Compositional Separate Modular Static Analysis of Programs by Abstract Interpretation

Patrick Cousot and Radhia Cousot

D partement d'informatique
cole normale sup rieure
45 rue d'Ulm
75230 Paris cedex 05, France
Patrick.Cousot@ens.fr
http://www.di.ens.fr/~cousot

Laboratoire d'informatique
cole polytechnique
91128 Palaiseau cedex, France
Radhia.Cousot@polytechnique.fr
http://lix.polytechnique.fr/~rcousot

Abstract— **The purpose of this paper is to present four basic methods for compositional separate modular static analysis of programs by abstract interpretation:**
- **Simplification-based separate analysis;**
- **Worst-case separate analysis;**
- **Separate analysis with (user-provided) interfaces;**
- **Symbolic relational separate analysis;**

as well as a fifth category which is essentially obtained by composition of the above separate local analyses and global analysis methods.

Keywords— **Abstract interpretation, Static program analysis, Global analysis, Whole program analysis, Separate analysis, Modular analysis, Component analysis, Composition of analyses.**

I. Introduction

Static program analysis is the automatic compile-time determination of run-time properties of programs. This is used in many applications from optimizing compilers, to abstract debuggers and semantics based program manipulation tools (such as partial evaluators, error detection and program understanding tools). This problem is undecidable so that program analyzers involve some safe approximations formalized by abstract interpretation of the programming language semantics. In practice, these approximations are chosen to offer the best trade-off between the precision of the information extracted from the program and the efficiency of the algorithms to compute this information from the program text.

Abstract interpretation based static program analysis is now in an industrialization phase and several companies have developed static analyzers for the analysis of software or hardware either for their internal use or to provide new software analysis tools to end-users, in particular for the compile-time detection of run-time errors in embedded applications (which should be used before the application is launched). Important characteristics of these analyzers is that all possible run-time errors are considered at compilation time, without code instrumentation nor user interaction (as opposed to debugging for example). Because of foundational undecidability problems, not all errors can be statically classified as certain or impossible and a small percentage remains as potential errors for which the analysis is inconclusive. In most commercial software, with low correctness requirements, the analysis will reveal many previously uncaught certain errors so that the percentage of potential errors for which the analysis is inconclusive is not a practical problem as long as all certain errors have been corrected and these corrections do not introduce new certain errors . However, for safety critical software, it is usually not acceptable to remain inconclusive on these few remaining potential errors [1]. One solution is therefore to improve the precision of the analysis. This is always theoretically possible, but usually at the expense of the time and memory cost of the program analyses, which can become prohibitive for very large programs.

The central idea is therefore that of *compositional separate static analysis of program parts* where very large programs are analyzed by analyzing parts (such as components, modules, classes, functions, procedures, methods, libraries, etc...) separately and then by composing the analyses of these program parts to get the required information on the whole program. Components can be analyzed with a high precision whenever they are chosen to be small enough. Since these separate analyzes are done on parts and not on the whole program, total memory consumption may be reduced, even with more precise analyzes of the parts. Since these separate analyzes can be performed in parallel on independent computers, the global program analysis time may also reduced.

II. Global Static Program Analysis

The formulation of global static program analysis in the abstract interpretation framework [1], [2], [3], [4], [5] consists in computing an approximation of a program seman-

This work was supported in part by the RTD project IST-1999-20527 DAEDALUS of the european IST FP5 programme.

[1] The number of residual potential errors, even if it is a low percentage of the possible errors (typically 5%), may be unacceptably large for very large programs.

tics expressing the properties of interest of the program P to be analyzed. The semantics can often be expressed as a least fixpoint $\mathcal{S}[\![\mathcal{P}]\!] = \mathrm{lfp}^{\sqsubseteq} F[\![P]\!]$ that is as the least solution to a monotonic system of equations $X = F[\![P]\!](X)$ computed on a poset $\langle D, \sqsubseteq\rangle$ where the semantic domain D is a set equipped with a partial ordering $\sqsubseteq$ with infimum $\perp$ and the endomorphism $F[\![P]\!] \in D \stackrel{\mathrm{m}}{\longmapsto} D$ is monotonic. The approximation is formalized through a Galois connection $\langle D, \sqsubseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{D}, \bar{\sqsubseteq}\rangle$ where a concrete program property $p \in D$ is approximated by any abstract program property $\bar{p} \in \bar{D}$ such that $p \sqsubseteq \gamma(\bar{p})$ and has a best/more precise abstraction $\alpha(p) \in \bar{D}$ (Other formalizations through closure operators, ideals, etc. are equivalent [1], [3]. The best abstraction hypothesis can also be relaxed [6]). Then global static program analysis consists in computing an abstract least fixpoint $\bar{\mathcal{S}}[\![P]\!] = \mathrm{lfp}^{\bar{\sqsubseteq}} \bar{F}[\![P]\!]$ which is a sound approximation of the concrete semantics in that $\mathrm{lfp}^{\sqsubseteq} F[\![P]\!] \sqsubseteq \gamma(\mathrm{lfp}^{\bar{\sqsubseteq}} \bar{F}[\![P]\!])$. This fixpoint soundness condition can be ensured by stronger local/functional soundness conditions such as $\bar{F}[\![P]\!]$ is monotonic and either $\alpha \circ F[\![P]\!] \mathrel{\bar{\sqsubseteq}} \bar{F}[\![P]\!] \circ \alpha$, $\alpha \circ F[\![P]\!] \circ \gamma \mathrel{\bar{\sqsubseteq}} \bar{F}[\![P]\!]$ or equivalently $F[\![P]\!] \circ \gamma \sqsubseteq \gamma \circ \bar{F}[\![P]\!]$ (see [3]). The least fixpoint is computed as the limit of the iterates $F^0 = \bar{\perp}$, …, $F^{n+1} = \bar{F}[\![P]\!](F^n)$, … where $\bar{\perp}$ is the infimum of the abstract domain $\bar{D}$. Convergence of the iterates can always be enforced using widening/narrowing techniques [2]. The result is correct but less precise than the limit $\bar{\mathcal{S}}[\![P]\!] = \bar{\bigsqcup}_{n\geq 0} F^n$ where $\bar{\sqcup}$ is the least upper bound (which does exist if the abstract domain $\bar{D}$ is a cpo, complete lattice, etc.) [2], [6].

For example, the reachability analysis of the following program with the interval abstract domain [2]:

```
0: x := 1;
1: while (x < 1000) do
2:   x := (x + 1)
3: od
4:
```

consists in solving the following system of fixpoint equations $\langle$`X0, X1, X2, X3, X4`$\rangle = \bar{F}[\![P]\!](\langle$`X0, X1, X2, X3, X4`$\rangle)$ [7] where `Xi` is the abstract environment associated to program point `i` $=$ `0`, …, `4`, each environment `Xi` maps program variables (here `x`) to a description of their possible values at run-time (here an interval), `U` is the union of abstract environments and `_O_` denotes the singleton consisting of the undefined initial value:

```
X0 = init(x, _O_)
X1 = assign[|x, 1|](X0) U X3
X2 = assert[|x < 1000|](X1)
X3 = assign[|x, (x + 1)|](X2)
X4 = assert[|x >= 1000|](X1)
```

The least solution to this system of equations is then approximated iteratively using widening/narrowing iterative convergence acceleration methods [2] as follows:

```
X0 = { x:_O_ }
X1 = { x:[1,1000] }
X2 = { x:[1,999] }
X3 = { x:[2,1000] }
X4 = { x:[1000,1000] }
```

Some static program analysis methods (such as typing [8] or set based-analysis [9]) consist in solving constraints (also called verification conditions, etc.), but this is equivalent to iterative fixpoint computation [10], maybe with widening [9] (since the least solution to the constraints $F[\![P]\!](X) \sqsubseteq X$ is the same as the least solution $\mathrm{lfp}^{\sqsubseteq} F[\![P]\!]$ of the equations $X = F[\![P]\!](X)$).

