ANTIQUE TEAM

Internship report

June 1 – July 24, 2015

Thread-Modular Analysis
designing relational abstractions of interferences

Author:
Raphaël Monat
ENS Lyon
raphael.monat@ens-lyon.org

Supervisor:
Antoine Miné
Junior researcher, CNRS
Attached part-time lecturer, ENS
mine@di.ens.fr

Abstract

In this document, we use the Abstract Interpretation framework to analyze concurrent programs using Thread-Modular Analysis. We designed a relational abstraction of interferences in order to infer more properties and go beyond the state of the art. We implemented a basic analyzer, studying the numerical properties of a simple language. We present the results obtained, as well as a study of the scalability of this approach.

1 Introduction

Simple software bugs can have dire consequences on critical systems. The crash of the Ariane V rocket[1] is a well-documented and classical example of the consequences of an integer overflow. More recently, the same kind of integer overflow was discovered on the Boeing 787[2], and could have cut electrical power.

To discover these bugs, the most widely used technique is testing. The problem is that covering all possible executions using testing is difficult and costly. On the contrary, Static Analysis covers every possible executions and finds every bug. Program analysis is still a really difficult problem as theoretical results such as the undecidability of the halting problem prevent automation. This way, Static Analysis can be used to guarantee that a software is bug-free.

There are different theories in Static Analysis, including Abstract Interpretation and Model Checking. We will here focus on Abstract Interpretation. It is a well-developped theory, and professional analyzers of sequential programs such as ASTRÉE have been successfully commercialized. However, the analysis of concurrent programs is a much more difficult task: programs can communicate implicitely through a shared memory by their execution, and there are a lot of possible orders of execution resulting in a combinatorial explosion, so that testing is not a real option anymore. This complexity also depends on the memory model provided.

One of the methods to analyze concurrent programs, is to sequentialize the program and analyze it as usual, though there is a combinatorial explosion in the number of cases. Another method is to analyze each thread almost separately. It is called a Thread-Modular Analysis. Developping a relational Thread-Modular Analysis is more difficult than creating a relational analysis of a sequential program. That is why the only analysis conducted before was range analysis, so that for each variable we only knew about an interval it was guaranteed to be in. But for precision concerns, keeping linear relations between variables is also interesting, though more costly. We developped an analysis that is capable of keeping relations between variables, and then tested its precision and scalability.

Our contribution is the definition of a Thread-Modular Analysis using a concrete denotational semantics, being then abstracted using relational numerical domains, and an implementation of a prototype called BATMAN, before comparing it with CONCURINTERPROC. CONCURINTERPROC is another academic prototype analyzing linear relations in concurrent programs which is not thread modular. We validated experimentally the efficiency of our approach: the experimental complexity of BATMAN depends on the sum of the lengths of the threads, whereas other methods typically depends on the product of the lengths of the threads. Our approach is still as precise as other methods.

In this document, we first introduce the reader to Abstract Interpretation, before presenting the state of the research in the analysis of concurrent programs. We then define the execution model we worked on, before giving semantics of our program and of our analysis, and then abstracting it. We present the prototype-analyzer I developped, BATMAN, and the results of our experiments.

This document is a report of the internship I did from June 1 to July 24, 2015, under the supervision of Antoine Miné, in the ANTIQUE team, at the ENS Ulm. In this internship, I started by gathering informatio on Abstract Interpretation, the state of the art of Thread-Modular Analysis, and CONCURINTERPROC. We then decided to try a relational abstraction of the interferences, and we tried to formalize it. Then I started to learn how to use Apron, which is a library handling numerical domains (such as intervals, octagons and polyhedra). I implemented an analyzer prototype to test our new analysis and compare it experimentally with CONCURINTERPROC. I had to take some time in order to understand some concepts and libraries, but my main approach was the one described above.

This internship was a part of my curriculum at the ENS Lyon[3], in order to obtain a *Licence* (that is, a diploma equivalent to a Bachelor) in Computer Science.

2 Introduction to Static Analysis by Abstract Interpretation

The goal of Abstract interpretation is to speed up computations, by using approximations. It was first developed by Patrick Cousot and Radhia Cousot, and presented in [CC77]. Here, we will present the case of static analysis.

[1] http://www.around.com/ariane.html
[2] In the discussion section of this document: http://www.gpo.gov/fdsys/pkg/FR-2015-05-01/pdf/2015-10066.pdf
[3] http://www.ens-lyon.fr/DI/?lang=en

2.1 Definitions

We will start by giving a definition of a lattice, a mathematical object that will be widely used in the following sections.

Lattice $(X, \preccurlyeq, \sqcup, \sqcap)$ is a lattice, if:

- $(X, \preccurlyeq)$ is a partially ordered set
- $\forall (a,b) \in X^2$, there is a least upper bound $a \sqcup b$ $(\in X)$
- $\forall (a,b) \in X^2$, there is a greatest lower bound $a \sqcap b$ $(\in X)$

For example, $(\mathbb{Z}, \leq, \max, \min)$ is a lattice.
$(X, \preccurlyeq, \sqcup, \sqcap, \bot, \top)$ is a complete lattice if:

- $(X, \preccurlyeq)$ is a partially ordered set
- $\forall S \subseteq X$, there is a least upper bound $\sqcup S$ $(\in X)$
- $\forall S \subseteq X$, there is a gratest lower bound $\sqcap S$ $(\in X)$
- $\bot = \sqcap X$
- $\top = \sqcup X$

The previous example is not a complete lattice $(\top \notin \mathbb{Z})$.

We will abbreviate “partially ordered set” into “poset”.

We also define for a function $f : A \to B$, $x \in A, y \in B$ the following notation:

$$\forall u \in A, f[x \mapsto y](u) = \begin{cases} f(u) & \text{if } u \neq x \\ y & \text{if } u = x \end{cases}$$

2.2 Galois connections

The goal of Abstract Interpretation is to use well-chosen approximations in order to accelerate computations, while keeping soundness properties. By soundness, we mean that if there is any error in the program, we want our approximations to still find it. Conversely, if the analysis detects errors, these might just be false alarms (and this is where things are approximated). One way to formalize these approximations is to use Galois connections.

Galois connections Let $(C, \subseteq)$ and $(A, \sqsubseteq)$ be two posets. We say that (α, γ) is a Galois connection, if $\alpha : C \to A$ and $\gamma : A \to C$ respect the following property:

$$\forall c \in C, \forall a \in A, \alpha(c) \sqsubseteq a \iff c \subseteq \gamma(a)$$

We use the following notation: $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$. We say that C is the concrete domain, and A is the abstract domain. Similarly, γ is called a concretization (function), and α an abstraction.

Remark We only used posets in the definition, but usually lattices are used. It is common use to add $^{\#}$ to abstract domains.

Abstracting $\mathcal{P}(\mathbb{Z})$ We will now develop a common abstraction of sets of integers into intervals. With this abstraction, we will be able to analyze programs to detect divisions by zero or overflows. Let $I = \{(a,b) \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\} \cup \bot$. We now have to define a lattice:

- $(a,b) \sqsubseteq (c,d) \iff c \leq a \land b \leq d$
- $(a,b) \sqcup (c,d) = (\min(a,c), \max(b,d))$
- $(a,b) \sqcup \bot = \bot \sqcup (a,b) = (a,b)$

- $(a,b) \sqcap (c,d) = \begin{cases} (\max(a,c), \min(b,d)) \text{ if } \max(a,c) \leq \min(b,d) \\ \perp \text{ otherwise} \end{cases}$

- $(a,b) \sqcap \perp = \perp \sqcap (a,b) = \perp$

- $\top = (-\infty, +\infty)$

Moreover, this lattice is complete.

We have the following Galois connection: $(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (I, \sqsubseteq)$, where:

$$\alpha(X) = \begin{cases} \perp \text{ if } X = \emptyset \\ (\min X, \max X) \text{ otherwise} \end{cases}$$

$$\begin{cases} \gamma(\perp) = \emptyset \\ \gamma((a,b)) = \{x \in \mathbb{Z} \mid a \leq x \leq b\} \end{cases}$$

Now we have a sound abstraction of $\mathcal{P}(\mathbb{Z})$, it would be convenient to develop sound abstractions of functions, such as $+$. We will say that $f^\#$ is a sound abstraction of $f : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$, if: $\forall d^\# \in I, f(\gamma(d^\#)) \subseteq \gamma(f^\#(d^\#))$. This is equivalent to $\alpha(f(\gamma(d^\#))) \sqsubseteq f^\#(d^\#)$. It really states that $f^\#$ has to be an over-approximation of f, so that we do not lose any case, even if the precision is reduced in some cases. This way, we cover every possible execution, and we are able to detect every error.

For example, we can define $-^\#$ such that: $(a,b) -^\# (c,d) = (a-d, b-c)$. We can notice that $-^\#$ is exact, ie $f(\gamma(d_1^\#), \gamma(d_2^\#)) = \gamma(f^\#(d_1^\#, d_2^\#))$. On the contrary, $\cup^\# = \sqcup$ is not exact, because $\gamma((0,1) \cup^\# (3,4)) = \gamma((0,4)) = [0;4]$, wheras $\gamma((0,1)) \cup \gamma((3,4)) = \{0,1,3,4\}$. Even if $\cup^\#$ is not exact, we still have that $\{0,1,3,4\} \subseteq [0;4]$, which states that we have an over-approximation of $\cup$.

