse the very small program presented, with its results, in Fig. 4.

The first random generator (for `x`) is "internal", whereas the second (for `y`) is "external". The initial states are the non-error states of program point (1),

and the final states are the states of program point (4) satisfying $x = 1$. The specification is that it is not possible to reach the final state when only the value of `x` is chosen[5]. Using the notations of the formula (1), it means that $A = F$, C are the states at program point (2), and B are the other states.

For each program point and each variable, we give:

1. The error status (`ini`, `err` or `top`), corresponding to the values of T.
2. Four integers (mm, Mm, mM, MM) describing the possible values of the variables. Informally, it means that for each generator, the variable has a value in $[mm, Mm]$ and in $[mM, MM]$. We changed the order of the integers (compared with section 3.1) so that this informal definition is more readable.
3. The list of variables which are dependent of this variable. Since the relation of dependence is symmetric, we give it only once.

For example, in program point (3), with only the forward analysis, we get `x:[ ini: (0,1,0,1) ]; y:[ ini: (0,0,1,1) ]`, which means informally that `x` is in $\{0, 1\}$, `y` is in $\{0, 1\}$ and takes all values in $\{0, 1\}$ whatever the value of `x` (as they are independent). If the computations were exact, the generators of the concrete lower closure operators would be $\{\{(x : 0, y : 0), (x : 0, y : 1)\}, \{(x : 1, y : 0), (x : 1, y : 1)\}\}$, which are exactly the concretization of the abstract environment. Here we do not lose information.

In program point (4), the result means that `x` is in $[0, 2]$ and takes at least one value in $[0, 1]$ and one value in $[1, 2]$, `y` takes all the values of $[0, 1]$, and the two variables are dependent. Note that the real generators would be $\{\{(x : 0, y : 0), (x : 1, y : 1)\}, \{(x : 1, y : 0), (x : 2, y : 1)\}$. We lose much information, but thanks to the following backward analysis this is not a problem.

With a backward analysis following the forward analysis, we get $x = 1$ in program point (4), and `y` takes all the values in $[0, 1]$, which gives the elements $\{(x : 1, y : 0), (x : 1, y : 1)\}$, Hence, in program point (3), it yields $\{(x : 1, y : 0), (x : 0, y : 1)\}$ which is not possible given the fact that `x` and `y` must be independent. Thus, the application of the lower closure operator gives `BOT` (which represents $\perp$), which proves our specification.

4.2 Second example

The program we analyse is described in Fig. 5. Here again we have two random operations. The first one (for the test) is internal, whereas the second (for n) is external. We want to be sure that by controlling the test, we can prevent $x = 0$ after the loop. Hence, with the notation of the equation 1, $A = F$, C are the states at program point (2), and B are the other states. The result of the analysis is given Fig. 5. Since we get $\perp$ for the initial point after the backward analysis, the analyzer proved our specification.

The difficult point in this analysis is the backward computation at program point (2). The analyzer detects that both branches of the analyzer are taken,

[5] This example was given in [8] as a verification which does not work with interval analysis.

```
(0)                 Initial states: F :  x = 1
(1)
(2)     while (n>0) do {
(3)           if (? in [0,1]=0) then
(4)                 x = x * (n-1);
(5)           else
(6)                 x = x * n;
(7)            fi
(8)      n = n - (? in [0,1]);
(9)      }
                    Final states: F :  x = 0
```

0 { n:[{ ini: (-oo,+oo,-oo,+oo) }]; x:[{ ini: (1,1,1,1) },n] }	0 { BOT }
1 { n:[{ ini: (-oo,+oo,-oo,+oo) }]; x:[{ ini: (0,+oo,0,+oo) },n] }	1 { n:[{ ini: (-oo,+oo,-oo,+oo) }]; x:[{ ini: (0,0,0,0) },n] }
2 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,+oo,0,+oo) },n] }	2 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,0,0,0) },n] }
3 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,+oo,0,+oo) },n] }	3 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,+oo,0,+oo) },n] }
4 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,+oo,0,+oo) },n] }	4 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,0,0,0) },n] }
5 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,+oo,0,+oo) },n] }	5 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,0,0,0) },n] }
6 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,+oo,0,+oo) },n] }	6 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,0,0,0) },n] }
7 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,+oo,0,+oo) },n] }	7 { n:[{ ini: (1,+oo,1,+oo) }]; x:[{ ini: (0,0,0,0) },n] }
8 { n:[{ ini: (0,+oo,0,+oo) }]; x:[{ ini: (0,+oo,0,+oo) },n] }	8 { n:[{ ini: (0,+oo,0,+oo) }]; x:[{ ini: (0,0,0,0) },n] }
9 { n:[{ ini: (-oo,0,-oo,0) }]; x:[{ ini: (0,+oo,0,+oo) },n] }	9 { n:[{ ini: (-oo,0,-oo,0) }]; x:[{ ini: (0,0,0,0) },n] }
Result after a forward analysis.	Result after a forward analysis followed by a backward analysis.