Such fixpoint computations constitute the basic steps of program analysis (e.g. for forward reachability analysis, backward ancestry analysis, etc.). More complex analyzes are obtained by combining these basic fixpoint computations (see e.g. [1], [11] or [12], [13] for abstract testing of temporal properties of program).

The type of static whole-program analysis methods that we have briefly described above is *global* in that the whole program text is needed to establish the system of equations and this system of equations is solved iteratively at once.

In practice, chaotic iteration strategies [14], [15] can be used to iterate successively on components of the system of equations (as determined by a topological ordering of the dependency graph of the system of equations). However, in the worst case, the chaotic iteration may remain global, on all equations, which may be both memory consuming (since the program hence the system of equations can be very large) and time consuming (in particular when convergence of the iterates is slow).

This problem can be solved by using less precise analyzes but this may simply lead to analyzes which are both imprecise and quite costly. Moreover, the whole program must be reanalyzed even if a small part only is modified. Hence the necessity to look for *local* methods for the static analysis of programs piecewise.

III. Separate Modular Static Program Analysis

A. *Formalization of Separate Modular Static Program Analysis in the Abstract Interpretation Framework*

In general programs $P[P_1, \ldots, P_n]$ are made up from parts $P_1, \ldots, P_n$ such as functions, procedures, modules, classes, components, libraries, etc. so that the semantics $S[\![P]\!]$ of the whole program P is obtained compositionally from the semantics of its parts $\mathrm{lfp}^{\sqsubseteq_i} F[\![P_i]\!]$, $i = 1, \ldots, n$ as follows:

$$S[\![P]\!] \quad = \quad \mathrm{lfp}^{\sqsubseteq} F[\![P]\!][\mathrm{lfp}^{\sqsubseteq_1} F[\![P_1]\!], \ldots, \mathrm{lfp}^{\sqsubseteq_n} F[\![P_n]\!]]$$

where $F[\![P_i]\!] \in D_i \stackrel{\mathrm{m}}{\longmapsto} D_i$, $i = 1, \ldots, n$ and $F[\![P]\!] \in (D_1 \times \ldots \times D_n) \stackrel{\mathrm{m}}{\longmapsto} (D \stackrel{\mathrm{m}}{\longmapsto} D)$ are (componentwise) monotonic (see e.g. [16]).

This compositional separate modular static analysis of the program $P[P_1, \ldots, P_n]$ is based on separate abstractions $\langle D_i, \sqsubseteq_i\rangle \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} \langle \bar{D}_i, \bar{\sqsubseteq}_i\rangle$, for each part P_i, $i =$

$1, \ldots, n$. The analysis of the parts then consists in computing separately an abstract information $\vec{A}_i \sqsupseteq_i \mathrm{lfp}^{\sqsubseteq_i} \bar{F}_i[\![P_i]\!]$, $i = 1, \ldots, n$ on each part P_i so that $\mathrm{lfp}^{\sqsubseteq_i} F[\![P_i]\!] \sqsubseteq_i \gamma_i(\vec{A}_i)$. Since the components P_i, $i = 1, \ldots, n$ are generally small, they can be analyzed with a high precision by choosing very precise abstractions $\langle D_i, \sqsubseteq_i\rangle \xrightleftharpoons[\alpha_i]{\gamma_i} \langle \bar{D}_i, \bar{\sqsubseteq}_i\rangle$, (see examples of precise abstract domains in e.g. [4], [5]). A typical example is the replacement of numerical interval analysis (using intervals $[a, b]$ where a and b are numerical constants) by a more precise symbolic interval analysis (using intervals $[L, H]$ where L and H are mathematical variables, which can be implemented through the octagonal abstract domain of [17]).

The global analysis of the program consists in composing the analyses $\vec{A}_i$, $i = 1, \ldots, n$ of these program parts P_i, $i = 1, \ldots, n$ to get the required information on the whole program by computing $\mathrm{lfp}^{\bar{\sqsubseteq}} \bar{F}[\![P]\!][\vec{A}_1, \ldots, \vec{A}_n]$.

Since these separate analyzes are done on parts and not on the whole program, total memory consumption may be reduced, even with more precise analyzes of the parts. Since the separate analyzes of the program parts can be performed in parallel on independent computers, the global program analysis time may also reduced. The global abstraction is composed from the abstractions of the program parts and has the form:

$$\langle D[D_1, \ldots, D_n], \sqsubseteq\rangle \xrightleftharpoons[\alpha[\alpha_1, \ldots, \alpha_n]]{\gamma[\gamma_1, \ldots, \gamma_n]} \langle \bar{D}[\bar{D}_1, \ldots, \bar{D}_n], \bar{\sqsubseteq}\rangle \ .$$

The local/functional soundness condition is:

$$\begin{array}{c}\alpha[\alpha_1, \ldots, \alpha_n](F[\![P]\!][\gamma_1(X_1), \ldots, \gamma_n(X_n)]) \quad \bar{\sqsubseteq} \\ \bar{F}[\![P]\!][X_1, \ldots, X_n]\end{array}$$

which implies that:

$$\begin{array}{c}\mathrm{lfp}^{\sqsubseteq} F[\![P]\!][\mathrm{lfp}^{\sqsubseteq_1} F[\![P_1]\!], \ldots, \mathrm{lfp}^{\sqsubseteq_n} F[\![P_n]\!]] \quad \sqsubseteq \\ \gamma[\gamma_1, \ldots, \gamma_n](\mathrm{lfp}^{\bar{\sqsubseteq}} \bar{F}[\![P]\!][\vec{A}_1, \ldots, \vec{A}_n]) \ .\end{array}$$

B. *Difficulty of Separate Static Program Analysis: Interference*

The theoretical situation that we have sketched above in Sec. III-A is ideal and sometimes very difficult to put into practice. This is because the parts $P_1, \ldots, P_n$ of the program P are not completely independent so that the separate analyses of the parts P_i are not independent of those of the other parts P_1, ..., P_{i-1}, P_{i+1}, ..., P_n and of that of the program P.

For example in an imperative program à la C, a function may call other functions in the program and use and/or modify global variables. In Pascal, a program may modify variables on the program execution stack at a program point where these variables are even not visible (see [18]).

A very simple formalization consists in considering that the semantics of the program can be specified in the following equational form:

$$\begin{cases} \vec{Y} &= F[\![P[P_1, \ldots, P_n]]\!]\langle \vec{Y}, \vec{X}_1, \ldots, \vec{X}_n\rangle \\ \vec{X}_i &= F[\![P_i]\!]\langle \vec{Y}, \vec{X}_1, \ldots, \vec{X}_n\rangle \\ i = 1, \ldots, n & \end{cases}$$

where $\vec{Y}$ represents the global information on the program while $\vec{X}_i$ represents that on the program part P_i, $i = 1, \ldots, n$. In general, the least solution is preferred for a componentwise ordering $\sqsubseteq \times \sqsubseteq_1 \times \ldots \times \sqsubseteq_n$ where $\langle D, \sqsubseteq\rangle$ and the $\langle D_1, \sqsubseteq_1\rangle$, ..., $\langle D_n, \sqsubseteq_n\rangle$ are the concrete domains (usually cpos, complete lattices, etc.) respectively expressing the properties of the program $P[P_1, \ldots, P_n]$ and its parts P_i, $i = 1, \ldots, n$.

In general the local properties $\vec{X}_i$ of the part P_i depend upon the knowledge of the local properties $\vec{X}_j$ of the other program parts P_j, $j \neq i$ and of the global properties $\vec{Y}$ of the program $P[P_1, \ldots, P_n]$. The properties $\vec{Y}$ of the program $P[P_1, \ldots, P_n]$ are also defined in fixpoint form so depend on themselves as well as on the local properties $\vec{X}_i$ of the part P_i, $i = 1, \ldots, n$. Usually, the abstraction yields to an abstract system of equations of the same form:

$$\begin{cases} \vec{Y} &= \bar{F}[\![P[P_1, \ldots, P_n]]\!]\langle \vec{Y}, \vec{X}_1, \ldots, \vec{X}_n\rangle \\ \vec{X}_i &= \bar{F}[\![P_i]\!]\langle \vec{Y}, \vec{X}_1, \ldots, \vec{X}_n\rangle \\ i = 1, \ldots, n & \end{cases}$$

on the abstract domains $\langle \bar{D}, \bar{\sqsubseteq}\rangle$ and the $\langle \bar{D}_1, \bar{\sqsubseteq}_1\rangle$, ..., $\langle \bar{D}_n, \bar{\sqsubseteq}_n\rangle$ with Galois connections $\langle D, \sqsubseteq\rangle \xrightleftharpoons[\alpha]{\gamma} \langle \bar{D}, \bar{\sqsubseteq}\rangle$ and $\langle D_i, \sqsubseteq_i\rangle \xrightleftharpoons[\alpha_i]{\gamma_i} \langle \bar{D}_i, \bar{\sqsubseteq}_i\rangle$, for all $i = 1, \ldots, n$.

Ideally, the separate analysis of program part P_i consists in computing a fixpoint:

$$\mathrm{lfp}^{\bar{\sqsubseteq}_i} \boldsymbol{\lambda} \vec{X}_i \cdot \bar{F}[\![P_i]\!]\langle \vec{Y}, \vec{X}_1, \ldots, \vec{X}_i, \ldots, \vec{X}_n\rangle$$

where the $\vec{Y}$, $\vec{X}_1$, ..., $\vec{X}_{i-1}$, $\vec{X}_{i+1}$, ..., $\vec{X}_n$ denote the abstract properties which are assumed/guaranteed on the objects of the program $P[P_1, \ldots, P_n]$ and its parts P_j, $j = 1, \ldots, i-1, i+1, \ldots, n$ which are external references within that part P_i (such as global variables of a procedure, external functions called within a module, etc.). The whole problem is to determine $\vec{Y}$, $\vec{X}_1$, ..., $\vec{X}_{i-1}$, $\vec{X}_{i+1}$, ..., $\vec{X}_n$ while analyzing program part P_i.

C. *Dependence Graph*

A classical technique (also used in separate compilation) consists in computing a dependence graph where a part P_i depends upon another part P_j, $i \neq j$ if and only if the analysis of P_i uses some information which is computed by the analysis of part P_j (formally P_i depends upon part P_j if and only if $\exists \vec{X}_j, \vec{X}'_j$: $F[\![P_i]\!]\langle \vec{Y}, \vec{X}_1, \ldots, \vec{X}_j, \ldots, \vec{X}_n\rangle \neq F[\![P_i]\!]\langle \vec{Y}, \vec{X}_1, \ldots, \vec{X}'_j, \ldots, \vec{X}_n\rangle$. It is often the case that this dependency graph is built before the analysis and parts are analyzed in sequence by their topological order (see e.g. [19], [20]). As in most incremental compilation systems, circular dependency may not be considered (i.e. all circularly dependent parts are grouped into a single part since

an iterative analysis is necessary). At the limit, the analysis will degenerate into a global analysis as considered in Sec. II, the dependence graph then corresponding to a particular chaotic iteration strategy [1], [16], [15]. Otherwise, the circularities must be broken using one of the compositional separate modular static program analysis methods considered in this paper:
– Simplification-based separate analysis;
– Worst-case separate analysis;
– Separate analysis with (user-provided) interfaces;
– Symbolic relational separate analysis;
or by a combined method which is essentially obtained by composition of the previous local ones and global analysis.

IV. Simplification-Based Separate Analysis

To start with, we consider ideas based upon the simplification of the equations to be solved. We do not consider here local simplifications of the equations (that is simplification of one equation independently of the others such as e.g. [21]) but global simplifications, where the simplification of one equation requires the examination of other equations. Since these systems of equations can be considered as functional programs, many program transformation and optimization techniques are directly applicable to do so such as algebraic simplification, constant propagation, partial evaluation [22], compilation, etc.

For each program part, the fixpoint transformer $\bar{F}[\![P_i]\!]$ (often expressed as a system of equations $X = \bar{F}[\![P_i]\!](X)$ or equivalently as constraints $\bar{F}[\![P_i]\!](X) \mathrel{\bar{\sqsubseteq}_i} X$, [10]) is simplified into $\bar{F}_s[\![P_i]\!]$. The global analysis of the program then consists in computing $\mathrm{lfp}^{\bar{\sqsubseteq}}\, \bar{F}[\![P]\!][\mathrm{lfp}^{\bar{\sqsubseteq}_1}\, \bar{F}_s[\![P_1]\!], \ldots, \mathrm{lfp}^{\bar{\sqsubseteq}_n}\, \bar{F}_s[\![P_n]\!]]$ so that the fixpoints for the parts are computed in a completely known context or environment.

Very often, $\bar{F}_s$ is obtained by abstract interpretation of $\bar{F}$. A frequently used variant of this idea consists in first using a preliminary global analysis of the whole program $P[P_1, \ldots, P_n]$ with a rough imprecise abstraction to collect some global information on the program in order to help in the simplification of the $\bar{F}[\![P_i]\!]$, designed with a more precise abstraction, into $\bar{F}_s[\![P_i]\!]$.

Examples of application of this simplification idea can be found in the analysis of procedures of [2, Sec. 4.2], in the *componential set-based analysis* of [23], in the *variable substitution transformation* of [24] and in the *summary optimization* of [25]. Another example is *abstract compilation* where the equations and fixpoint computation are compiled (often in the same language as the one to be analyzed so that program analysis amounts to the execution of an abstract compilation of program), see e.g. [26], [27], [28], [29], [30].

Since the local analysis phases of the program parts P_i, which consist in computing the fixpoints $\mathrm{lfp}^{\bar{\sqsubseteq}_i}\, \bar{F}[\![P_i]\!]$ are delayed until the global analysis phase, which consists in computing $\mathrm{lfp}^{\bar{\sqsubseteq}}\, \bar{F}[\![P]\!][\mathrm{lfp}^{\bar{\sqsubseteq}_1}\, \bar{F}_s[\![P_1]\!], \ldots, \mathrm{lfp}^{\bar{\sqsubseteq}_n}\, \bar{F}_s[\![P_n]\!]]$, not much time and memory resources are saved in this computation, even though the simplified fixpoint operators $\bar{F}_s[\![P_i]\!]$ are used in place of the original ones $\bar{F}[\![P_i]\!]$. The main reason is that the simplification often saves only a linear factor [2], which may be a negligible benefit when compared to the cost of the iterative fixpoint computation. In our opinion, this explains why this approach does not scale up for very large programs [31].

V. Worst-Case Separate Analysis

We have seen that the problem of separate analysis of a program part P_i consists in determining the properties $\vec{Y}, \vec{X}_1, \ldots, \vec{X}_{i-1}, \vec{X}_{i+1}, \ldots, \vec{X}_n$ of the external objects referenced in the program part P_i while computing the local fixpoint:

$$\mathrm{lfp}^{\bar{\sqsubseteq}_i}\, \boldsymbol{\lambda}\, \vec{X}_i \cdot F[\![P_i]\!]\langle \vec{Y}, \vec{X}_1, \ldots, \vec{X}_i, \ldots, \vec{X}_n\rangle$$

The worst-case separate analysis consists in considering that absolutely no information is known on the interfaces $\vec{Y}, \vec{X}_1, \ldots, \vec{X}_{i-1}, \vec{X}_{i+1}, \ldots, \vec{X}_n$. Traditionally in program analysis by abstract interpretation the top symbol $\bar{\top}$ is used to represent such an absence of information ($\bar{\top}$ is the supremum of the complete lattice $\bar{D}_i$ representing the abstract program properties ordered by the approximation ordering $\bar{\sqsubseteq}_i$ corresponding to the abstraction of the logical implication). The worst-case separate analysis therefore consists in first separately computing or effectively approximating the local abstract fixpoints:

$$\vec{A}_i \quad \mathrel{\bar{\sqsupseteq}_i} \quad \mathrm{lfp}^{\bar{\sqsubseteq}_i}\, \boldsymbol{\lambda}\, \vec{X}_i \cdot F[\![P_i]\!]\langle \bar{\top}, \bar{\top}, \ldots, \vec{X}_i, \ldots, \bar{\top}\rangle$$

for all program parts P_i. Then the global program analysis is:

$$\mathrm{lfp}^{\bar{\sqsubseteq}}\, \boldsymbol{\lambda}\, \vec{Y} \cdot \bar{F}[\![P]\!][\vec{Y}, \vec{A}_1, \ldots, \vec{A}_n]\ .$$

The main advantage of this approach is that all analyzes of the parts P_i, $i = 1, \ldots, n$ can be done in parallel. Moreover the modification of a program part requires only the analysis of that part to be redone before the global program analysis. This explains why the worst-case separate analysis is very efficient. However, because nothing is known about the interfaces of the parts with the program and with the other parts, this worst-case analysis is often too imprecise.

An example is the procedure analysis of [16, Sec. 4.2.1 & 4.2.2] where the effect of procedures (in particular the values of result/output parameters) are computed by a local analysis of the procedure assuming that the properties of value/input parameters is unknown in the main call (and a widening is used in recursive calls both to cope with possible non-termination of calls with identical parameters and with the possibility of having infinitely many calls with different parameters).

Another example is the escape analysis of *higher-order* functions by [32]. Escape analysis aims at determining which local objects of a procedure do not escape out of the call (so that they can be allocated on the stack, the

[2]Sometimes the simplification can save an exponential factor, see e.g. [21].

escaping object have to be allocated on the heap since their lifetime is longer than that of the procedure call). In this analysis, the higher-order functions which are passed as parameter to a procedure are assumed to be unknown, so that e.g. any call to such an unknown external higher-order function may have any possible side-effect.

Yet another example is the worst-case separate analysis of library modules in the points-to and side-effect analyses of [25].

A last example is the abstract interpretation-based analysis for automatically detecting all potential interactions between the agents of a part of a mobile system interacting with an unknown context [33].

As considered in Sec. IV, an improvement consists in using a preliminary global analysis of the whole program $P[P_1, \ldots, P_n]$ with a rough imprecise abstraction to collect some global information on the program in order to get information on the interface $\vec{Y}$, $\vec{X}_1$, ..., $\vec{X}_{i-1}$, $\vec{X}_{i+1}$, ..., $\vec{X}_n$ more precise than the unknown $\bar{\top}$. An example is the preliminary inexpensive whole-program points-to analysis made by [34] before their modular/fragment analysis.

VI. Separate Analysis with (User-Provided) Interfaces

The idea is to ask the user to provide information about the properties $\vec{Y}$, $\vec{X}_1$, ..., $\vec{X}_{i-1}$, $\vec{X}_{i+1}$, ..., $\vec{X}_n$ of the external objects referenced in the program part P_i while computing the local abstract fixpoints:

$$\mathrm{lfp}^{\sqsubseteq_i} \boldsymbol{\lambda} \vec{X}_i \cdot F[\![P_i]\!]\langle \vec{Y},\ \vec{X}_1,\ \ldots,\ \vec{X}_i,\ \ldots,\ \vec{X}_n\rangle\ .$$

$i = 1, \ldots, n$ as well as the global abstract fixpoint:

$$\mathrm{lfp}^{\sqsubseteq} \boldsymbol{\lambda} \vec{Y} \cdot F[\![P[P_1, \ldots, P_n]]\!]\langle \vec{Y},\ \vec{X}_1,\ \ldots,\ \vec{X}_n\rangle\ .$$

The information provided on the interface of the program part with the external world takes the form of:

- the *assumptions* $\vec{J}$ on the program and $\vec{I}_1$, ..., $\vec{I}_{i-1}$, $\vec{I}_{i+1}$, ..., $\vec{I}_n$ on the other program parts P_j, $i \neq j$ that can be made in the local analysis of the program part P_i. These assumptions will have to be guaranteed by the local analyzes of the other parts and the global analysis of the program when using this part P_i. These assumptions make possible the analysis of the program part P_i independently of the context in which that program part P_i is used (or more generally several possible contexts may be considered);
- the *guarantee* $\vec{I}_i$ on the program part P_i that must be established by the local analysis of that part P_i. The global program analysis and that of the other program parts will rely upon this guarantee when using that part P_i (considering only the possible behaviors of that part P_i which are relevant to its context of use).

Typically, the interface should be precise enough so that the assumptions (or preconditions) $\vec{J}$ on the program and $\vec{I}_1$, ..., $\vec{I}_{i-1}$, $\vec{I}_{i+1}$, ..., $\vec{I}_n$ are the weakest possible so that the analysis of a part P_i only requires the source code of that part P_i while the guarantee (or postcondition) $\vec{I}_i$ should be the strongest possible so that analyzes using that part P_i never need to access the source code of that part P_i.

Formally, the separate analysis with interfaces $\vec{J}$, $\vec{I}_1$, ..., $\vec{I}_n$ consists in computing or approximating the local abstract fixpoints:

$$\vec{A}_i \quad \bar{\sqsupseteq}_i \quad \mathrm{lfp}^{\bar{\sqsubseteq}_i} \boldsymbol{\lambda} \vec{X}_i \cdot F[\![P_i]\!]\langle \vec{J},\ \vec{I}_1,\ \ldots,\ \vec{X}_i,\ \ldots,\ \vec{I}_n\rangle\ .$$

One must also check that one can rely upon the assumptions $\vec{J}$, $\vec{I}_1$, ..., $\vec{I}_{i-1}$, $\vec{I}_{i+1}$, ..., $\vec{I}_n$ made during the analysis on the program part P_i by verifying that it is guaranteed by the analysis of the other parts P_j, $j \neq i$ in that:

$$\forall i = 1, \ldots n: \quad \vec{A}_i \quad \bar{\sqsubseteq}_i \quad \vec{I}_i$$

as well as for the global assumption $\vec{J}$ on the program that should be guaranteed by the global program analysis:

$$\vec{A} \quad \bar{\sqsupseteq} \quad \mathrm{lfp}^{\bar{\sqsubseteq}} \boldsymbol{\lambda} \vec{Y} \cdot F[\![P[P_1, \ldots, P_n]]\!]\langle \vec{Y},\ \vec{I}_1,\ \ldots,\ \vec{I}_i,\ \ldots,\ \vec{I}_n\rangle,$$

in that:

$$\vec{A} \quad \bar{\sqsubseteq} \quad \vec{J}\ .$$

This technique is classical in program typing (e.g. user specified number, passing mode and type of parameters of procedures which are assumed in the type checking of the procedure body and must be guaranteed at each procedure call) and in program verification (see e.g. the rely/guarantee specifications of [35]). Examples of user-provided interfaces in static program analysis are the control-flow analysis of [36] and the notion of *summary information* of [25], [37].

A particular case is when no assumption is made on the interface of each program part with its external environment so that the automatic generation of the properties guaranteed by the program part essentially amounts to the worst-case analysis of Sec. V or its variants.

Instead of asking the user to provide the interface, this interface can sometimes be generated automatically. For example, a backward analysis of absence of run-time errors or exceptions (such as the backward analysis using greatest fixpoints introduced in [38]) or any other ancestry analysis (e.g. to compute necessary termination conditions [38] or success conditions for logic programs [39]) can be used to automatically determine conditions on the interface which have to be assumed to ensure that the program part P_i is correctly used in the whole program $P[P_1, \ldots, P_n]$. A forward reachability analysis will provide information on what can be guaranteed on the interface of the program part P_i with its environment, that is the other parts P_j, $j \neq i$ and the program P. A refinement is to combine the forward and backward analyses [11], [12], [13].

As considered in Sec. IV and Sec. V, an improvement consists in using a preliminary fast global analysis of the whole program $P[P_1, \ldots, P_n]$ with a rough imprecise abstraction to collect some global information on the program in order to get information on what is guaranteed on the interfaces $\vec{J}$, $\vec{I}_1$, ..., $\vec{I}_n$.

Moreover simplification techniques, as considered in Sec. IV can be applied to simplify the automatically synthesized or user-provided interface.

VII. Symbolic Relational Separate Analysis

To start with, we consider a powerful but not well-known compositional separate modular static program analysis method that we first introduced in [16]. Symbolic relational separate analysis is based on the use of relational abstract domains and a relational semantics of the program parts (see e.g. [40]). The idea is to analyze a program part P_i separately by giving symboloc names to all external objects used or modified in that part P_i. The analysis of the part consists in relating symbolically the local information within the part P_i to the external objects through these names. External actions have to be handled in a lazy way and their possible effects on internal objects must be delayed[3] (unless the effect of these actions is already known thanks to a previous static analysis, see Sec. III-C). When the part is used, the information about the part is obtained by binding the external names to the actual values or objects that they denote and evaluating the delayed effects. The concrete semantics can be understood either as a relational semantics or as a program symbolic execution [1, Ch. 3.4.5] which is abstracted without loosing information about the relationships between the internal and external objects of the program part thanks to the use of a relational domain.

An example is the pointer analysis using collections [42] of [16, Sec. 4.2.2]. There pointer variables are organized in equivalence classes where variables in different classes cannot point, even indirectly, to the same position on the heap. This analysis is relational and can be started by giving names to actual parameters which are in the same class as the formal parameters (as well as their potential aliases, as specified in the assumption interface).

Another example, illustrated below, uses the polyhedral abstract domain [43] so that functions (or procedures in the case of imperative programs) can be approximated by relations. These relations can be further approximated by linear inequalities between values of variables [43]. Let us illustrate this method using a Pascal example taken from [44]:

```
procedure Hanoi (n : integer; var a, b, c : integer;
                              var Ta, Tb, Tc : Tower);
  begin
   { n = n0 ∧ a = a0 ∧ b = b0 ∧ c = c0 }
   if n = 1 then begin
       b := b + 1; Tb[b] := Ta[a]; Ta[a] := 0;  a := a − 1;
     { n = n0 = 1 ∧ a = a0 − 1 ∧ b = b0 + 1
        ∧ c = c0 }
   end else begin
     { n = n0 ∧ a = a0 ∧ b = b0 ∧ c = c0 }
     Hanoi(n − 1, a, c, b, Ta, Tc, Tb);
     { n = n0 > 1 ∧ a = a0 − n + 1 ∧ b = b0
        ∧ c = c0 + n − 1 }
     b := b + 1; Tb[b] := Ta[a];  Ta[a] := 0;  a := a − 1;
     { n = n0 > 1 ∧ a = a0 − n ∧ b = b0 + 1
        ∧ c = c0 + n − 1 }
     Hanoi(n − 1, c, b, a, Tc, Tb, Ta);
      { n = n0 > 1 ∧ a = a0 − n ∧ b = b0 + n
         ∧ c = c0 }
   end;
   { n = n0 ≥ 1 ∧ a = a0 − n0 ∧ b = b0 + n0
      ∧ c = c0 }
  end;
```

[3] [41] is another example of lazy static program analysis used in the context of demand-driven analysis.

The result of analyzing this procedure, which is given above between brackets {...} is independent of the values of the actual parameters provided in calls. This is obtained by giving formal names n_0, a_0, b_0 and c_0 to the values of the actual parameters corresponding to the initial values of the formal parameters `n`, `a`, `b` and `c` (array parameters `Ta`, `Tb` and `Tc` are simply ignored, which corresponds to a worst-case analysis) and by establishing a relation with the final value of these formal parameters. The result is a precise description of the effect of the procedure in the form of a relation between initial and final values of its parameters:

$$\phi(n_0, a_0, b_0, c_0, n, a, b, c) = (n = n_0 \geq 1 \wedge a = a_0 - n_0 \wedge b = b_0 + n_0 \wedge c = c_0)$$

Observe that it is automatically shown that $n_0 \geq 1$, which is a necessary condition for termination.

In a function call, n_0, a_0, b_0 and c_0 are set equal to the values of the actual parameters in ϕ and eliminated by existential quantification. For example:

```
a := n; b := 0; c := 0;
{ n = a ∧ b = 0 ∧ c = 0 }
Hanoi(n, a, b, c, Ta, Tb, Tc);
{ ∃n0, a0, b0, c0 : n0 = a0 ∧ b0 = 0 ∧ c0 = 0 ∧
n = n0 ≥ 1 ∧ a = a0 − n0 ∧ b = b0 + n0 ∧ c = c0 }
```

This last post-condition can be simplified by projection as:

```
{ a = 0 ∧ n = b ≥ 1 ∧ c = 0 }
```

In recursive calls, successive approximations of the relation ϕ must be used, starting from the empty one. A widening (followed by a narrowing) [2], [16] can be used to ensure convergence.

Such relational analyzes are also very useful in the more classical context where functions are analyzed in the order of the dependence graph (see Sec. III-C) since, as shown above, the relational analysis of the function determines a relationship between the inputs and the outputs of the function. This allows the function to be analyzed independently of its call sites and therefore the analysis becomes "context-sensitive" which improves the precision (and may decrease the cost if the function/procedure may be analyzed only once, not for all different possible contexts).

An example of such a symbolic relational separate analysis is the notion of *summary transfer function* of [45], [46] in the context of points-to analysis for C++. A summary transfer function for a method expresses the effects of the method invocation on the points-to solution parameterized by unknown symbolic initial values and conditions on these values.

Another example of symbolic relational separate analysis is the strictness analysis of higher-order functions [47] using a symbolic representation of boolean higher order functions called Typed Decision Graphs (TDGs), a refinement of Binary Decision Diagrams (BDDs).

A last example the backward escape analysis of *first-order* functions in [32] since the escape information for each parameter is computed as a function of the escape information for the result. For Java™, it is not a function but a relation between the various escape information available on the parameters and the result [48].

This symbolic relational separate analysis may degenerate in the simplification case of Sec. IV if no local iteration is possible. However this situation is rare since it is quite uncommon that all program parts circularly depend upon one another.

VIII. Combination of Separate Analysis Methods

The last category of methods essentially consists in combining the previous local separate analysis methods and/or some form of global analysis. We provide a few examples below.

A. Preliminary Global Analysis and Simplification

We have already indicated that a preliminary rough global program analysis can always be performed to improve the information available before performing a local analysis. A classical example is pointer analysis [49], [50], [51], [52], [53], [54], [55], [56], [57], [25], [58], see an overview in [59]. A preliminary pointer analysis is often mandatory since making conservative assumptions regarding pointer accesses can adversely affect the precision and efficiency of the analysis of the program parts requiring this information. Such pointer alias analysis attempts to determine when two pointer expressions refer to the same storage location and is useful to detect potential side-effects through assignment and parameter passing.

Also the simplification algorithms considered in Sec. IV are applicable in all cases.

B. Iterated Separate Program Static Analysis

Starting with a worst case assumption $\vec{Y}^0 = \bar{\top}$, $\vec{X}_1^0 = \bar{\top}$, ..., $\vec{X}_n^0 = \bar{\top}$ a separate analysis with interfaces as considered in Sec. VI can be iterated by successively computing:

$$\begin{array}{rcl} \vec{X}_i^{k+1} & = & \mathrm{lfp}^{\bar{\sqsubseteq}_i} \boldsymbol{\lambda} \vec{X}_i \cdot \bar{F}[\![P_i]\!]\langle \vec{Y}^k,\, \vec{X}_1^k,\, \ldots,\, \vec{X}_i,\, \ldots,\, \vec{X}_n^k \rangle \\ i = 1, \ldots, n & & \qquad\qquad\qquad\qquad\qquad\qquad (1) \\ \vec{Y}^{k+1} & = & \mathrm{lfp}^{\bar{\sqsubseteq}} \boldsymbol{\lambda} \vec{Y} \cdot \bar{F}[\![P[P_1, \ldots, P_n]]\!]\langle \vec{Y},\, \vec{X}_1^k,\, \ldots,\, \vec{X}_n^k \rangle \end{array}$$

Note that this decreasing iteration is similar to the *iterative reduction* idea of [60, Sec. 11.2] and different from and less precise than a chaotic iteration for the global analysis (which would start with $\vec{Y}^0 = \bar{\bot}$, $\vec{X}_1^0 = \bar{\bot}$, ..., $\vec{X}_n^0 = \bar{\bot}$). However the advantage is that one can stop the analysis at any step $k > 0$, the successive analyzes being more precise as k increases (a narrowing operation [2] may have to be used in order to ensure the convergence when $k \to +\infty$).

A variant consists in starting with the user provided interfaces $\vec{Y}^0 = \vec{J}$, $\vec{X}_1^0 = \vec{I}_1$, ..., $\vec{X}_n^0 = \vec{I}_n$. Then the validity of the final result $\vec{Y}^k$, $\vec{X}_1^k$, ..., $\vec{X}_n^k$ must be checked as indicated in Sec. VI.

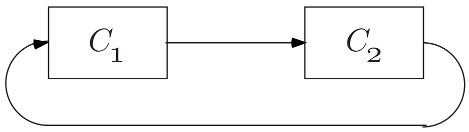
A particular case is when some program parts are missing so that their initial interfaces are initially $\bar{\top}$ and are refined by a new iteration (1) as soon as they become available. Again after each iteration k, the static program analysis of the partial program is correct.

Yet another variant consists in successively refining the abstract domains $\langle D, \sqsubseteq \rangle$, $\langle D_1, \sqsubseteq_1 \rangle$, ..., $\langle D_n, \sqsubseteq_n \rangle$ between the successive iterations k. The choice of this refinement can be guided by interaction with the user. Sometimes, it can also be automated [61], [62], [63].

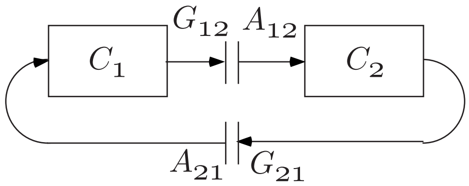
C. Creating Parts Through Cutpoints

Most often the parts P_1, ..., P_n of a program $P[P_1, \ldots, P_n]$ are determined on syntactic criteria (such as components, modules, classes, functions, procedures, methods, libraries, etc.). A preliminary static analysis can also be used to determine the parts on semantic grounds.

For example in Sec. II on global static analysis, we have considered chaotic iteration strategies [14], [15] that can be used to iterate successively on components of the system of equations (as determined by a topological ordering of the dependency graph of the system of equations). Such dependences can also be refined on semantic grounds (such as definition-use chains [64]). These dependences can be used as a basis to split the whole program into parts by introducing interfaces as considered in Sec. VI. For example, with a dependence graph of the form:



the iteration will be $((C_1)^\star; (C_2)^\star)^\star$ where $(C_i)^\star$ denotes the local iteration within the connected component C_i, $i = 1, 2$, ";" is the sequential composition and the external iteration $(\ldots)^\star$ handles the external loop. By designing interfaces at the two cutpoints:



one can have a parallel treatment of the two components as $((C_1)^\star \parallel (C_2)^\star)^\star$. Moreover a preliminary dependency analysis of the variables can partition the variables into the global ones and those which scope is restricted to one connected component only, so as to reduce the memory size needed to separately analyze the parts. If we have $G_{12} \Rightarrow A_{12}$ and $G_{21} \Rightarrow A_{21}$ then G_{12} and G_{21} are invariants in the sense of Floyd [65] so that no global iteration is needed. Otherwise the external iteration can be used to strengthen the interface until a fixpoint is reached, as done in Sec. VIII-B. The limit of this approach is close to classical proof methods with user-provided invariants at cutpoints of all loops [18].

D. Refinement of the Abstract Domain into a Symbolic Relational Domain

Separate non-relational static program analyzes (such as sign analysis, interval analysis, etc.) expressing properties of individual objects of programs (such as ranges of values of numerical variables) but no relationships between objects manipulated by the program (such as the equality of the values of program variables at some program point) cannot be successfully used for the relational separate analysis considered in Sec. VII which, in absence of user-provided information, amounts to the worst-case separate analysis of Sec. V. In this case, and whenever the symbolic relational separate analysis considered in Sec. VII is not applicable, it is always possible to refine the non-relational abstract domain into a relational one for which the separate analysis method is applicable. This can be feasible in practice if the considered program parts are small enough to be analyzed at low cost using such precise abstract domains. A classical example consists in analyzing locally program parts (e.g. procedures) with the polyhedral domain of linear inequalities [43] and the global program with the much less precise abstract domain of intervals [2]. If the polyhedral domain is too expensive, the less precise domain of difference bound matrices [17] can also be used for the local relational analyzes of program parts. This is essentially the technique used by [66].

E. Unknown Dependence Graph

Separate static program analysis is very difficult when the dependence graph is not known in a modular way (which is the case with higher-order functions in functional languages or with virtual methods in object-oriented languages). When the dependence graph is fully known and can be decomposed modularly, the symbolic relational separate analysis technique of Sec. VII is very effective. If the graph is not modular and parts can hardly be created through cutpoints as suggested in Sec. ssec:CPTP or the dependence graph is partly unknown, the difficulty in the lazy symbolic representation of the unknown part of Sec. VII is when the effect of this unknown part must later be iterated. Iin the worst case, the delaying technique of Sec. VII then amounts to a mere simplification as considered in Sec. IV. As already suggested, computational costs can then only cut down through of the worst-case separate analysis of Sec. V or by an over-estimation of the dependence graph (such as the 0-CFA control-flow analysis in functional languages [67] or the class hierarchy analysis in object-oriented languages [68]).

IX. Conclusion

The wide range of program static analysis techniques that have been developed over the past two decades allows to analyze very large programs (over 1.4 million lines of code) in a few seconds or minutes but with a very low precision [69] up to precise relational analyses which are able to analyze large programs (over 120 thousands lines of code) in a few hours or days [70] and to very detailed and precise analyzes that do not scale up for programs over a few hundred lines of code.

If such static program analyses are to scale up to precise analysis of huge programs (some of them now reaching 30 to 40 millions of lines), compositional separate modular methods are mandatory. In this approach very precise analyzes (in the style of Sec. VII) can be applied locally to small program parts. This local analysis phase can be fast if all these preliminary analyzes are performed independently in parallel. Then a cheap global program analysis can be performed using the results of the previous analyzes, using maybe less precise analyzes which have a low cost. The idea can obviously be repeatedly applied in stages to larger and larger parts of the program with less and less refined abstract domains.

Moreover the design of specification and programming languages including user-specified of program parts interfaces can considerably facilitate such compositional separate modular static analysis of programs.

Acknowledgements

We thank Bruno Blanchet, Jer me Feret, Charles Hymans, Francesco Logozzo, Laurent Mauborgne, Antoine Min and Barbara G. Ryder for their comments on a preliminary version of this paper.

References

[1] P. Cousot, *M thodes it ratives de construction et d'approximation de points fixes d'op rateurs monotones sur un treillis, analyse s mantique de programmes*, Th se d' tat s sciences math matiques, Universit scientifique et m dicale de Grenoble, Grenoble, France, 21 March 1978.

[2] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, 1977, pp. 238–252, ACM Press, New York, New York, United States.

[3] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, Texas, 1979, pp. 269–282, ACM Press, New York, New York, United States.

[4] P. Cousot, "Abstract interpretation: Achievements and perspectives, invited paper," in *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 224 and electronic proceedings `http://www.ssgrr.it/en/ssgrr2000/proceedings.htm`, L'Aquila, Italy, 31 July – 6 August 2000, Scuola Superiore G. Reiss Romoli.

[5] P. Cousot, "Abstract interpretation based formal methods and future challenges, invited paper," in *« Informatics — 10 Years Back, 10 Years Ahead »*, R. Wilhelm, Ed., vol. 2000 of *Lecture Notes in Computer Science*, pp. 138–156. Springer-Verlag, Berlin, Germany, 2000.

[6] P. Cousot and R. Cousot, "Abstract interpretation frameworks," *Journal of Logic and Computation*, vol. 2, no. 4, pp. 511–547, August 1992.

[7] P. Cousot and R. Cousot, "Introduction to abstract interpretation," Course notes for the "NATO International Summer School 1998 on Calculational System Design", Marktoberdorff, 28 July – 9 August 1998.

[8] P. Cousot, "Types as abstract interpretations, invited paper," in *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997, pp. 316–331, ACM Press, New York, New York, United States.

[9] P. Cousot and R. Cousot, "Formal language, grammar and set-constraint-based program analysis by abstract interpretation," in

Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture, La Jolla, California, 25–28 June 1995, pp. 170–181, ACM Press, New York, New York, United States.

[10] P. Cousot and R. Cousot, "Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, invited paper," in *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95*, P. Wolper, Ed. 3–5 July 1995, Li ge, Belgium, Lecture Notes in Computer Science 939, pp. 293–308, Springer-Verlag, Berlin, Germany.

[11] P. Cousot and R. Cousot, "Abstract interpretation and application to logic programs," *Journal of Logic Programming*, vol. 13, no. 2–3, pp. 103–179, 1992, (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).

[12] P. Cousot and R. Cousot, "Abstract interpretation based program testing, invited paper," in *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 248 and electronic proceedings `http://www.ssgrr.it/en/ssgrr2000/proceedings.htm`, L'Aquila, Italy, 31 July – 6 August 2000, Scuola Superiore G. Reiss Romoli.

[13] D. Mass , "Combining forward and backward analyzes of temporal properties," in *Proceedings of the Second Symposium PADO '2001, Programs as Data Objects*, 0. Danvy and A. Filinski, Eds. 2001, rhus, Denmark, 21–23 May 2001, Lecture Notes in Computer Science 2053, pp. 155–172, Springer-Verlag, Berlin, Germany.

[14] P. Cousot and R. Cousot, "Automatic synthesis of optimal invariant assertions: mathematical foundations," in *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, New York, ACM SIGPLAN Notices 12(8):1–12, 1977.

[15] F. Bourdoncle, "Efficient chaotic iteration strategies with widenings," in *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, D. Bjørner, M. Broy, and I.V. Pottosin, Eds. 28 June –2 July 1993, Akademgorodok, Novosibirsk, Russia, Lecture Notes in Computer Science 735, pp. 128–141, Springer-Verlag, Berlin, Germany.

[16] P. Cousot and R. Cousot, "Static determination of dynamic properties of recursive procedures," in *IFIP Conference on Formal Description of Programming Concepts, St-Andrews, N.B., Canada*, E.J. Neuhold, Ed. 1977, pp. 237–277, North-Holland Pub. Co., Amsterdam, The Netherlands.

[17] A. Min , "A new numerical abstract domain based on difference-bound matrices," in *Proceedings of the Second Symposium PADO '2001, Programs as Data Objects*, 0. Danvy and A. Filinski, Eds. 2001, rhus, Denmark, 21–23 May 2001, Lecture Notes in Computer Science 2053, pp. 155–172, Springer-Verlag, Berlin, Germany.

[18] P. Cousot, "Methods and logics for proving programs," in *Formal Models and Semantics*, J. van Leeuwen, Ed., vol. B of *Handbook of Theoretical Computer Science*, chapter 15, pp. 843–993. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[19] M. Codish, S. Debray, and R. Giacobazzi, "Compositional analysis of modular logic programs," in *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, 1993, pp. 451–464, ACM Press, New York, New York, United States.

[20] O. Lee and K. Yi, "A proof method for the correctness of modularized kCFAs," Technical Memorandum ROPAS-2000-9, Research On Program Analysis System, Korea Advanced Institute of Science and Technology, November 2000, `http://ropas.kaist.ac.kr/~cookcu/paper/tr2000b.ps.gz`.

[21] C. Flanagan and J.B. Saxe, "Avoiding exponential explosion: generating compact verification conditions," in *Conference Record of the Twentyeight Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM SIGPLAN Notices 36(3)*, London, Great Britain, January 2001, pp. 193–205, ACM Press, New York, New York, United States.

[22] N. Jones, C.K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, International Series in Computer Science. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, United States, June 1993.

[23] C. Flanagan and M. Felleisen, "Componential set-based analysis," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 2, pp. 370–416, February 1999.

[24] A. Rountev and S. Chandra, "Off-line variable substitution for scaling points-to analysis," in *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI). ACM SIGPLAN Notices 35(5)*, Vancouver, British Columbia, Canada, 18–21 June 2000, pp. 47–56.

[25] A. Rountev and B. Ryder, "Points-to and side-effect analyses for programs built with precompiled libraries," in *Proceedings of the Tenth International Conference on Compiler Construction, CC '2001*, R. Wilhelm, Ed. 2–6 April 2001, Genova, Italy, Lecture Notes in Computer Science 2027, pp. 20–36, Springer-Verlag, Berlin, Germany.

[26] G. Amato and F. Spoto, "Abstract compilation for sharing analysis," in *Proceedings of the FLOPS 2001 Conference*, H. Kuchen and K. Ueda, Eds., Tokyo, Japan, March 2001, vol. 2024 of *Lecture Notes in Computer Science*, pp. 311–325, Springer-Verlag, Berlin, Germany.

[27] D. Boucher and M. Feeley, "Abstract compilation: A new implementation paradigm for static analysis," in *Proceedings of the Sixth International Conference on Compiler Construction, CC '96*, T. Gyimothy, Ed. 24–26 April 1996, Link ping, Sweden, Lecture Notes in Computer Science 1060, pp. 192–207, Springer-Verlag, Berlin, Germany.

[28] M. Codish and B. Demoen, "Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop," in *Proceedings of the First International Symposium on Static Analysis, SAS '94*, B. Le Charlier, Ed., Namur, Belgium, 20–22 september 1994, Lecture Notes in Computer Science 864, pp. 281–296. Springer-Verlag, Berlin, Germany, 1994.

[29] S.K. Debray and D.S. Warren, "Automatic mode inference for logic programs," *Journal of Logic Programming*, vol. 5, no. 3, pp. 207–229, september 1988.