Point-wise lifting We just need to focus on the extension of the previously-mentionned abstraction of $\mathcal{P}(\mathbb{Z})$ to an abstraction of $\mathcal{V} \to \mathcal{P}(\mathbb{Z})$. This last set can be seen as a memory domain, where we map each variable a set of integers. We need to extend the operations explained above to $\mathcal{V} \to I$. This is more a notation issue than a real problem, and there are no theoretical difficulties. We need to extend our lattice on I in a "point-wise way", where the points are the elements of $\mathcal{V}$. We will add a dot over the operations to emphasize the use of functions (here from $\mathcal{V} \to I$) rather than sets (I). Let f, g be in $\mathcal{V} \to I$:

- $f \mathrel{\dot{\subseteq}} g \Longleftrightarrow \forall V \in \mathcal{V}, f(v) \subseteq g(v)$

- $f \mathbin{\dot{\sqcup}} g = \lambda V. f(v) \sqcup g(v)$

- $f \mathbin{\dot{\sqcap}} g = \lambda V. f(v) \sqcap g(v)$

This extended structure is still a lattice and is called a Cartesian lattice. We can then extend the Galois connection mentionned above into a Cartesian Galois connection:

$$(\mathcal{P}(\mathcal{V} \to \mathbb{Z}) \subseteq) \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} (\mathcal{V} \to I, \dot{\subseteq})$$

$$\dot{\alpha}(X) = \lambda V. \alpha(\{\rho(V) \mid \rho \in X\})$$

$$\dot{\gamma}(f) = \{\rho \mid \forall V \in \mathcal{V}, \rho(V) \in \gamma(f(V))\}$$

2.3 Application to static analysis

An application of this interval abstraction is to be able to do a range analysis, by bounding each variable at any point of a program. This can be useful, if we want to detect integer overflows and division by zero. We first define a simple language, supporting only boolean and integers:

A simple imperative language. Let $\mathcal{V}$ be the set of integer variables, $k \in \mathbb{Z}$. We can now define a basic language:

$\langle$*arithmetical expression*$\rangle$ ::= $[k_1; k_2]$ | k | X | $a_1 \dagger a_2$, $\dagger \in \{+, -, \times, /, \%\}$

$\langle$*boolean expression*$\rangle$ ::= $b_1 \bullet b_2$ | not b_1 | $a_1 \boxdot a_2$, $\bullet \in \{\vee, \wedge\}$, $\boxdot \in \{<, >, \leq, \geq, ==, \neq\}$

$\langle$*commands*$\rangle$ ::= c_1 ; c_2
| $X = a$
| if b then c_1 else c_2 endif
| while b do c done

Concrete denotational semantics. Now, we will define a denotational semantics of this language, to describe mathematically what every command of this language does. Denotational semantics uses functions computing a modification of the memory given another. There are other semantics, but we will not talk about them here. For a detailed study, one can read [Cou02] (there is hierarchy of semantics page 54).

This representation is interesting here because it is really similar to what one can implement in a real analyzer.

We define three operators: one to evaluate a given arithmetical or boolean expression using a memory domain, $\mathbb{E}[\![.]\!]$, another $\mathbb{F}[\![.]\!]$ to filter memory domains and keep only the ones satisfying a boolean expression, and $\mathbb{S}[\![.]\!]$, to compute the effects of commands on a set of memory domains.

$$\begin{aligned} \mathbb{E}[\![\textit{arithmetical or boolean expression}]\!] &: \quad (\mathcal{V} \to \mathbb{Z}) \longrightarrow \mathcal{P}(\mathbb{Z} \cup \{\text{true}, \text{false}\}) \\ \mathbb{F}[\![\textit{boolean expression}]\!] &: \quad \mathcal{P}(\mathcal{V} \to \mathbb{Z}) \longrightarrow \mathcal{P}(\mathcal{V} \to \mathbb{Z}) \\ \mathbb{S}[\![\text{commands}]\!] &: \quad \mathcal{P}(\mathcal{V} \to \mathbb{Z}) \longrightarrow \mathcal{P}(\mathcal{V} \to \mathbb{Z}) \end{aligned}$$

To compute the effect of an arithmetical expression such as $a_1 \dagger a_2$, we have to recursively evaluate the possible values of a_1 and a_2, and then evaluate all possible results. It is really similar for the evaluation of boolean expressions.

$$\begin{aligned} \mathbb{E}[\![k]\!](\rho) &= \{k\} \\ \mathbb{E}[\![[k_1; k_2]]\!](\rho) &= \{x \in \mathbb{Z} \mid k_1 \leq x \leq k_2\} \\ \mathbb{E}[\![X]\!](\rho) &= \{\rho(X)\} \\ \mathbb{E}[\![a_1 \dagger a_2]\!](\rho) &= \{v_1 \dagger v_2 \mid v_1 \in \mathbb{E}[\![a_1]\!](\rho), v_2 \in \mathbb{E}[\![a_2]\!](\rho), \dagger \in \{+, -, \times\} \vee (\dagger \in \{/, \%\} \wedge v_2 \neq 0)\} \\ \mathbb{E}[\![\text{not } b]\!](\rho) &= \{\neg v \mid v \in \mathbb{E}[\![b]\!](\rho)\} \\ \mathbb{E}[\![b_1 \bullet b_2]\!](\rho) &= \{v_1 \bullet v_2 \mid v_1 \in \mathbb{E}[\![b_1]\!](\rho), v_2 \in \mathbb{E}[\![b_2]\!](\rho), \bullet \in \{\vee, \wedge\}\} \\ \mathbb{E}[\![a_1 \boxdot a_2]\!](\rho) &= \{v_1 \boxdot v_2 \mid v_1 \in \mathbb{E}[\![a_1]\!](\rho), v_2 \in \mathbb{E}[\![a_2]\!](\rho), \boxdot \in \{<, >, \leq, \geq, ==, \neq\}\} \end{aligned}$$

The case of the assignment is simple, we just have to return the new set of memories and take into account all the possible assignments. To compute the effect of a sequence of statements, we just need to compose these statements.

$$\begin{aligned} \mathbb{S}[\![X = a]\!](P) &= \{\rho[X \mapsto v] \mid \rho \in P, v \in \mathbb{E}[\![a]\!](\rho)\} \\ \mathbb{S}[\![\text{stat}_1 ; \text{stat}_2]\!] &= \mathbb{S}[\![\text{stat}_2]\!] \circ \mathbb{S}[\![\text{stat}_1]\!] \end{aligned}$$

To express the semantics of the if and the while statements, we will need to filter the memory domains to apply modifications to the only domains satisfying the good condition. The case of the if statement is quite simple: an analysis is conducted in each part of the statement, after having filtered each part, and the results are merged afterwards. The while statement just needs the computation of a loop invariant via a fixpoint.

$$\begin{aligned}
\mathbb{F}[\![b]\!](P) &= \{\rho \mid \rho \in P,\ \text{true} \in \mathbb{E}[\![b]\!](\rho)\}\\
\mathbb{S}[\![\text{if } b \text{ then } t \text{ else } f]\!](P) &= \\
\text{let } T &= \mathbb{S}[\![t]\!] \circ \mathbb{F}[\![b]\!]P \text{ in}\\
\text{let } F &= \mathbb{S}[\![f]\!] \circ \mathbb{F}[\![\neg b]\!]P \text{ in}\\
&T \cup F\\
\mathbb{S}[\![\text{while } b \text{ do } c]\!](P) &= \mathbb{F}[\![\neg b]\!] \circ \text{lfp}\, \lambda Y.(P \cup \mathbb{S}[\![c]\!] \circ \mathbb{F}[\![b]\!]Y)
\end{aligned}$$

Abstract denotational semantics. The abstract semantics is really similar to the concrete one, with only small modifications: we first abstract $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$ to $\mathcal{V} \to \mathcal{P}(\mathbb{Z})$, and then into $\mathcal{V} \to I$, using the Galois connection presented in section 2.2. This way, we do not need to manipulates memory sets anymore.