Fig. 5. Example of section 4.2: program and results.

independently of the variables `x` and `n`, using the weak relational analysis. Thus, it computes an intersection of the environments of program points (3) and (5).

5 Discussion

In this section we examine the improvement obtained by property-checking driven analyses, with respect to the precision of the analysis.

Starting from the definition of $\mathcal{S}$ (as in equation (1)), the first idea to overapproximate $\mathcal{S}$ with abstract interpretation-based analyses is to choose an abstract domain, develop in this abstract domain approximations for pre and $\widetilde{pre}$, and directly approximate the fixpoint using the fixpoint transfer theorem. A more precise method, still easier than property-checking driven analysis, is to combine the previous analysis with a reachability analysis starting from I, a method proposed in [7]. This method gives a backward-forward analysis quite similar to ours, but which does not use the lower closure framework. Thus, comparing the two analyses is a good method to see what is gained with this framework.

The first thing we can see is that when the specification does not use $\widetilde{pre}$ (e.g. we just want to prove that some states are unreachable), our technique is useless. Then the analysis is at worst as imprecise as a classical interval analysis (though the complexity is worse).

To show that our analysis can be useful, let us examine a very small example:

$I : y \in]-\infty, +\infty[$

```
x:= ? in [0,2];
y:= f(x,y) ;
```

$A : y \in [0, 8]$

Here, `f(x,y)` should be read as an arbitrary expression depending on x and y, e.g. $x+y$ or $x*y$. We want y to be in $[0, 8]$ at the end of the program, whatever the result of the non-deterministic operator.

A "classical" analyzer will know that x is in $[0, 2]$ when $f(x, y)$ is computed. In the backward analysis, it must be able to carry a relation between x and y, sufficient to deduce the correct constraints on y when it analyses the command `x:= ? in [0,2]`. The relation the analyzer can carry is closely related to the abstract domain, and may not be suitable for the function f.

On the other hand, our analyzer can deduce precise constraints on y directly during the backward analysis of $y := f(x, y)$, knowing that x must take the values 0 and 2. Hence it does not need to transmit complex relations.

Let us consider this example when $f(x, y) = x * y$. Even with an expensive abstract domain like polyhedra, we cannot express the good relation between x and y, and the result of the analysis remain imprecise (i.e. it finds that y is in $[-\infty, +\infty]$ at the beginning of the program).Whereas our analyzer finds that y must take values in $[0, 4]$, which is the optimal solution. This example shows that our approach can give better results than the traditional approach, even with expensive abstract domains.

6 Conclusion and future work

We presented in this paper the construction of an abstract domain used in a small prototype analyzer for automatic program verification of temporal properties. This analyzer uses a technique based on lower closure operators to exploit the information given by the specification in the analysis. We showed how to abstract the domain of lower closure operators, from the classical interval abstraction, to keep relational information about the states of the program. Even with these weak abstractions, the method may give more precise results than classical abstract analyses with expensive relational domains. Though we analyse only a small class of specifications, the same method can be used for larger classes by including temporal formulas in the concrete domain of the semantics (as was done in [8]).

Our analyzer uses similar abstractions for both analyses (the direct one, on sets of states, and the "reverse" one, on lower closures). Future work can stem from the idea that these abstractions are, in fact, orthogonal. Thus, we can use our abstractions on lower closures along with polyhedra on sets of states, improving the precision of the analysis (without designing a complex abstraction on lower closures based on polyhedra). In general, we believe that property-checking driven analysis can be applied to other abstraction-related analyses. Especially, it would be interesting to examine exactly how it can be compared

with abstraction refinements and domain completions [6] and how we can use both technique efficiently.

Acknowledgments

This work was partly done while the author was a visitor in the Cambridge University Computer Laboratory. I wish to thank the anonymous referees, as well as Alan Mycroft and Radhia Cousot for their comments and suggestions.

References

1. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer-Verlag, October 2002.
2. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 46–55, 1993.
3. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999. Generic Abstract Interpreter available on http://www.di.ens.fr/~cousot/Marktoberdorf98.shtml.
4. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
5. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
6. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.
7. D. Massé. Combining backward and forward analyses of temporal properties. In O. Danvy and A. Filinski, editors, *Proceedings of the Second Symposium PADO'2001, Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Sciences*, pages 155–172, Århus, Denmark, 21 – 23 May 2001. Springer-Verlag, Berlin, Germany.
8. D. Massé. Semantics for abstract interpretation-based static analyzes of temporal properties. In M. Hermenegildo, editor, *Proceedings of the Ninth Static Analysis Symposium SAS'02*, volume 2477 of *Lecture Notes in Computer Sciences*, pages 428 – 443, Madrid, Spain, 17 – 20 September 2002. Springer-Verlag, Berlin, Germany.
9. D. Massé. Property checking driven abstract interpretation-based static analysis. In *Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Sciences*, pages 56–69, New York, USA, January 9–11 2003. Springer-Verlag, Berlin, Germany. Available on http://www.stix.polytechnique.fr/~dmasse/papiers/VMCAI2003.pdf.
10. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf.