Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 267 (2010) 29–42

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Some Experience on the Software Engineering of Abstract Interpretation Tools

Bertrand Jeannet[2]

INRIA Rhône-Alpes, Grenoble, France

Abstract

The "right" way of writing and structuring compilers is well-known. The situation is a bit less clear for static analysis tools. It seems to us that a static analysis tool is ideally decomposed into three building blocks: (1) a front-end, which parses programs, generates semantic equations, and supervises the analysis process; (2) a fixpoint equation solver, which takes equations and solves them; (3) and an abstract domain, on which equations are interpreted. The expected advantages of such a modular structure is the ability of sharing development efforts between analyzers for different languages, using common solvers and abstract domains. However putting in practice such ideal concepts is not so easy, and some static analyzers merge for instance the blocks (1) and (2).
We show how we instantiated these principles with three different static analyzers (addressing resp. imperative sequential programs, imperative concurrent programs, and synchronous dataflow programs), a generic fixpoint solver (Fixpoint), and two different abstract domains. We discussed our experience on the advantages and the limits of this approach compared to related work.

Keywords: Abstract interpretation, software engineering, static analysis

1 Introduction

The right way of writing and structuring compilers is well-known, described in details in the relevant textbooks, and effectively put in practice in the vast majority of compilers. Modern and even old compilers follows the architecture sketched on Fig. 1. The question of what is the "right" architecture of a static analyzer has been less discussed (see for instance [29,27]). By static analyzers we mean in this paper mainly verification tools based on abstract interpretation, performing sophisticated analyses on the reachable states of a program, such as linear relation analysis [15] or shape analysis [7].

In our mind, the ideal architecture of a static analyzer is the one depicted on Fig. 2. Static analyzers for programming languages shares some common aspects

[1] This work was supported by the french ANR project ASOPT.

[2] Email: `bertrand.jeannet@inrialpes.fr`

1571-0661 © 2010 Elsevier B.V. Open access under CC BY-NC-ND license.

doi:10.1016/j.entcs.2010.09.016

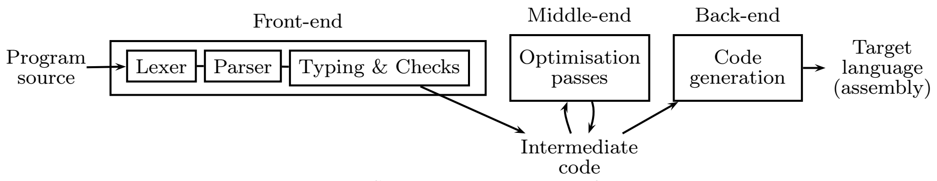


Fig. 1. Standard architecture of a compiler

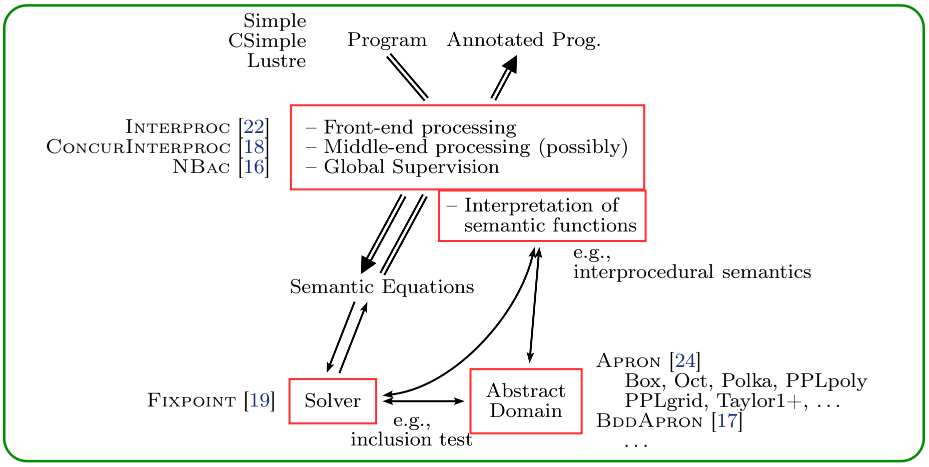
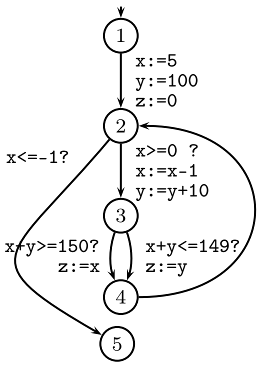


Fig. 2. Typical/ideal architecture of a static analyzer, with the components considered in this paper

```
(1) x := 5;
    y := 100;
    z := 0;
(2) while x>=0 do
      x := x-1;
      y := y+10;
(3)   if x+y>=150 then
       z := x
      else
       z := y;
      endif;
(4) done;
(5)
```



State-space $S = [1,5] \times \mathbb{Z}^3$

Equation system on $\wp(\mathbb{Z}^3)$:

$$\begin{aligned}
X_1 &= \top\\
X_2 &= [\![\mathtt{z := 0}]\!] \circ [\![\mathtt{y := 100}]\!] \circ [\![\mathtt{x := 5}]\!](\mathtt{X_1}) \cup \mathtt{X_4}\\
X_3 &= [\![\mathtt{y := y + 10}]\!] \circ [\![\mathtt{x := x - 1}]\!] \circ [\![\mathtt{x >= 0?}]\!](\mathtt{X_2})\\
X_4 &= [\![\mathtt{z := x}]\!] \circ [\![\mathtt{x + y >= 150?}]\!](\mathtt{X_3}) \cup\\
&\quad\ [\![\mathtt{z := y}]\!] \circ [\![\mathtt{x + y <= 149?}]\!](\mathtt{X_3})\\
X_5 &= [\![\mathtt{x <= -1?}]\!](\mathtt{X_2})
\end{aligned}$$

(a) Program (b) Control Flow Graph (c) Equation system

Fig. 3. Translating from a Simple program to an equation system

with compilers: processing the program source, translating it into an intermediate representation, and dealing with the semantics of the prog. language. Fig. 3 illustrates this process. But the back-end step of compilers is replaced in static analyzers by the solving of *fixpoint equations* (see Fig. 3(c)) on *an abstract domain*. Moreover, static analyzers need to convert the computed solution of the fixpoint equations in term of relevant information w.r.t. the program and the user: this may be a program annotated with invariants, or control points turned in red because some errors have been discovered.

Theoretically, the three components of the architecture of Fig. 2 should be largely independent of each other. This implies that

- the APIs of each component are well identified;
- there does not exist implicit dependencies between them;
- so that a component can be substituted by another one in an analyzer without requiring a large re-engineering of the other components.

In practice things are less ideal. A good way to evaluate the modularity of a modular analyzer is to investigate to which extend are the different components sensitive to:
(1) a new input language ?
(2) a new datatype to be analyzed ?
(3) a new abstract domain ?

We report in this paper our experience about these questions, which is based on our implementation of several static analyzers and associated components depicted on Fig. 2. More precisely, we will consider

1. Three (front-end) analyzers:
   - the Interproc analyzer [24,22] for a simple imperative language, Simple, describing sequential programs with recursive procedures and call-by-value parameter passing;
   - the ConcurInterproc analyzer [21,18], which extends it to concurrent language composed of a fixed number of threads, CSimple;
   - The NBac analyzer [23,16] for the synchronous dataflow language Lustre.
2. A generic fixpoint solver, called Fixpoint, that is used by our three analyzers, as well as other external users [19].
3. Two (families of) abstract domains, namely Apron [24], dedicated to the analysis of numerical variables, and BddApron [17], that handles combination of finite-state and numerical variables.

We first give in §2 an example of the analysis of a simple program to illustrate the architecture. Then we study the abstract domain component in §3. We discuss in §4 the issues which motivated the design on the Fixpoint generic fixpoint solver, common to our three analyzers, and last we look at the frontend parts of our analyzers in §5. In all these sections we will discuss what are the success and the limitations of our approaches, and we investigate whether recent advances in abstract domains or in equation solving fit in our framework.

2 Example: analysis of a small program

We show in this section how the program of Fig. 3(a) is analyzed to illustrate the principles of Fig. 2.

- One first builds its control flow graph (CFG), see Fig. 3(b). On the CFG basic blocks have been identified and label edges, branch instructions are encoded in the edges, and the remaining instructions are elementary instructions such as assignments and conditionals (marked by "`?`").
- One then translates the CFG to the equation system of Fig. 3(c), in which $[\![\texttt{instr}]\!] : \wp(\mathbb{Z}^3) \to \wp(\mathbb{Z}^3)$ denotes the effect of the instruction "`instr`" on a

$$\begin{aligned}
Y_k^0 &= \bot,\ k \in [1,5] \\
Y_1^\infty &= \top \\
Y_2^{n+1} &= Y_2^n \nabla (Y_4^n \sqcup \\
&\quad [\![\mathtt{z := 0}]\!]^\sharp \circ [\![\mathtt{y := 100}]\!]^\sharp \circ [\![\mathtt{x := 5}]\!]^\sharp(\mathtt{Y}_1^\infty)) \\
Y_3^{n+1} &= [\![\mathtt{y := y + 10}]\!]^\sharp \circ [\![\mathtt{x := x - 1}]\!]^\sharp \circ \\
&\quad [\![\mathtt{x >= 0?}]\!]^\sharp(\mathtt{Y}_2^\mathtt{n}) \\
Y_4^{n+1} &= [\![\mathtt{z := x}]\!]^\sharp \circ [\![\mathtt{x + y >= 150?}]\!]^\sharp(\mathtt{Y}_3^\mathtt{n}) \sqcup \\
&\quad [\![\mathtt{z := y}]\!]^\sharp \circ [\![\mathtt{x + y <= 149?}]\!]^\sharp(\mathtt{Y}_3^\mathtt{n}) \\
Y_5^\infty &= [\![\mathtt{x <= -1?}]\!]^\sharp(\mathtt{Y}_2^\infty)
\end{aligned}$$

(a) Sequence implementing an iteration strategy with widening.

$$\begin{aligned}
Z_k^0 &= Y_k^\infty,\ k \in [1,5] \\
Z_1^N &= \top \\
Z_2^{n+1} &= (Z_4^n \sqcup \\
&\quad [\![\mathtt{z := 0}]\!]^\sharp \circ [\![\mathtt{y := 100}]\!]^\sharp \circ [\![\mathtt{x := 5}]\!]^\sharp(\mathtt{Z}_1^\mathtt{N})) \\
Z_3^{n+1} &= [\![\mathtt{y := y + 10}]\!]^\sharp \circ [\![\mathtt{x := x - 1}]\!]^\sharp \circ \\
&\quad [\![\mathtt{x >= 0?}]\!]^\sharp(\mathtt{Z}_2^\mathtt{n}) \\
Z_4^{n+1} &= [\![\mathtt{z := x}]\!]^\sharp \circ [\![\mathtt{x + y >= 150?}]\!]^\sharp(\mathtt{Z}_3^\mathtt{n}) \sqcup \\
&\quad [\![\mathtt{z := y}]\!]^\sharp \circ [\![\mathtt{x + y <= 149?}]\!]^\sharp(\mathtt{Z}_3^\mathtt{n}) \\
Z_5^N &= [\![\mathtt{x <= -1?}]\!]^\sharp(\mathtt{Z}_2^\mathtt{N})
\end{aligned}$$

(b) Descending Sequence, computed up to N steps.

Fig. 4. Solving iteratively with Ascending and Descending Sequences

concrete property on variables, which belong to $\wp(\mathbb{Z}^3)$. For instance,

$$\begin{aligned}
&[\![\mathtt{x := x + 1}]\!](X) = \\
&\qquad \{(x', y', z') \in \mathbb{Z}^3 \mid \exists (x, y, z) \in X \ :\ x' = x+1 \wedge y' = y \wedge z' = z\}.
\end{aligned}$$

The generation of such equations is done by the front-end part of the analyzer.

Now these equations need to be solved. The classical abstract interpretation approach consists in transposing them from the concrete domain of properties (here, $\wp(\mathbb{Z}^3)$) to some abstract domain, and to solve them iteratively. In this process, the abstract domain component is in charge of interpreting original operations ($[\![instr]\!]$, $\cup, \dots$) in the abstract domain, with the help of the front-end component for complex operations that are not supported by the abstract domain (*e.g.,* procedure calls and returns) , whereas the solver is in charge of managing the iterative process and of testing the convergence.

For instance, the choice of an iteration strategy for solving the equation system Fig. 3(c) may result in the sequence of Fig. 4(a) in which $[\![\mathtt{instr}]\!]^\sharp$ and $\sqcup$ are the interpretation of $[\![\mathtt{instr}]\!]$ and $\cup$ in the abstract domain, ∇ is the widening operator applied at the loop head, and Y_k^∞ indicates the value of Y_k obtained after convergence. This sequence means that we compute Y_1^∞ once, one updates the values of Y_2, Y_3, Y_4 until convergence, and then one computes Y_5^∞. After having computed a post-fixpoint, one can try to recover some information lost by widening by computing the descending sequence of Fig. 4(b), in which the widening operator is not applied any more. This sequence does not converge in general and is stopped after a number N of steps. With the abstract domain of convex polyhedra [15] one obtains$Y_5^\infty = (x \leq -1) \wedge (10x + y = 150)$. and $Z_5^1 = (x = -1) \wedge (y = 160) \wedge (z = -1)$.

At last, the computed solution should be exploited, for instance by emitting a positive verdict whenever a property has been proved (as done by NBac), or by displaying the computed invariants in comments in the original program (as done by Interproc).

3 Abstract domains and their APIs: the examples of Apron and BddApron

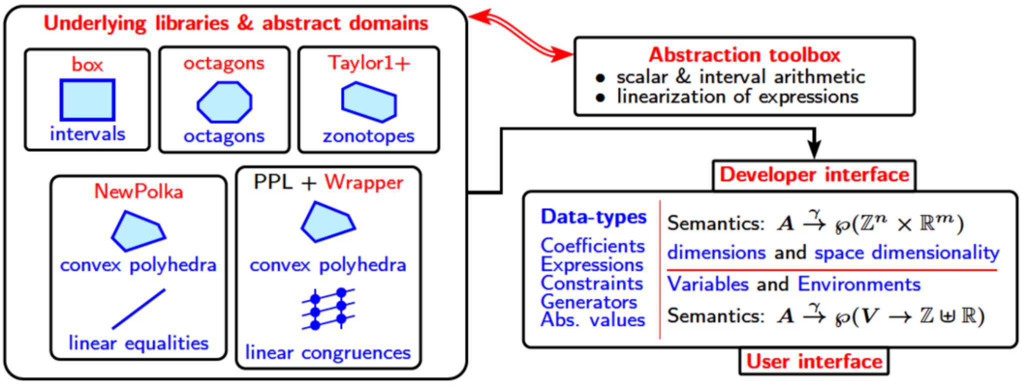

Underlying libraries & abstract domains
box
intervals
octagons
octagons
Taylor1+
zonotopes
NewPolka
convex polyhedra
linear equalities
PPL + Wrapper
convex polyhedra
linear congruences
Abstraction toolbox
• scalar & interval arithmetic
• linearization of expressions
Developer interface
Data-types
Coefficients
Expressions
Constraints
Generators
Abs. values
Semantics: $A \xrightarrow{\gamma} \wp(\mathbb{Z}^n \times \mathbb{R}^m)$
dimensions and space dimensionality
Variables and Environments
Semantics: $A \xrightarrow{\gamma} \wp(V \to \mathbb{Z} \uplus \mathbb{R})$
User interface

Fig. 5. Architecture of the APRON library.

As explained in the previous sections, the abstract domain component is in charge of interpreting in the abstract domain the operators and functions appearing in the concrete fixpoint equations, see the table besides. We first focus on abstract domains representing invariants on scalar variables, before discussing the case of those dedicated to shape analysis.

Constructors	$\bot$, $\top$, ...
Set operations	$\cap$, $\cup$, widening (∇)
Prog. lang. instructions	assignments of a variable by an expression, conditionals, ...
Change of state-space	adding & removing dimensions (scoping rules in prog. lang.)
Tests	emptiness & inclusion tests, ...
...	

Typical operations provided by an abstract domain

3.1 The APRON *library dedicated to numerical properties*

The main motivation for the implementation of the APRON library [24] was the observation that at that time (2004) many abstract domains for numerical variables were available, but none of them was providing the same functionalities nor sharing the same datatypes (for identical notions such as expressions, constraints, ...). The goal was thus precisely to provide a generic API for the abstraction of numerical variables, that would enable the effortless replacement in an analyzer of an abstract domain (e.g., octagons [28]) by another one (e.g., linear congruences [14]). The current version implements the API itself and provides several underlying libraries as depicted on Fig. 5.

The challenge in such cases is to satisfy the specific needs of the underlying abstract domains, while being uniform and high-level for the client analyzer. A secondary requirement was to minimize the effort for connecting an existing domain to the APRON interface. We think that this challenge was successfully addressed for the following reasons:

(1) The concrete semantics to be abstracted by abstract libraries was defined carefully from scratch and implemented by *datatypes that are independent of abstract libraries*. Moreover, it was chosen to be *very general*: it includes for instance non-linear and floating-point expressions and constraints, casts between numerical types. Abstract libraries are then free to abstract this rich semantics in a sound (and possibly imprecise) way, possibly using the provided toolbox to

minimize the implementation effort.

(2) Genericity was obtained by a kind of object orientation, in which abstract values are opaque datatypes and the effective underlying domain is controlled by a context manager. The domain-dependent code is thus located in the allocation of the manager.

It is instructive to compare with the approach followed by the PPL library [2] which progressively supported new abstract domains and extended the concrete semantics, but without unifying it (for instance, linear and congruence constraint systems are still not unified with a single notion), and keeping it less general (no support for non-linear expressions).

We now discuss the three evaluation criteria defined in the introduction.

New input language: our experience is that the implementation of the (ConCur)Interproc analyzer for the (C)Simple imperative language (see Fig. 2) *after* the implementation of Apron did not require a modification of the Apron API. This is because the concrete semantics implemented by Apron is quite rich. However, Apron does not support trigonometric nor exponential functions (partly because the IEEE754 standard does not cover these functions). This means that if one adds such functions in the input language, it is up to the front-end of the analyzer *and not to the abstract domain* to provide a sound abstraction of such operators in term of Apron operators. For instance the expression `x+cos(y)` can be safely abstracted by the valid Apron expression `x+[-1,1]`.

New datatype: this question is relevant here only for a new *numerical* datatype. Roughly speaking, Apron API provides operators that can implement all the numerical datatypes available in current prog. languages, *provided* that there is no overflow nor special numbers (such as floating-point `NaN`). This is because Apron implements directly only unbounded integers and reals, and emulates other datatypes on top of them. For instance, the C code `unsigned long x; x=x+1` is handled correctly (but not very precisely) if `x+1` is translated to the non-linear Apron expression `x+1 mod 2^32`, but not correctly if one translates it to the linear expression `x+1` and if `x` may be equal to `ULONG_MAX`. It is an on-going project to support more directly in the Apron toolbox bounded numerical types, but the right way to do it is not yet clear, and it involves precision/efficiency tradeoffs

New abstract domain: in this context, this means a new numerical abstract domain. Our experience is that we could connect without problem the external PPL library [2] to Apron, and that the more innovative Taylor1+ domain [11] was also connected by an external team without requiring a single change in the API. The connection of the MaxPlus polyhedra library [1] would require the addition of `max` and `min` operators in Apron expressions, and their treatment in the Apron toolbox. From the client side, our Interproc analyzer needed just a modification of a few initialization lines to enable the analysis of Simple programs with those new domains.

However some numerical domains does not fit so easily in our general framework. This is the case of an analysis based on *template polyhedra* [31], in which the analyzer must first be provided the template expressions associated to each control point before the analysis starts to infer bounds on these template expressions. Not only APRON does not provide a mean to specify these templates, but also the frontend and the solver components should take care of them, so supporting templates in INTERPROC would require significant modifications, that would be moreover specific to this abstract domain.

A benefit of the APRON approach is also the ability of building new domains on top of it, as illustrated in the next section.

3.2 The BDDAPRON *library dedicated to logico-numerical properties*

The aim of the BDDAPRON library [17] is to extend APRON with finite types, such as Booleans, enumerated types, and bounded integers encoded with N bits. For this project we kept the same principles learned from the APRON project.

(1) We defined first the concrete semantics to be supported. We added the new datatypes and related expressions. BDDAPRON supports any well-typed combination of APRON and finite-type expressions, such as

```
z+(if e=label or b and x+y*y>=2 then x+2*y else x).
```

Internally, expressions are represented with normalized BDDs/MTBDDs. Compared to APRON, BDDAPRON do not need any more the notion of constraints, which are replaced by general Boolean expressions. Last, the operations supported by BDDAPRON are exactly the same as those supported by APRON.

(2) We reused the same principle of a context manager, which contains here the underlying APRON manager.

Concerning the abstract domain itself, the domains implemented in BDDAPRON are based on the distinction between (i) the abstraction of each datatype (Booleans, numerics), and (ii) the coupling of the different datatypes. The implemented choices are to perform no abstraction for the purely finite part, to use an APRON domain for the numerical part, and to consider the abstract domain

$$\gamma : \overbrace{(\mathbb{B}^p \to A^{\mathcal{N}})}^{A} \longrightarrow \big(\mathbb{B}^{|} \to \wp(\mathbb{Z}^q \times \mathbb{R}^r)\big) \simeq \wp(\mathit{Var} \to \overbrace{\mathbb{B} \uplus \mathbb{E} \uplus \mathbb{I}}^{\text{finite types}} \uplus \overbrace{\mathbb{Z} \uplus \mathbb{R}}^{\text{numerics}})$$

which provides a precise coupling from finite types to numerical types. Internally, abstract values are efficiently represented by MTBDDs with APRON abstract values at their leaves.

We discuss again the three criteria defined in the introduction.

New input language: similarly as for APRON, as far as the sub-language of expressions in the input language is supported by BDDAPRON, a new input language has no impact on the abstract domain component.

New datatype: a new numerical datatype would concern mainly the underlying APRON API, and its impact on BDDAPRON would consists in delegating their effective treatment to APRON. On the other hand, BDDAPRON covers all known *scalar* finite types.[3]

[3] but not sum types available in OCaml, that may be finite but that are structured types

New abstract domain: we think that the concrete semantics is general enough to support any new abstract domain for logico-numerical properties.

3.3 Discussion

Our conclusion is that the modular approach and the requirement for an uniform API followed by the APRON and BDDAPRON projects induces very minor limitations and brings strong benefits. An explanation is that they address relatively known problems:

(i) There is a consensus on the concrete semantics to be provided. Even if the semantics of floating-point numbers is subtle, it is covered by the IEEE754 standard.

(ii) Abstract domains for numerical variables were available for many years, and there were also several experiences on the combination of Boolean and numerical properties [23,20,5,32]

The question is whether one could successfully extend this approach to more complex, non-scalar datatypes, for instance the "memory heap" datatype considered in shape analysis ? For instance, is it possible to design an uniform API for (a) the 3-valued logical structures abstract domain used by the TVLA tool [30], and (b) the inductive separation logic-based abstract domain used by the XISA tool [6] ? The challenge is related to the above-mentioned points:

(i) The concrete semantics is much more complex (see for instance [26]), and it is not independent of the prog. languages. For instance should it include C pointer arithmetic or not ?

(ii) The shape abstract domains are more complex and they need more inputs from the user. Whereas logico-numerical abstract domains *infer* properties with possibly no hint from the user (as in §2 or in the ASTREE analyzer [9]), shape domains always require the user to provide the data-structures invariants to guide the abstraction:
   - TVLA uses the notion of *instrumentation predicates* for this purpose of tuning the abstraction and preserving the important information;
   - XISA uses the notion of *inductive checkers*.

   To our knowledge, no shape analysis is currently able to *discover* automatically the invariant of the black-red tree data-structure from the type definition and the associated operations, although some of them (like XISA) are able to check the right invariant *if provided by the user* [6].

   Thus an uniform API would at least require a standard way to specify such invariants. But it appears difficult to convert an invariant specified with TVLA instrumentation predicates (based on first-order logic with transitive logic) to a checker (which is a formula in inductive separation logic); we conjecture that the expressivity of the two logics is incomparable.

Clarifying the point (i) and classifying the options chosen by existing tools would be beneficial any way for research in shape analysis.

In the middle way, we are currently working on the analysis of concurrent and recursive programs *with pointers*, which requires to analyze aliasing properties. We are optimistic in the possibility of packaging the corresponding semantic operations in an abstract domain on top of BddApron, partly because we do not encounter the point (ii) above.

4 The solver component

4.1 Design of the generic Fixpoint solver

As explained in §2, in our framework the solver component is in charge of solving iteratively the equation system produced by the front-end of the analyzer, relying on the abstract domain (and possibly the front-end) component(s) for computing the operations on the abstract properties. This means finding good iteration strategies and applying widening in good way (see [4,13]), while optimizing time and memory consumption (worklists algorithms, ...).

However, besides algorithmical issues listed above, it is important to clarify the type of equations it should support (this is the equivalent of the concrete semantics supported by an abstract domain). We distinguish three typical contexts.

Intraprocedural analysis (e.g., NBac): this is typically the case considered in textbooks and it is illustrated in §2. Such programs generates equations of the form $X_k = F_{1,k}(X_1) \cup \ldots \cup F_{n,k}(X_n)$, that is, a variable depends on an union of functions taking as input a *single* variable. Such equations can be modeled by a graph (CFG) in which nodes represent variables and oriented edges functions.

Interprocedural analysis (e.g., Interproc): such analysis techniques generate more complex equations, in which functions modeling procedure returns are applied to two variables, typically:

$$X^{caller}_{returnpoint} = F(X^{caller}_{callpoint}, X^{callee}_{exitpoint})$$

Intuitively at the return-point, the values of most local variables depends on their values at call-point, whereas the values of global variables depends on their values at the exit-point of the callee.

Analysis of concurrent programs (e.g., ConcurInterproc): they induce a specific algorithmic problem, if an interleaving semantics is considered. It is well-known in model-checking that the product of automata should be explored on-the-fly. The same phenomenon arise here: if one has a fixed number of threads with their CFG, building first the product CFG and corresponding equation system before starting the analysis is not efficient, as the fixpoint analysis is likely to discover that most variables of the equation system have an empty solution, because of synchronisations arising between the threads, which depends on the possible values of variables.

To fulfill these requirements, the Fixpoint solver uses first *hypergraphs* as a general model for equations, in which nodes represent variables and oriented hyperedges representing functions with possibly several predecessors.

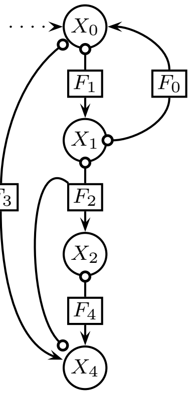
It then accepts as input a “manager” (provided by the front-end component) for interpreting semantic functions on the effective abstract domain, and an hypergraph, defined either explicitly or implicitly with an exploration function that allows it to dynamically explore its non-empty part. Last, it implements several techniques published in the literature:

- the principles of [4] for iteration strategies and widening application, combined with a worklist algorithm;
- optionally, the guided iteration technique of [13], built on top of the standard technique above.

When the hypergraph is defined implicitly by an exploration function, the analysis follows the same principle as the guided iteration principle: it alternates single propagation phases that discovers newly reachable variables/nodes, and full analysis phases on the subgraph so far explored.

```
     proc fact(n:int)
     returns (r:int)
(0)    if n>=2 then
(1)      r = n*fact(n-1)
(2)    else
r = 1
(3a)   endif
(3) end
```

$$\begin{aligned} X_0 &= \ldots \cup F_0(X_1) \\ X_1 &= F_1(X_0) \\ X_2 &= F_2(X_1, X_3) \\ X_3 &= \underbrace{F_3(X_0)}_{X_{3a}} \sqcup F_4(X_2) \end{aligned}$$



Hypergraphs representing equation systems

We come back to the three criteria defined in the introduction.

New input language: the Fixpoint solver fit the need of our three analyzers of Fig. 2, although they addresses different analysis challenges: partition refinement for tuning the precision/efficiency tradeoff (NBac), interprocedural analysis (Interproc), concurrent analysis (ConcurInterproc).

New datatype and/or new abstract domain: the first version of Interproc was based on Apron, and was later connected to BddApron with no change at all inside Fixpoint. We are convinced that the way we parametrize the solver by a manager in charge of interpreting semantic functions in the abstract domain makes it insensitive to the introduction of new datatypes, or more generally a new concrete semantics, or a new domain. In particular, the solver is not constrained by a specific expression language for the functions appearing in the equation system. The front-end/supervisor needs of course some changes, which are very minor besides the extension of the expression sub-language.

Our experience is thus that the solver component is the most generic component of our framework, *as long as* one focuses on iterative solving techniques. The advantage of isolating this component is that any improvement of it (such as [13] or [3]) can benefit to all the analyzers using it.

4.2 Alternative solving approaches

Some alternative solving technique were more recently explored.

• Reduction to an optimization problem: in some cases (*e.g.* interval analysis) equations $\boldsymbol{X} = \boldsymbol{F}(\boldsymbol{X})$ on abstract values can be transformed into equations $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$ on vectors of numbers of known size, and the least solution of $\boldsymbol{X} = \boldsymbol{F}(\boldsymbol{X})$ can be selected in $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$ by minimizing some expression.

- The policy iteration technique [8], which is applicable in the same cases, consists in (i) simplifying an equation $\boldsymbol{X} = \boldsymbol{F}(\boldsymbol{X})$ into $\boldsymbol{Y} = \boldsymbol{F}_\pi(\boldsymbol{Y})$ with the help of a *policy* π, (ii) solving $\boldsymbol{Y} = \boldsymbol{F}_\pi(\boldsymbol{Y})$ in a suitable way; (iii) and testing whether the obtained solution $\boldsymbol{Y}^{\boldsymbol{\infty}}$ is also a solution of $\boldsymbol{F}$. If it is the case, the analysis ends, otherwise $\boldsymbol{Y}^{\boldsymbol{\infty}}$ provide some hints to define a better policy π', and the process is iterated. For instance, in numerical equations of the form $x_k = \min_i(\boldsymbol{a}_i\boldsymbol{x}_i + b_i)$, a policy π typically consists in removing the min operator by selecting one of its argument.

The main advantage of these two alternative approaches is an improved precision, because widening is not required any more. However, they are applicable to equations on abstract values that can be reduced to equations on numbers, such as template polyhedra, and not to convex polyhedra that may have an unknown number of constraints. So they are inherently less general than the iterative approach. Another drawback is that they complicate a lot (at best) the composition of abstract domains, such as the one performed by BddApron.

Besides, the optimization approach follows avery different path compared to our framework: the abstract domain component is transformed into a translator from semantic equations to an optimization problem, and the fixpoint solver becomes a numerical solver. We are more optimistic in the possibiliy to exploit the policy iteration technique in our architecture.

4.3 The Iterator-based Approach

An alternative approach to generating semantic equations and then solving them is the *iterator* approach, which takes as input the higher-level Abstract Syntax Tree of the program and interprets directly the semantics of compound commands. This technique is described in [29] and is used in the large-scale ASTREE [9] and FLUCTUAT [10] analyzers. Among the advantages of this approach: (i) it makes easy the implementation of techniques like on-the-fly loop unrolling or procedure inlining (whereas our framework requires this process to be done before the analysis); (ii) it also enables without effort an efficient memory management (abstract values are not stored in a global graph, but in local variables of recursive procedures, so a garbage collector can free them automatically). However it presents also strong drawbacks. It combines our front-end and solver components and it is inherently less modular in this respect. Handling gotos (especially backward goto) is complicated, and more generally it is not suited to the unstructured systems handled by NBac (because of partitioning) or ConcurInterproc (because of the interleaving of threads). Combining forward and backward analyses is also made difficult.

In conclusion this approach enable more efficient implementations, but on less general equation systems.

5 The frontends

We already discussed the impact of a new datatype or abstract domain on the front-end component of the analyzer. In addition, a new input language has obviously a strong impact on it. The discussion about modularity should be considered here from a different angle.

The process of transforming an integrated analyzer into a modular analyzer can be seen as delegating as many tasks as possible to the solver and abstract domain components. For instance, the integration of the floating-point semantics inside Apron frees the client analyzer of taking care of it. So the question is: what is left to the analyzer front-end, besides its task of parsing the input program and supervising the full process ? In the case of the three analyzers of Fig. 2, the answer is not unique:

- For NBac, the frontend also manages the automatic partition refinement process, which is a complicated task.
- For Interproc and ConcurInterproc, the frontend is also in charge of the *control abstraction.* Indeed, the interprocedural (resp. concurrent) analysis technique used in the (Concur)Interproc analyzers can be derived as an abstraction of the operational semantics [25,21], which used call-stacks, into a more simple semantics which does not require the concept of call-stack and which can be expressed in the BddApron concrete semantics.

So in this respect the front-end component also perform abstractions, but these are too specialized to be incorporated inside an abstract domain with a well-defined API.

6 Conclusion and Related Work

Our experience is that in the modular architecture we promote for static analyzers the most modular component is the fixpoint solver component, followed by the abstract domain component, whereas modularity does not really make sense for the front-end part of the analyzer. The good practice is to try to push as much things as possible into the two first components to minimize the duplication of similar code in the front-end of analyzers. Designing and implementing such modular components is demanding in term of efforts, but we think it definitively worth it. In the case of abstract domains, it enables effortless comparisons between them, as shown by the Interproc analyzer which was implemented to illustrate the Apron library and which has been exploited since by several research papers.

The risk of a modular design is the increased complexity of the APIs. We think that we avoided successfully this risk for the Fixpoint solver, which has been used by several external users (for instance [12]). Concerning Apron, the flexibility of the coefficient and linear expression sub-language for instance induces some burden for the user. About this issue, we point out that the use of a high-level language featuring higher-order functions and polymorphic typing (namely OCaml) was greatly responsible for the quality of the API.

Among our future projects, there is the addition of machine integer types inside Apron (see § 3.1 and the design of a shape abstract domain following the principles we defend in §3.3.

The most mature work related to ours is the PAG tool [27], which makes different architectural choices compared to us. First, it integrates our solver, abstract domain and part of front-end components in a single package. Second, it provides a specialized specification language for defining abstract domain datatypes and transfer functions.

Its solver offers a large range of iteration strategies, but it has more limitations than ours. It supports interprocedural analysis, but only using the call-string approach or a relational approach in which procedure summaries are represented with explicit tables, which restricts its applicability to finite lattices (this excludes for instance convex polyhedra). The fundamental reason of this limitation is that PAG manipulates standard graphs instead of hypergraphs like the FIXPOINT solver (see §4).

PAG with its integrated approach is certainly more user-friendly with its elegant specification language for building abstract domains. On the other hand, it is less modular: it is unlikely than an abstract domain specified in PAG can be reused outside PAG. Conversely, implementing in PAG a domain requiring an external multi-precision arithmetic library or a BDD library (*e.g.,* convex polyhedra or BDDAPRON) is certainly complicated, if ever possible.

We thus think that PAG is better suited to analyses based on abstract domains with small to middle complexity, which do not depend on external libraries.

References

[1] X. Allamigeon, S. Gaubert, and E. Goubault. Inferring Min and Max Invariants Using Max-plus Polyhedra. In *Static Analysis Symposium, SAS'08*, volume 5079 of *LNCS*, 2008.

[2] R. Bagnara, P. M. Hill, and E. Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming (SCP)*, 72(1-2):3–21, 2008.

[3] G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Refining the control structure of loops using static analysis. In *Embedded software, EMSOFT'09.* ACM, 2009.

[4] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *International Conference on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, 1993.

[5] T. Bultan, R. Gerber, and C. League. Composite model-checking: verification with type-specific symbolic representations. *ACM Trans. on Softw. Eng. Methodol.*, 9(1), 2000.

[6] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *Principles of Programming Languages, POPL'08.* ACM, 2008.

[7] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Prog. language design and implementation.* ACM Press, 1990.

[8] A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *Computer Aided Verification, CAV'05*, volume 3576 of *LNCS*, 2005.

[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3), 2009.

[10] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems, FMICS'09*, volume 5825 of *LNCS*, 2009.

[11] K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain taylor1+. In *Computer Aided Verification, CAV'09*, volume 5643 of *LNCS*, 2009.

[12] L. Gonnord. The ASPIC tool: Accelerated Symbolic Polyhedral Invariant Computation. http://laure.gonnord.org/pro/aspic/aspic.html.

[13] D. Gopan and T. W. Reps. Guided static analysis. In *Static Analysis Symposium, SAS'07*, volume 4634 of *LNCS*, Aug. 2007.

[14] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493 of *LNCS*, 1991.

[15] N. Halbwachs, Y. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2), Aug. 1997.

[16] B. Jeannet. NBAC tool. http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/nbac/.

[17] B. Jeannet. The BDDAPRON logico-numerical abstract domains library. http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/bddapron/.

[18] B. Jeannet. The CONCURINTERPROC interprocedural analyzer for concurrent programs. http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi.

[19] B. Jeannet. The FIXPOINT equation solver. http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/fixpoint/.

[20] B. Jeannet. Representing and approximating transfer functions in abstract interpretation of hetereogeneous datatypes. In *Static Analysis Symposium, SAS'02*, volume 2477 of *LNCS*, Madrid (Spain)), Sept. 2002.

[21] B. Jeannet. Relational interprocedural verification of concurrent programs. In *Software Engineering and Formal Methods, SEFM'09*. IEEE, Nov. 2009.

[22] B. Jeannet, M. Argoud, and G. Lalire. The INTERPROC interprocedural analyzer. http://pop-art.inrialpes.fr/interproc/interprocweb.cgi.

[23] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99*, volume 1694 of *LNCS*, 1999.

[24] B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, CAV'2009*, volume 5643 of *LNCS*, pages 661–667, 2009. http://apron.cri.ensmp.fr/library/.

[25] B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *Int. Conf. on Algebraic Methodology and Software Technology, AMAST'04*, volume 3116 of *LNCS*, 2004.

[26] X. Leroy and S. Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[27] F. Martin. PAG – an efficient program analyzer generator. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2(1):46–67, 1998.

[28] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1), 2006.

[29] T. C. D. of a Generic Abstract Interpreter. Patrick cousot. In *Calculational System Design*. 1998. http://www.di.ens.fr/~cousot/COUSOTpapers/Marktoberdorf98.shtml.

[30] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Prog. Languages and Systems*, 24(3), 2002.

[31] S. Sankaranarayanan, T. Dang, and F. Ivancic. Symbolic model checking of hybrid systems using template polyhedra. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08*, volume 4963 of *LNCS*, 2008.

[32] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01*, volume 2031 of *LNCS*, 2001.