$$\begin{aligned}
\mathbb{E}^{\#}[\![\textit{arithmetical or boolean expression}]\!] &: \quad (\mathcal{V} \to I) \longrightarrow I\\
\mathbb{F}^{\#}[\![\textit{boolean expression}]\!] &: (\mathcal{V} \to I) \longrightarrow (\mathcal{V} \to I)\\
\mathbb{S}^{\#}[\![\text{commands}]\!] &: (\mathcal{V} \to I) \longrightarrow (\mathcal{V} \to I)
\end{aligned}$$

$$\begin{aligned}
\mathbb{E}^{\#}[\![k]\!](\rho^{\#}) &= (k, k)\\
\mathbb{E}^{\#}[\![[k_1; k_2]]\!](\rho^{\#}) &= (k_1, k_2)\\
\mathbb{E}^{\#}[\![X]\!](\rho^{\#}) &= \rho^{\#}(X)\\
\mathbb{E}^{\#}[\![a_1 \dagger a_2]\!](\rho^{\#}) &= \mathbb{E}^{\#}[\![a_1]\!] \dagger^{\#} \mathbb{E}^{\#}[\![a_2]\!]\\
\mathbb{S}^{\#}[\![X = a]\!](\rho^{\#}) &= \rho^{\#}[X \mapsto \mathbb{E}^{\#}[\![e]\!]]\\
\mathbb{S}^{\#}[\![\text{stat}_1 \ ; \text{stat}_2]\!] &= \mathbb{S}^{\#}[\![\text{stat}_2]\!] \circ \mathbb{S}^{\#}[\![\text{stat}_1]\!]
\end{aligned}$$

The case of filtering is a bit more difficult. By transforming the cases, we only have to define the abstract condition filtering for $\vee$, $\wedge$ and $\leq$, as we can distribute the *not* in boolean expressions, or change the comparison operator: we are using integers, so that $x < y \Leftrightarrow x \leq y - 1$. To filter a domain with an expression such as $X \leq c$, we will have to "cut" the assignment of X to respect the condition. For example, if we know that $a \leq X \leq b$, we have two cases: if $a > c$, then there is no domain satisfying the condition; if $a \leq c$, and $c < b$, we have to cut the domain, so that $a \leq X \leq c$. Otherwise, we do not have to change the interval. The general case is more complicated and presented in [Min04], section 2.4, "Abstract Operators and Transfer Functions on $\mathcal{D}^{\#}$".

We suppose that $\rho^{\#}(X) = (a, b)$. We now present the semantics of the if statement:

$$\begin{aligned}
\mathbb{F}^{\#}[\![X \leq c]\!](\rho^{\#}) &= \begin{cases} \rho^{\#}[X \mapsto (a, \min(b, c)) \text{ if } a \leq c \\ \dot{\perp} \text{ if } a > c \end{cases}\\
\mathbb{F}^{\#}[\![b_1 \vee b_2]\!](\rho^{\#}) &= \mathbb{F}^{\#}[\![b_1]\!](\rho^{\#}) \ \dot{\cup}^{\#} \ \mathbb{F}^{\#}[\![b_2]\!](\rho^{\#})\\
\mathbb{F}^{\#}[\![b_1 \wedge b_2]\!](\rho^{\#}) &= \mathbb{F}^{\#}[\![b_1]\!](\rho^{\#}) \ \dot{\cap}^{\#} \ \mathbb{F}^{\#}[\![b_2]\!](\rho^{\#})\\
\mathbb{S}^{\#}[\![\text{if } b \text{ then } t \text{ else } f]\!]\rho^{\#} &= \\
\text{let } T &= \mathbb{S}^{\#}[\![t]\!] \circ \mathbb{F}^{\#}[\![b]\!]\rho^{\#} \text{ in}\\
\text{let } F &= \mathbb{S}^{\#}[\![f]\!] \circ \mathbb{F}^{\#}[\![\neg b]\!]\rho^{\#} \text{ in}\\
&T \ \dot{\cup}^{\#} F
\end{aligned}$$

The case of loops. We left aside the case of loops, because we need to introduce a new concept called widening before. The problem of the lfp operator presented in the concrete semantics is that it might require infinite calculations. We deal with this using a widening operator, less precise, but more efficient.

We say that $\nabla : (I \times I) \to I$ is a widening operator, if:

– $\forall (x, y) \in I^2$, $\gamma(x) \cup \gamma(y) \subseteq \gamma(x \nabla y)$

- for a sequence $(x_n) \in I^{\mathbb{N}}$, the following sequence $\begin{cases} y_0 = x_0 \\ y_{n+1} = y_n \triangledown x_{n+1} \end{cases}$ stabilizes in finite time: $\exists N \in \mathbb{N}, \forall n \geq N, y_n = y_N$

This definition can easily be extended to any abstract domain.
For example, $i \triangledown j = (-\infty, +\infty)$ is a widening operator, although not an interesting one. To be more precise while ensuring convergence, we can decide to change unstable bounds into infinite ones, to obtain the following widening: $\forall ((a,b),(c,d)) \in I^2$, $\bot \triangledown (a,b) = (a,b) \triangledown \bot = (a,b)$ and:

$$(a,b) \triangledown (c,d) = \left(\begin{cases} a \text{ if } a \leq c \\ -\infty \text{ if } a > c \end{cases}, \begin{cases} b \text{ if } b \geq d \\ +\infty \text{ if } b < d \end{cases} \right)$$

We can then define the abstract semantics for the while loop, as the widening operator ensures the convergence in finite time of the limit below:

$$\mathbb{S}^\# [\![\text{while } b \text{ do } c]\!] \rho^\# = \mathbb{F}^\# [\![\neg b]\!] (\lim \lambda Y. Y \dot{\triangledown} (\rho^\# \dot{\cup}^\# \mathbb{S}^\# [\![c]\!] \circ \mathbb{F}^\# [\![b]\!] Y))$$

Increasing & decreasing iterations Consider the following program:

```
i = 0;
while (i < 10) do
  i = i + 1;
done;
```

Using the concrete semantics, we would find that $i = 10$ at the end of the program. Let's apply the abstract semantics:

- $y_0(i) = (0,0)$
- $y_1(i) = y_0(i) \triangledown (0,1) = (0,0) \triangledown (0,1) = (0,+\infty)$

After the filtering, we obtain only an over-approximation of the result, which is $i \in [10;+\infty]$, as $\mathbb{F}^\# [\![\neg(i < 10)]\!] [i \mapsto (0,+\infty)] = [i \mapsto (10,+\infty)]$. In some cases, we could refine this result using a decreasing iteration: after the widening, we can continue iterating, but without the widening.

$$\begin{aligned} &\mathbb{F}^\# [\![i < 10]\!] (y_1) \dot{\cup}^\# \mathbb{S}^\# [\![i = i + 1]\!] \circ \mathbb{F}^\# [\![i < 10]\!] (y_1) \\ &= [i \mapsto (0,9) \cup^\# (1,10)] \\ &= [i \mapsto (0,10)] \\ &\mathbb{S}^\# [\![\text{while } (i < 10) \text{ do } i = i+1; \text{ done}]\!] \\ &= \mathbb{F}^\# [\![\neg i < 10]\!] [i \mapsto (0,10)] \\ &= [i \mapsto (10,10)] \end{aligned}$$

This way, we can gain some precision at an interesting cost. Sometimes, delaying the widening, replacing with an abstract join for the first few iterations, can improve the precision. It is also possible to unroll the loop, which means analyzing the first few iterations. This is not the same as delaying the widening, because these iterations are not joined abstractly. Generally, both techniques are used.

Widening with thresholds In some cases, the decreasing iterations are not precise enough. For example, we cannot infer that $i = 10$ at the end of the following program using what is described above:

```
i = 0;
while (true) do
  if (i < 10) then
    i = i + 1;
  endif;
done;
```

The reason why it fails is a bit too complex to explain it in this report. This phenomenon is explained in [HH12].

We can define the widening with thresholds on T, where T is a finite set of integers, containing also $+\infty$ and $-\infty$:

$$\forall((a,b),(c,d)) \in I^2, (a,b)\nabla(c,d) = \left(\begin{cases} a \text{ if } a \leq c \\ \max\{t \in T \mid t \leq c\} \text{ if } a > c \end{cases}, \begin{cases} b \text{ if } b \geq d \\ \min\{t \in T \mid t \geq d\} \end{cases} \right)$$

If $T = \{-\infty, 10, +\infty\}$, we can infer that $i = 10$. If $T = \{-\infty, a, +\infty\}$, with $a \geq 10$, we will only find that $i \in [10; a]$.

Soundness of the abstraction To prove the soundness of this abstraction, we have to prove that $\mathbb{S}[\![stat]\!](\dot\gamma(i)) \subseteq \dot\gamma(\mathbb{S}^\#[\![stat]\!](i))$. This can be done by induction on the program commands.

Detecting errors We have not presented how to catch errors such as division by zero using the analysis presented above. Let Ω be the set of possible errors. We can extend $\mathbb{S}^\#[\![.]\!]$, $\mathbb{F}^\#[\![.]\!]$ to $\mathcal{P}(\mathcal{V} \to \mathbb{Z}) \times \mathcal{P}(\Omega) \longrightarrow \mathcal{P}(\mathcal{V} \to \mathbb{Z}) \times \mathcal{P}(\Omega)$ and $\mathbb{E}^\#[\![.]\!]$ to $(\mathcal{V} \to \mathbb{Z}) \longrightarrow \mathcal{P}(\mathbb{Z} \cup \{\text{true, false}\}) \times \mathcal{P}(\Omega)$. This way, we can detect errors during the evaluation of arithmetical expressions (for example, a division by zero), and propagate them throughout the analysis, so that the semantics $\mathbb{S}^\#[\![.]\!]$ outputs the set of possible errors encountered during the execution of the program.

2.4 Relational domains

Relational domains are able to keep relations beetween variables (usually, linear relations).

The need for relational domains.

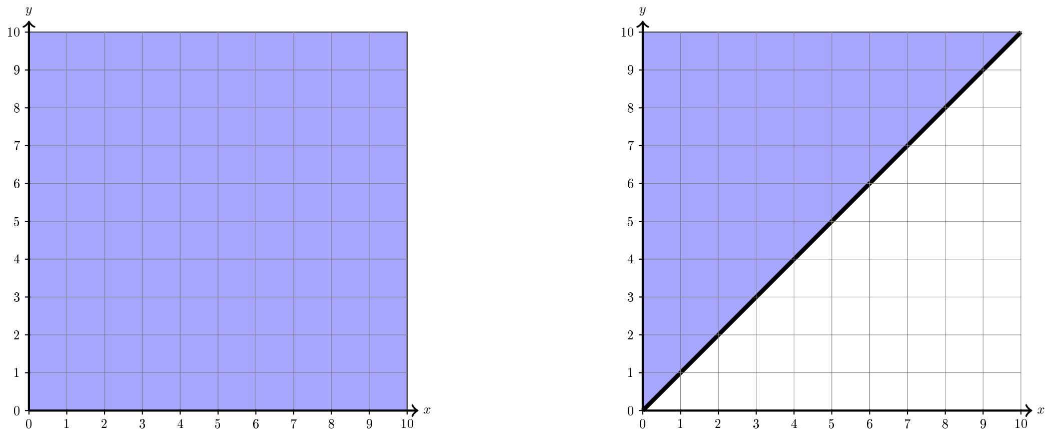
```
assume 0 <= x <= 10;
assume 0 <= y <= 10;

if (x > y) then
  z = x;
else
  z = y;

assert (z - y) >= 0;
```

Using an interval analysis, we would get that $z, x, y \in [0; 10]$ so that $z - y \in [-10; 10]$. But $z \geq y$ holds during the whole program. It might be interesting to have domains that are able to store that kind of relation. We will call them relational domains.

Polyhedra. We will present an example using the polyhedra domain, introduced by Patrick Cousot and Nicolas Halbwachs, in [CH78]. The numerical domain called polyhedra is represented by convex polyhedra, that might be unbounded. However, operations on polyhedra can be costly to compute, with some operations having an exponential complexity in practice (in the number of variables). We will present how this relational domain works. Consider the following code:

```
1 assume 0 <= x <= 10;
2 assume 0 <= y <= 10;
3 if (x > y) then
4   y = x;
5 endif
```

At the end of line 2, the domain is the square represented in figure 1a. During the analysis of the if statement, the domain is separated into one corresponding to the assignement $x = y$ if $x > y$ (the dark thick line), and the other corresponding to the empty else (the blue triangle). We then join the domains to obtain the figure 1b. This way, we know that the variables at the end of the program are contained in the blue triangle, that is we have: $x \geq 0, y \leq 10, x \leq y$.



(a) At the end of line 2 (b) At the end of line 5

Figure 1: Polyhedra corresponding to the example above

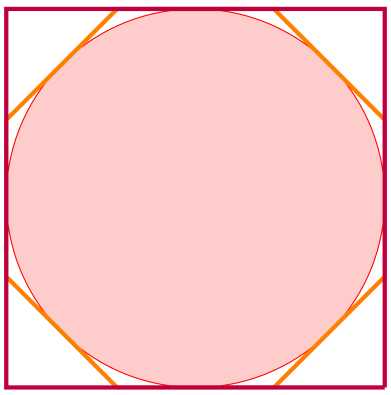


Figure 2: Best abstractions of a circle using boxes and octagons.
There is not best abstraction of a circle in the polyhedra domain

We assume we have some built-in *assign* function, to assign an arithmetic expression to a variable over a domain, and functions called *add*, *rename* and *delete* to be able to manage variables over a domain. We do not delve into the details; the reader can consult [CH78] for more informations.

The polyhedra domain has only a concretization function γ, but no best abstraction α: it is not possible to have a best abstraction of a circle (cf figure 2: even if we use only regular polygons, we can always increase the number of sides to improve the abstraction of the circle). Having only a concretization function but no Galois connection leads to a slightly weaker version of abstract interpretation (developped in [CC92]), but it is not a problem in practice. In particular, it does not affect the soundness of the analysis, which is of paramount importance.

Weakly-relational domains. Some weakly-relational domains exist. The relationships we are able to infer are a subset of general linear constraints, but the time and space complexities are usually better. For example, octagons ([Min06]) can infer properties $\pm X \pm Y \leq c$, where X and Y are variables and c is a constant. It has a worst case complexity of $\mathcal{O}(n^3)$, where n is the number of variables ([Min04], page 152).

2.5 The Apron library

Apron is a library handling different numerical domains, among which octagons and polyhedra. It provides a global interface so that the use of a numerical domain is independant from the one chosen. It is written in C, and released under an LGPL license. Ocaml bindings are available.

This library is presented in [JM09]. It is available online[4] and it can also be installed via opam, the Ocaml Package Manager.

[4]http://apron.cri.ensmp.fr/library/

A fork of the Apron library is called Bddapron ([Jea]). It handles, in addition to numerical domains, finite sets using Binary Decision Diagrams.

3 Related work

A lot of related work has already been presented in the last section. We refer the reader to [Rin01] for a survey of the analysing concurrent programs. We present Thread-Modular Analysis, on which our approach is based, and CONCURINTERPROC, another analyzer of concurrent programs providing a relational static analysis.

3.1 Thread-Modular Analysis

The concept of Thread-Modular Analysis within Abstract Interpretation is formalized in [CH09], [Min12], [Min13b], [Min14]. The idea of Thread-Modular Analysis is to analyze each thread almost separately, by just taking into account "interferences": a change of a global variable value created by another thread. Then, the interferences created by each thread are computed. The last two steps are iterated until a fixpoint is reached (ie, the computed domains and interferences are stable by the iteration). This approach highly depends on the hypothesis on the execution model.

This Thread-Modular Analysis is based on Jone's Rely-Guarantee reasoning presented in [Jon81]. This reasoning is itself based on Hoare logic. Hoare triples are replaced by: $R, G \vdash \{P\}\ \mathit{state}\ \{Q\}$. This intuitively means that if the property P is valid on the program states, before *stat* is executed, and that the actions of other threads are described in R, then Q is true after the execution of *stat*, and the effects of *stat* on the other threads are included in G. This proof method provides rules to abide by, but no automatic computation. Still, fundamental ideas are presented in [Jon81], and developped in [Min12], [Min13b], [Min14].

[Min14] extends [Min12] and [Min13b], by describing new abstractions providing a level of relationality. We will present here a Thread-Modular Static Analysis by Abstract Interpretation, but we will expose a different approach enabling strong relationality in the analysis.

We will present an example of Thread-Modular Analysis in section 4.2, once the execution model is defined.

3.2 Concurinterproc

CONCURINTERPROC is an academic analyzer of concurrent programs designed by Bertrand Jeannet ([Jea13]). It provides a relational static analysis, which is not thread-modular. CONCURINTERPROC is able to analyze procedures, and do a combination of forward and backward analysis. It uses a sort of interleaving semantics to model the preemption of threads. It is freely available online, and has a web interface.

4 Theoretical framework

4.1 Going parallel

In this internship, I used a model presented in [Min12], [Min13b], [Min14].

We assume we have a fixed number of threads, using only global variables, and that the threads only communicate through shared memory. There are no dynamic creation of threads: they are statically created at the initialization. The really important hypothesis here is that we suppose that the execution of assignments and the evaluation of boolean expressions are atomic: we use a sequentially consistent memory model.

For example, if we consider the following threads:

```
if not (x == 0) then     ||
  y = y / x;             ||    x = 0;
```

This can be executed in three different ways:

```
x = 0;
if not (x == 0) then
  y = y / x;
```

```
if not (x == 0) then
  x = 0;
  y = y / x;
```

```
if not (x == 0) then
  y = y / x;
  x = 0;
```

We can see that depending on the order of execution, the second thread might "interfere" with the first thread, and create a division by zero error. We can notice that interferences are only created by assignments of variables.

4.2 An example of Thread-Modular Analysis

We will describe the Thread-Modular Analysis on the example below. Basically, thread t1 and t2 alternate their executions and actively wait for $flag$ to have the good value to (re)start their executions.

```
var flag:int;
initial flag == 1;
```

```
1 thread t1:
2 begin
3   while true do
4     while not (flag == 1) do done;
5     flag = 2;
6   done;
7 end
```

```
1 thread t2:
2 begin
3   while true do
4     while not (flag == 2) do done;
5     flag = 1;
6   done;
7 end
```

We will only study the interferences and not the domains at the end, which are both $\perp$ as the threads do not stop.

At the first iteration: we have the following interference created by thread t1 (and corresponding to assignment at line 5): $flag : 1 \rightsquigarrow 2$. Thread t2 does not create any interference, because it is waiting busily at line 4 (at the beginning, $flag = 1$ and we do not know – yet – that this can change in t1).

At the second iteration: the interferences created by thread t1 do not change, but now we can apply t1's interference to t2, and pass the busy waiting at line 4. This way we have a new interference for t2 $flag : 2 \rightsquigarrow 1$

At the third iteration: we do not detect new interferences: a fixpoint is reached, and thus our analysis is finished.

To represent $flag : 1 \rightsquigarrow 2$ in numerical domains, we will represent transitions as couples (here, $(1; 2)$). We then have to use two variables: we will use one non-primed, initial variable, and one primed, final variable. Thus, we adopt the following notation: $(flag; flag') = (1; 2)$ to state that we have an interference on the variable $flag$, and that the value can change from 1 to 2. This primed-based notation is standard in static analysis: for example, it is used in [ACI10].

5 Denotational semantics

We will now formalize the interferences, and the Thread-Modular Analysis, using denotational semantics, and then abstract this semantics.

5.1 Concrete semantics

Let $\mathcal{T}$ be the set of threads, $\mathcal{L}$ be the set of program points and $\mathcal{V}$ be the set of variables. We define control points as $\mathcal{C} = \mathcal{T} \to \mathcal{L}$: this way, we assign a current location to each thread. Environments are defined as $\mathcal{E} = \mathcal{V} \to \mathbb{Z}$, and the memory $\mathcal{M}$ as $\mathcal{C} \times \mathcal{E}$. We denote the interferences as $\mathcal{I} \subseteq \mathcal{T} \times \mathcal{M}^2$: a transition of states created by a certain thread. We present a concrete semantics with interferences, and explain it below.

$$\mathbb{S}[\![\mathit{stat}]\!]_t, \mathbb{B}[\![\mathit{boolean\ expression}]\!]_t : \mathcal{P}(\mathcal{M}) \times \mathcal{P}(\mathcal{I}) \to \mathcal{P}(\mathcal{M}) \times \mathcal{P}(\mathcal{I}) \tag{1}$$

$$\begin{aligned}
\mathbb{S}[\![^{l_1} X \leftarrow e^{l_2}]\!]_t(R, I) = \\
&\text{let } I_1 = \{t, (c,\rho), (c[t \mapsto l_2], \rho[x \mapsto v]) \mid (c,\rho) \in R, v \in \mathbb{E}[\![e]\!]_t \rho\} \text{ in} \\
&\text{let } R_1 = \{(c', \rho') \mid \exists (c,\rho), (t, (c,\rho), (c,\rho')) \in I_1\} \text{ in} \\
&\text{let } R_2 = \text{lfp}\, \lambda S.\ R_1 \cup \{(c',\rho') \mid \exists t' \in \mathcal{T} \setminus \{t\},\ (c,\rho) \in S,\ (t', (c,\rho), (c',\rho')) \in I\} \text{ in} \\
&\quad R_2, I \cup I_1
\end{aligned} \tag{2}$$

$$\begin{aligned}
\mathbb{B}[\![b]\!]_t(R, I) = \\
&\text{let } R_1 = \{(c,\rho) \in R \subseteq \mathcal{C} \times \mathcal{E} \mid \ \text{true} \ \in \mathbb{E}[\![b]\!]_t \rho\} \text{ in} \\
&\text{let } R_2 = \text{lfp}\, \lambda S.\ R_1 \cup \{(c',\rho') \mid \exists t' \in \mathcal{T} \setminus \{t\},\ (c,\rho) \in S,\ (t', (c,\rho), (c',\rho')) \in I\} \text{ in} \\
&\quad R_2, I
\end{aligned} \tag{3}$$

In (1), we express the fact that $\mathbb{S}[\![.]\!]_t$ needs a statement *stat* of a thread t, a global memory map, and a set of interferences, in order to compute a resulting memory domain and a new set of interferences. Similarly, we can filter a domain with a boolean expression using $\mathbb{B}[\![.]\!]_t$, though in this case, interferences are unchanged.

In (2), I_1 are the interferences created by the new assignment $X \leftarrow e$: it is just a transition from the state before the assignment to the state after the assignment. R_1 is the memory domain before any interference is applied (we only apply the assignment). R_2 is the memory domain after all possible combination of interferences had been taken into account. It can be seen as the smallest set stable by the application of any interference in I, and thus as the *least fixed point* of a function.

(3) is quite similar: we first filter the memory domain to keep environments where b is verified, and we then compute the interferences that can be created by the other threads.

On the contrary to (2) and (3), the following rules are similar to the usual denotational semantics for sequantial programs.

$$\mathbb{S}[\![\text{stat}_1\ ;\ \text{stat}_2]\!]_t = \mathbb{S}[\![\text{stat}_2]\!]_t \circ \mathbb{S}[\![\text{stat}_1]\!]_t \tag{4}$$

$$\begin{aligned}
\mathbb{S}[\![\text{if } b \text{ then } t \text{ else } f]\!]_t X = \\
&\text{let } T = \mathbb{S}[\![t]\!]_t \circ \mathbb{B}[\![b]\!]_t X \text{ in} \\
&\text{let } F = \mathbb{S}[\![f]\!]_t \circ \mathbb{B}[\![\neg b]\!]_t X \text{ in} \\
&\quad T \cup F
\end{aligned} \tag{5}$$

$$\mathbb{S}[\![\text{while } b \text{ do } c]\!]_t X = \mathbb{B}[\![\neg b]\!]_t \circ \text{lfp}\, \lambda Y.(X \cup \mathbb{S}[\![c]\!]_t \circ \mathbb{B}[\![b]\!]_t Y) \tag{6}$$

Let stats_t be the statements of thread t, and M a memory containing initialization of variables. Doing a Thread-Modular Analysis is equivalent to computing lfp f, with:

$$f : \left\{ \begin{array}{ccl} (\mathcal{T} \to \mathcal{P}(\mathcal{M})) \times (\mathcal{T} \to \mathcal{P}(\mathcal{I})) & \longrightarrow & (\mathcal{T} \to \mathcal{P}(\mathcal{M})) \times (\mathcal{T} \to \mathcal{P}(\mathcal{I})) \\ R, I & \longmapsto & t \mapsto R'_t,\ t \mapsto I'_t \\ & & \text{where } R'_t, I'_t = \mathbb{S}[\![\text{stats}_t]\!]_t(M,\ \bigcup_{t' \in \mathcal{T} \setminus \{t\}} I(t')) \end{array} \right.$$

It is necessary to compute a *fixpoint* of f, and not only the first iterations: at the beginning, some interferences are not yet discovered, and thus the analysis is not yet sound. When the set of interferences is stable, then the analysis is sound.

This semantics is sound (all execution cases are taken into account) and complete (any property of the program can be proved using this semantics). It is however too precise to be computed: we will first abstract this semantics in the next sections.

5.2 Abstractions

We use the following abstractions:

$$\alpha_{R,t} : \left\{ \begin{array}{ccl} \mathcal{P}(\mathcal{M}) & \longrightarrow & \mathcal{L} \to \mathcal{P}(\mathcal{E}) \\ X & \longmapsto & \lambda L.\{e \mid (c,e) \in X\ \wedge c(t) = L\} \end{array} \right.$$

$$\alpha_I : \left\{ \begin{array}{ccl} \mathcal{P}(\mathcal{I}) & \longrightarrow & \mathcal{T} \to \mathcal{P}(\mathcal{E}^2) \\ X & \longmapsto & \lambda t.\{(b,e) \in \mathcal{E}^2 \mid \exists (c_b, c_e) \in \mathcal{C}^2, (t, (c_b, b), (c_e, e)) \in X\} \end{array} \right.$$

Their purpose is to forget the control points of the other threads, in order to reduce the cost of computation in the coming abstract semantics. It also permits to reduce the cost of representation of interferences. Using $\alpha_{R,t}$, we keep the control points of the current thread in order to be locally flow-sensitive – that is the analysis is sensitive to the order of execution on the current thread t. In α_I, we associate to each thread the set of interferences it created.

We suppose we have a concretization $\gamma_{\mathcal{E}}$ of a numerical domain into the memory domain, and another concretization $\gamma_{\mathcal{ITF}}$ of a relational numerical domain into interferences.

$$\gamma_{\mathcal{E}} : (\mathcal{L} \to \mathcal{D}^{\#}) \longrightarrow (\mathcal{L} \to \mathcal{P}(\mathcal{E}))$$
$$\gamma_{\mathcal{ITF}} : (\mathcal{T} \to \mathcal{I}^{\#}) \longrightarrow (\mathcal{T} \to \mathcal{P}(\mathcal{E}^2))$$

$\mathcal{P}(\mathcal{E}^2)$ can represent a relation: instead of storing one value for one variable, we store a couple of values for a variable and its primed sibling. We thus have an initial element and a final element, and they define a transition. This was mentionned in section 4.2. A more relational-oriented example is the following: if we want to say that an interference can only increase a variable x, we could write this as $x' \geq x$.

In this internship, I mainly worked on the polyhedra abstract domain, but we have seen previously that other domains such as octagons exist.

5.3 Abstract semantics

We suppose we have some built-in operations, such as assigning a set of expressions to a set of variables, adding uninitialized variables to a domain, renaming and deleting a variable in a domain. We also suppose the we have a join (ie an abstraction of the union), as well as intersection and widening operators. These are really standard operations, implemented in both Apron and Bddapron. Let $\mathcal{A}$ be the set of arithmetical expressions. Here are the signatures of the previously mentionned functions:

$$\begin{aligned} \text{assign} &: \mathcal{D}^{\#} \times \mathcal{P}(\mathcal{V} \times \mathcal{A}) \to \mathcal{D}^{\#} \\ \text{add} &: \mathcal{D}^{\#} \times \mathcal{P}(\mathcal{V}) \to \mathcal{D}^{\#} \\ \text{rename} &: \mathcal{D}^{\#} \times \mathcal{P}(\mathcal{V}^2) \to \mathcal{D}^{\#} \\ \text{delete} &: \mathcal{D}^{\#} \times \mathcal{P}(\mathcal{V}) \to \mathcal{D}^{\#} \end{aligned}$$

We would like a general method to apply interferences to a domain. Let us start with an example before formalizing the functions: suppose we have a memory domain $R = 0 \leq x \leq 10, -10 \leq y \leq 10$, and interferences like $I = 0 \leq x \leq 12, x' = x + 1, y' = y$ created by thread r. This means that we know an interference that changes x in $x + 1$, if $0 \leq x \leq 12$, but that this interference does not modify y. We here give an example over the abstract domain. It corresponds to finding the following concrete set: $\{(c', \rho') \mid \exists t \in \mathcal{T} \setminus \{r\}, \exists (c, \rho) \in R, (t, (c, \rho), (c', \rho')) \in I\}$

1. We need to add variables to the memory domain, so it has the same dimension as the interferences. So we need to add the variables x', y' to the intial domain, but we do not initialize them: we just want to gain dimensions, but without imposing any conditions on these new dimensions – for now.

2. Then, we can apply the interferences by intersecting the modified memory domain and the interferences. We get that $0 \leq x \leq 10, -10 \leq y \leq 10, x' = x + 1, y' = y$. In the concrete world, this means that if we have a memory domain R and an interference I, we are searching for $\{(c, \rho), (c', \rho') \mid \exists t \in \mathcal{T} \setminus \{r\}, \exists (c, \rho) \in R,\ (t, (c, \rho), (c', \rho')) \in I\}$

3. We finally need to get the result: we should remove the variables x and y, and rename x' into x and y' into y. We obtain $1 \leq x \leq 11, -10 \leq y \leq 10$.

We can see that we get a sound abstraction of the concrete formula proposed above.

On top of the previously mentionned functions, we implement new functions called *extend*, *img* and *apply*. We introduce some notations: $\mathcal{D}^{\#}_n$ is the set of domains on $\mathcal{D}^{\#}$ with n variables, $\Delta^{\#}_{2n} \subseteq \mathcal{D}^{\#}_{2n}$ must satisfy the following property: $\exists X \subseteq \text{Vars}(\Delta^\#_{2n}), \forall x \in X, |X| = n \ \wedge \ x' \in \text{Vars}(\Delta^\#_{2n})$. This last set will be useful to denote an environment where each variable has a copy of itself. $\text{Var} : \mathcal{D}^\# \to \mathcal{P}(\mathcal{V})$ associates to each domain its variables. We suppose that $n = |\mathcal{V}|$.

$$\text{extend} : \begin{cases} \mathcal{D}^\#_n & \longrightarrow & \Delta^\#_{2n} \\ R^\# & \longmapsto & \text{add}(R^\#, \{x' \mid x \in \text{Vars}(R^\#)\}) \end{cases}$$

$$\text{img} : \begin{cases} \Delta^\#_{2n} & \longrightarrow & \mathcal{D}^\#_n \\ R^\# & \longmapsto & \text{let } X = \{x \in \text{Vars}(R^\#) \mid x' \in \text{Vars}(R^\#)\} \text{ in} \\ & & \text{let } R^\#_1 = \text{delete}(R^\#, \{x \mid x \in X\}) \text{ in} \\ & & \text{rename}(R^\#_2, \{(x', x) \mid x \in X\}) \end{cases}$$

$$\text{apply} : \begin{cases} \mathcal{D}^\#_n \times \mathcal{I}^\# & \longrightarrow & \mathcal{D}^\#_n \\ R^\#, I^\# & \longmapsto & \text{let } R^\#_1 = \text{extend}(R^\#) \text{ in} \\ & & \text{let } R^\#_2 = R^\#_1 \cap^\# I^\# \text{ in} \\ & & \text{img}(R^\#_2) \end{cases}$$

The function *extend* creates a copy of every variable of the domain, whereas *img* returns the image set of a relation. With these two functions, we can now give a procedure computing the result of an interference, giving an initial memory domain: we first have to add copies of the variables of the memory domain (as in the first part in the example above). We can then intersect the resulting memory domain $R^\#_1$ with the interferences (second part of the example above). We then have to get the image of the relation, which is the part where the variables have quotes (third part of the example above). This is achieved with the function *img*.

The obtained abstract semantics has the following signature:

$$\mathbb{S}^\#[\![stat]\!]_t : (\mathcal{L} \to \mathcal{D}^\#) \times (\mathcal{T} \to \mathcal{I}^\#) \longrightarrow (\mathcal{L} \to \mathcal{D}^\#) \times (\mathcal{T} \to \mathcal{I}^\#)$$

We now abstract the concrete semantics (2), (3):

$$\begin{aligned} \mathbb{S}^\#[\![^{l_1} X \leftarrow e^{l_2}]\!]_t(R^\#, I^\#) = & \\ & \text{let } I^\#_g &&= \bigcup^\#_{t' \in \mathcal{T} \setminus \{t\}} \{I^\#(t')\} \text{ in} \\ & \text{let } I^\#_l &&= \text{extend}(R^\#(l_1)) \text{ in} \\ & \text{let } I^\#_l &&= \text{assign}(I^\#_l, \{(X', e)\} \cup \{(Y, Y') \mid Y \in Var(R^\#) \setminus \{X\}\}) \text{ in} \\ & \text{let } R^\#_1 &&= \text{img}(I^\#_l) \text{ in} \\ & \text{let } R^\#_2 &&= \lim \lambda Y^\#.\ Y^\# \triangledown (R^\#_1 \cup^\# \text{apply}(Y^\#, I^\#_g)) \text{ in} \\ & && R^\#[l_2 \mapsto R^\#_2], I^\#[t \mapsto I^\#(t) \cup^\# I^\#_l] \\ \mathbb{B}^\#[\![^{l_1} b^{l_2}]\!]_t(R^\#, I^\#) = & \\ & \text{let } I^\#_g &&= \bigcup^\#_{t' \in \mathcal{T} \setminus \{t\}} \{I^\#(t')\} \text{ in} \\ & \text{let } R^\#_1 &&= \mathbb{F}^\#[\![b]\!](R^\#(l_1)) \text{ in} \\ & \text{let } R^\#_2 &&= \lim \lambda Y^\#.\ Y^\# \triangledown (R^\#_1 \cup^\# \text{apply}(Y^\#, I^\#_g)) \text{ in} \\ & && R^\#[l_2 \mapsto R^\#_2], I^\# \end{aligned}$$

We changed the lfp operator for a limit and a widening operator. This way, convergence is ensured by the definition of the widening operator.
This abstraction of the other parts of the semantics is similar to what was presented in section 2.3.

$$\begin{aligned} \mathbb{S}^\#[\![\text{if } b \text{ then } t \text{ else } f]\!]_t(R^\#, I^\#) = & \\ & \text{let } T &&= \mathbb{S}^\#[\![t]\!]_t \circ \mathbb{B}^\#[\![b]\!]_t(R^\#, I^\#) \text{ in} \\ & \text{let } F &&= \mathbb{S}^\#[\![f]\!]_t \circ \mathbb{B}^\#[\![\neg b]\!]_t(R^\#, I^\#) \text{ in} \\ & && T \dot\cup^\# F \\ \mathbb{S}^\#[\![\text{while } b \text{ do } c]\!]_t(R^\#, I^\#) & &&= \mathbb{B}^\#[\![\neg b]\!]_t(\lim \lambda Y. Y \dot\triangledown (R^\# \dot\cup^\# \mathbb{S}^\#[\![c]\!]_t \circ \mathbb{B}^\#[\![b]\!]_t(Y, I^\#)) \end{aligned}$$

6 Implementation of a BAsic Thread-Modular ANalyzer

I also implemented an analyzer prototype, called Batman, in order to assess the precision and scalability of the analysis presented above. It has roughly 1700 lines of OCaml code at the time of writing. It uses the Ocaml bindings of the Apron ([JM09]) or the Bddapron ([Jea]) library to manipulate abstract domains. I implemented a simple widening with thresholds, as well as increasing and decreasing iterations. I am very grateful to Antoine Miné, who made the source code of a prototype analyzer presented in [Min13a] available. It has been really helpful to have an example of an analyzer using Apron. The analyzer uses functors, so that changing from one relational domain to another is really simple.

In order to compare the performances of ConcurInterProc and Batman, I used a similar type of input. The basic language I defined supports integer and boolean variables, if and while statements, assignments. It does not supports procedures, on the contrary to ConcurInterProc

This program is available at: http://rmonat.fr/batman.html. It is released under the GPLv3 license.

7 Results

We will first present some detailed examples, and then study the scalability of this approach.

7.1 Some examples

In this section, I will show some detailed examples.

Clock example. This example was mentionned in [Min14]: using a combination of abstractions that were able to express whether a variable is monotonic, the author was able to analyze the following program. However, the abstractions were very specialized, and [Min14] has to resort to a complicated trace semantics (compared with the state-based semantics we use here). The technique we proposed express simply the monotonicity with $x' \geq x$, with general and more robust domains.

```
var z:int, h:int, c:int, t:int, l:int, r:int;
initial z == 0 and h == 0 and c == 0 and t == 0 and l == 0;
```

```
thread t1:
begin
 while (z < 1000) do
  z = z + 1;
  if (h < 100) then
   h = h + 1;
  endif;
 done;
end
```

```
thread t2:
begin
 while (z < 1000) do
  z = z + 1;
  c = h;
 done;
end
```

```
thread t3:
begin
 while (z < 1000) do
  if ([0, 1] == 0) then
   t = 0;
  else
   t = t + c - l;
  endif;
  l = c;
 done;
end
```

Using Batman, we can find that $0 \leq t \leq l \leq c \leq h \leq 100$, $z \geq 1000$. This illustrates the fact that we can find hidden relations. We can see that $l \leq c$ in the assignment corresponding in thread t3, and $c \leq h$ in thread t2, and $h < 100$ in thread t1. We can then find that $0 \leq t \leq l$. ConcurInterProc does almost the same relations, except that $h < 100$ is replaced by $h \leq z$ (but this might be only a benefit from the widening with threshold implemented in Batman). However, ConcurInterProc uses $0.746s$ to analyze this program, whereas Batman uses $0.426s$. This difference may be due to the use of Bddapron instead of Apron, but we will see in the scalability section that Batman is usually faster than ConcurInterProc.

Flow-insensitivity. We have seen that our abstractions removed the flow-sensitivity of the interferences. This kind of analysis, as mentionned in [Min14], does not analyze the following program properly:

```
x = x + 1;
```

```
x = x + 1;
```

We can only infer that interferences are of the form $x' = x + 1$, but we do not have any bound, because they are unstable by the application of the analysis (we would first have $x' = x + 1, x = 0$, but then $x' = x + 1, 0 \leq x \leq 1$, and so on until the widening breaks the unstable bound). Thus, BATMAN can only infer that $x \geq 1$, whereas CONCURINTERPROC finds that $x = 2$ at the end.

Mutual exclusion algorithms. At the time of writing, mutual exclusion algorithm cannot be proved by BATMAN, mainly because of the flow-insensitivity illustrated above. We tried to support boolean variables using Bddapron. Bddapron is able to partition numerical domains with respect to the values of boolean variables. Sadly, this was not sufficient: too much interferences where considered appliable in the analysis. We were not able to infer any information on *turn*; on $b1$ in $T0$ and on $b0$ in $T1$. This code has been taken from CONCURINTERPROC's examples, and is an implementation of Peterson's mutual exclusion algorithm. Improvements to tackle this issue are discussed in section 8.

```
var b0:bool, b1:bool, turn:bool;
initial not b0 and not b1;
```

```
thread T0:
begin
  while true do
    b0 = true;
    turn = false;
    assume(b1==false or turn==true);
    b0 = false;
  done;
end
```

```
thread T1:
begin
  while true do
    b1 = true;
    turn = true;
    assume(b0==false or turn==false);
    b1 = false;
  done;
end
```

A kind of synchronization algorithm. However, some kind of synchronization algorithm are correctly analyzed (even with Apron):

```
var flag:int, x:int;
initial flag == 1 and x == 1;
```

```
1 thread t1:
2 begin
3   while(true) do
4     while not (flag == 1) do done;
5     x = 1;
6     flag = 2;
7   done;
8 end
```

```
1 thread t2:
2 begin
3   while(true) do
4     while not (flag == 2) do done;
5     x = 2;
6     flag = 1;
7   done;
8 end
```

In this program really similar to what was presented in section 4.2, threads t1 and t2 alternate their executions. In addition, we can infer that line 5 contains a mutual exclusion, so that we are sure that $x = 1$ in t1, and $x = 2$ in t2. One of the interferences created by t2 is $flag' + x = 3$. We can deduce from this that if $x = 1$ then $flag' = 2$ and if $x = 2$ then $flag' = 1$.

7.2 Scalability

We will discuss scalability of this analysis in terms of number of variables and number of threads. I built some Python scripts to automatize the benchmarks presented in figure 3.

Scalability in the number of variables. The BATMAN analyzer does not scale in the number of variables. This is not really surprising because the polyhedra domain is used, and analyzers of sequential programs using the polyhedra domain do not scale in the number of variables too. As BATMAN uses Apron, it is really easy to switch to another domain – such as octagons. To test the octagons domain, and compare it with polyhedra, I focused on one program. On this example, the conclusion is that octagons scale much better than polyhedra: octagons look like they have a cubic complexity in the number of variables, whereas polyhedra have an exponential complexity. Other methods such as variable packing presented in [Min06] could also be used to improve scalability in this domain.

The graph in figure 3a has been generated by copying a simple program model, but by changing the variables name in each thread, so that we have n instances of a same program running in parallel but not interfering. The y-axis has a log scale. Even without any interference between any threads, this shows that the current tools do not scale well in the number of variables.

Scalability in the number of threads. We tested the scalability in the number of threads, mostly by copying a program composed of one or two threads and changing one parameter (such as a bound). We first used a model based on the following program, presented in [Min13b] and [Min14]:

```
var x:int,y:int,z:int;
initial x == 0 and y == 0 and z == 0;
```

```
thread t1:
begin
  while (z < 10000) do
    z = z + 1;
    if (y < 10) then
      y = y + 1;
    endif;
  done;
end
```

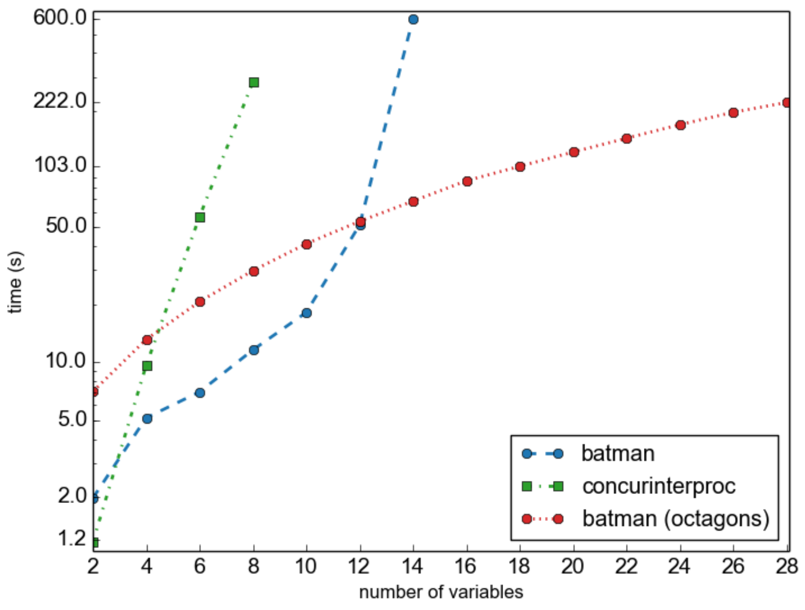
```
thread t2:
begin
  while (z < 10000) do
    z = z + 1;
    if (x <= y) then
      x = x + 1;
    endif;
  done;
end
```

The $y < 10$ was replaced by $y < c$ where c is a random value between 10 and 100. One thing difficult to infer, and mentionned in [Min14], is that $x + 1 \leq y$. This is true, because at the beginning, $x = y$ and then x is incremented if and only if $x \leq y$ (and the only possible interference increments y). Moreover, this is correctly inferred by Batman. The results are displayed in figure 3b. The y-axis has a log scale.

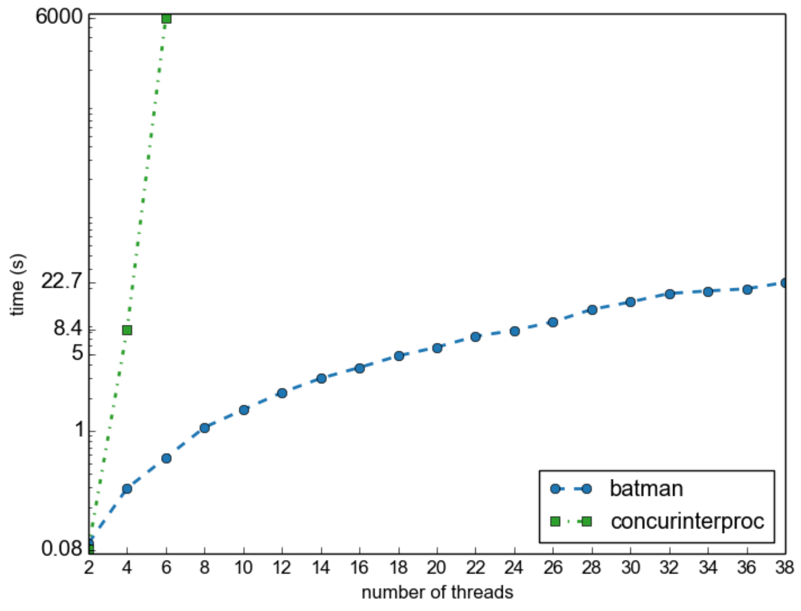
We present another result in figure 3c, by considering multiple copies of the "alternating threads" presented in section 7.1. We changed the transition to a nondeterministic one: line 6 has been replaced by $flag = [1, m]$, meaning that the value is chosen at random between 1 and m. Here m is the total number of threads. We can see that on this example, Batman scales pretty well, though it appears it has a quadratic complexity in the number of threads. This is due to the following point: in our thread-modular analysis, we analyze each one of the m threads, and in each thread we should take into account $m - 1$ interferences. We do not show ConcurInterProc's results here, as it did not scale as well as Batman (for example, it was already above 20 minutes of computation with 50 threads).

We should highlight that this experimental quadratic complexity in the number of threads is a real improvement over the theoretical cost. An analysis using explicitely the product of control spaces would have a complexity exponential in the number of threads: let c be the average number of control points of a thread, and t be the number of threads. The product of control spaces has a size c^t, and thus the explicit analysis has a complexity that is exponential in the number of threads.

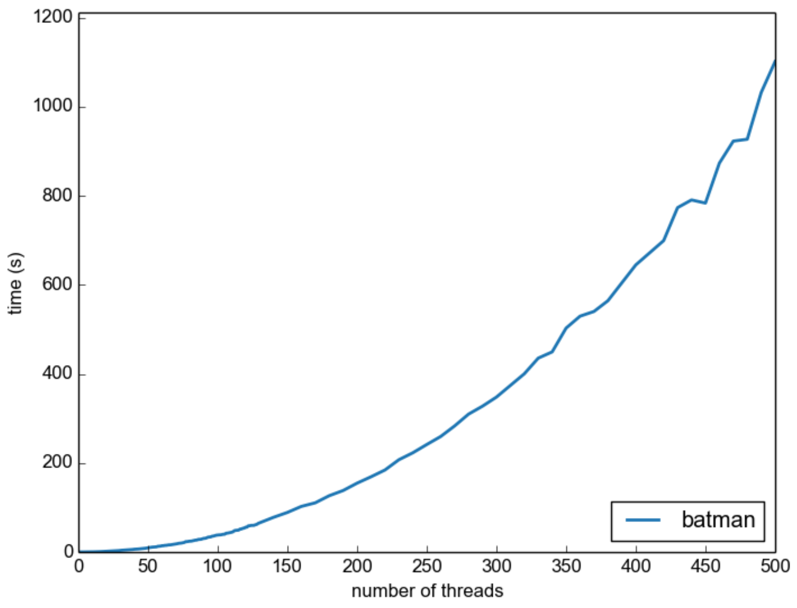
Memory use. We did not measure precisely the memory usage, but we still noticed a big difference between Batman and ConcurInterProc. In the tests corresponding to figure 3b, ConcurInterProc used 15GB of RAM when processing 6 threads, whereas Batman used less than 500MB during these tests. On longer tests (as in 3c), the memory use has never been a bottleneck in Batman: it was using less than 500MB, and the memory usage looked roughly linear in the number of threads.



(a) Scalability in the number of variables



(b) Scalability in the number of threads



(c) Scalability in the number of threads – “alternating threads”

Figure 3

8 Conclusion

We have developped a relational abstraction of interferences, that permits us to analyze more precisely programs. This approach is based on Abstract Interpretation and Thread-Modular Analysis, and has been validated by developping and testing an analyzer called BATMAN. This approach promises to be scalable in the number of threads, on the contrary to other methods.
There are many other things to explore, among which:

- weaker memory models: we have assumed a sequentially consistent model of execution, but computers may execute programs under weakly consistent memories, where threads do not have a coherent view of the memory. This memory model is much harder to analyze, and was out of the scope of this internship.
- locks: it could be interesting to take into account locks, and divide interferences in the cases of whether a variable is locked or not. This could be implemented using Bddapron.
- setting the flow-insensitivity: being able to set the flow-sensitivity in the interferences would result in more precise analyses, such as CONCURINTERPROC does. This way, we might be able to prove usual mutual exclusion algorithms, which is not possible at the time of writing. This would still be different from CONCURINTERPROC, as we would be able to choose a precision, from what is currently proposed to what CONCURINTERPROC proposes. Also, we could locally be more precise, and only where it matters, to scale better.
- other abstractions: interferences can be represented and abstracted in other ways, but I did not have the time to try anything else.
- weaker relational domains: we did not test weaker relational domains such as octagons extensively, but they surely could handle an analysis almost as precise but at a reduced cost.
- parallelization: when we analyze one thread t, we only need to read the other interferences, and then to modify the domain and interferences of the thread t. This way, the analysis of each thread is independant and could be parallelized in the implementation.
- iterations of the analysis: at each step of the analysis, we are currently analyzing each thread. It might be interesting to use other iterations, more intelligent, such as chaotic iterations.
- on the implementation side, it would be interesting to support backward analysis, as well as procedures.
- it would also be interesting to test this approach on a real programming language, such as the C, and test it on real programs then.

9 Acknowledgments

I would like to thank a lot my supervisor, Antoine Miné, for all the time he gave me, and his availability. The many advices and ressources helped a lot to quickly understand Abstract Interpretation. The very interesting discussions permitted to achieve an enlightening work within a restricted time. This introduction to research has been a thrilling experience I am looking forward to continue. I also want to thank the members of the ANTIQUE team I have seen, for the warm welcome they gave me, and the very interesting discussions I had with them.

References

[ACI10] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*, 267(1):3–16, 2010.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[CH09] Jean-Loup Carre and Charles Hymans. From single-thread to multithreaded: An efficient static analysis algorithm. *arXiv preprint arXiv:0910.5833*, 2009.

[Cou02] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1):47–103, 2002.

[HH12] Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In *Static Analysis*, pages 198–213. Springer, 2012.

[Jea] Bertrand Jeannet. Bddapron. http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/bddapron.pdf.

[Jea13] Bertrand Jeannet. Relational interprocedural verification of concurrent programs. *Software & Systems Modeling*, 12(2):285–306, 2013.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667. Springer, 2009.

[Jon81] Cliff B Jones. *Development methods for computer programs including a notion of interference.* PhD thesis, Oxford University Computing Laboratory, Jun. 1981.

[Min04] Antoine Miné. *Weakly relational numerical abstract domains.* PhD thesis, École Polytechnique, Dec. 2004. http://www.di.ens.fr/~mine/these/these-color.pdf.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC)*, 19(1):31–100, 2006. http://www.di.ens.fr/~mine/publi/article-mine-HOSC06.pdf.

[Min12] Antoine Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):63, Mar. 2012. http://www.di.ens.fr/~mine/publi/article-mine-LMCS12.pdf.

[Min13a] Antoine Miné. Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. *Science of Computer Programming (SCP)*, page 33, Oct. 2013. http://www.di.ens.fr/~mine/publi/article-mine-SCP13.pdf.

[Min13b] Antoine Miné. *Static analysis by abstract interpretation of concurrent programs.* Habilitation à diriger des recherches, Ecole Normale Supérieure de Paris - ENS Paris, November 2013. https://tel.archives-ouvertes.fr/tel-00903447/file/hdr-compact-col.pdf.

[Min14] Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In *Proc. of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'14)*, volume 8318 of *Lecture Notes in Computer Science (LNCS)*, pages 39–58. Springer, Jan. 2014. http://www.di.ens.fr/~mine/publi/article-mine-VMCAI14.pdf.

[Rin01] Martin Rinard. Analysis of multithreaded programs. In *Static Analysis*, pages 1–19. Springer, 2001.

[Urb15] Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs.* PhD thesis, École Normale Supérieure, July 2015. https://tel.archives-ouvertes.fr/tel-01176641/file/main.pdf.

A Presentation of the ANTIQUE team

The ANTIQUE team is one of the teams of the Computer Science Department of the ENS Ulm. These teams are also affiliated with CNRS and INRIA. The lab was located in the center of Paris, but I did not meet members of the other teams of the Computer Science Department. The head of the team is Xavier Rival, and there are 4 other permanent members: Vincent Danos, Cezara Drăgoi, Jerôme Feret and Antoine Miné. I also saw one postdoc, Ilias Garnier, and 5 PhD students: Mehdi Bouaziz, Huisong Li, Jiangchao Liu, Thibault Suzanne, Antoine Toubhans. I also assisted to the PhD defense of Caterina Urban ([Urb15]). There were two INRIA engineers in the team: François Berenger and Quyen Ly Kim.