[30] F. Mal sieux, O. Ridoux, and P. Boizumault, "Abstract compilation of Lambda-Prolog," in *Joint International Conference and Symposium on Logic Programming, JICSLP '98*, J. Jaffar, Ed. 15–19 June 1992, Manchester, United Kindom, pp. 130–144, MIT Press, Cambridge, Massachusetts, United States.

[31] M. Felleisen, "Program analyses: A consumer's perspective and experiences, invited talk," in *Proceedings of the Seventh International Symposium on Static Analysis, SAS '2000*, J. Palsberg, Ed., Santa Barbara, California, United States, Lecture Notes in Computer Science 1824. Springer-Verlag, Berlin, Germany, 29 June – 1 July 2000, Presentation available at URL `http://www.cs.rice.edu:80/~matthias/Presentations/SAS.ppt`.

[32] B. Blanchet, "Escape analysis: Correctness proof, implementation and experimental results," in *Conference Record of the Twentyfifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, United States, 19–21 January 1998, pp. 25–37, ACM Press, New York, New York, United States.

[33] J. Feret, "Confidentiality analysis of mobile systems," in *Proceedings of the Seventh International Symposium on Static Analysis, SAS '2000*, J. Palsberg, Ed., Santa Barbara, California, United States, Lecture Notes in Computer Science 1824, pp. 135–154. Springer-Verlag, Berlin, Germany, 29 June – 1 July 2000.

[34] A. Rountev, B.G. Ryder, and W. Landi, "Data-flow analysis of program fragments," in *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lecture Notes in Computer Science 1687*, O. Nierstrasz and M. Lemoine, Eds., Toulouse, France, september 1999, pp. 235–252.

[35] P. Colette and C.B. Jones, "Enhancing the tractability of rely/guarantee specifications in the development of interfering operations," in *Proof, Language and Interaction*, G. Plotkin, C. Stirling, and M. Tofte, Eds., chapter 10, pp. 277–307. MIT Press, Cambridge, Massachusetts, United States, 2000.

[36] Y.M. Tang and P. Jouvelot, "Separate abstract interpretation for control-flow analysis," in *Proceedings of the International Conference on Theoretical Aspects of Computer Software, TACS '95, Lecture Notes in Computer Science 789*, M. Hagiya and J.C. Mitchell, Eds., Sendai, Japan, pp. 224–243. Springer-Verlag, Berlin, Germany, 19–22 April 1994.

[37] M.J. Harrold, D. Liang, and S. Sinha, "An approach to analyzing and testing component-based systems," in *Proceedings of the First International ICSE Workshop on Testing Distrib-*

uted Component-Based Systems. Los Angeles, California, United States, 17 May 1999.

[38] P. Cousot, "Semantic foundations of program analysis," in *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D. Jones, Eds., chapter 10, pp. 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, United States, 1981.

[39] R. Giacobazzi, "Abductive analysis of modular logic programs," in *Proceedings of the International Symposium on Logic Programming, ILPS '1994*, M. Bruynooghe, Ed. 13–17 November 1994, Ithaca, New York, United States, pp. 377–391, MIT Press, Cambridge, Massachusetts, United States.

[40] P. Cousot and R. Cousot, "Galois connection based abstract interpretations for strictness analysis, invited paper," in *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, D. Bjørner, M. Broy, and I.V. Pottosin, Eds. 28 June – 2 July 1993, Akademgorodok, Novosibirsk, Russia, Lecture Notes in Computer Science 735, pp. 98–127, Springer-Verlag, Berlin, Germany.

[41] C. Hankin and D. Le M tayer, "Lazy type inference and program analysis," *Science of Computer Programming*, vol. 25, no. 2–3, pp. 219–249, 1995.

[42] P. Cousot and R. Cousot, "Static determination of dynamic properties of generalized type unions," in *ACM Symposium on Language Design for Reliable Software*, Raleigh, North Calorina, ACM SIGPLAN Notices 12(3):77–94, 1977.

[43] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Tucson, Arizona, 1978, pp. 84–97, ACM Press, New York, New York, United States.

[44] P. Cousot and R. Cousot, "Relational abstract interpretation of higher-order functional programs," *Actes JTASPEFL '91, Bordeaux, France. BIGRE*, vol. 74, pp. 33–36, October 1991.

[45] R. Chatterjee, B.G. Ryder, and W. Landi, "Relevant context inference," in *Conference Record of the Twentysixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, Texas, United States, 19–21 January 1999, pp. 133–146, ACM Press, New York, New York, United States.

[46] R. Chatterjee, B.G. Ryder, and W. Landi, "Relevant context inference," Technical report DCS-TR-360, Department of Computer Science, Rutgers University, January 1999, `ftp://athos.rutgers.edu/pub/technical-reports/dcs-tr-360.ps.Z`.

[47] L. Mauborgne, "Abstract interpretation using typed decision graphs," *Science of Computer Programming*, vol. 31, no. 1, pp. 91–112, May 1998.

[48] B. Blanchet, "Escape analysis for object-oriented languages: Application to Java," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99). ACM SIGPLAN Notices 34(10)*, Denver, Colorado, United States, 1–5 November 1999, pp. 20–34.

[49] M. Emami, R. Ghiya, and L.J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI). ACM SIGPLAN Notices 28(6)*, Orlando, Florida, United States, June 1994, pp. 242–256, ACM Press, New York, New York, United States.

[50] R. Ghiya and L.J. Hendren, "Putting pointer analysis to work," in *Conference Record of the Twentyfifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, United States, 19–21 January 1998, pp. 121–133, ACM Press, New York, New York, United States.

[51] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 4, pp. 848–894, July 1999.

[52] M. Hind and A. Pioli, "Assessing the effects of flow-sensitivity on pointer alias analyses," in *Proceedings of the Fifth International Symposium on Static Analysis, SAS '98*, G. Levi, Ed., Pisa, Italy, 14–16 september 1998, Lecture Notes in Computer Science 1503, pp. 57–81. Springer-Verlag, Berlin, Germany, 1998.

[53] S. Horwitz, "Precise flow-insensitive may-alias analysis is NP-hard," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 1, pp. 1–6, January 1997.

[54] S. Jagannathan, P. Thiemann, S. Weeks, and A.K. Wright, "Single and loving it: Must-alias analysis for higher-order languages," in *Conference Record of the Twentyfifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, United States, 19–21 January 1998, pp. 329–341, ACM Press, New York, New York, United States.

[55] W.A. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, December 1992.

[56] D. Liang and M.J. Harrold, "Efficient points-to analysis for whole-program analysis," in *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lecture Notes in Computer Science 1687*, O. Nierstrasz and M. Lemoine, Eds., Toulouse, France, september 1999, pp. 199–215.

[57] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1467–1471, september 1994.

[58] A. Venet, "Automatic analysis of pointer aliasing for untyped programs," *Science of Computer Programming, Special Issue on SAS'96*, vol. 35, no. 1, pp. 223–248, september 1999.

[59] B.G. Ryder, W. Landi, P.A. Stocks, S. Zhang, and R. Altucher, "A schema for interprocedural side effect analysis with pointer aliasing," *ACM Transactions on Programming Languages and Systems*, 2001, To appear.

[60] P. Cousot, "The calculational design of a generic abstract interpreter," in *Calculational System Design*, M. Broy and R. Steinbr ggen, Eds., vol. 173, pp. 421–505. NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, Amsterdam, The Netherlands, 1999.

[61] R. Giacobazzi and E. Quintarelli, "Incompleteness, counterexamples and refinements in abstract model-checking," in *Proceedings of the Eight International Symposium on Static Analysis, SAS '01*, P. Cousot, Ed. 16–18 July 2001, Paris, France, Lecture Notes in Computer Science 2126, pp. 356–373, Springer-Verlag, Berlin, Germany.

[62] R. Giacobazzi and F. Ranzato, "Refining and compressing abstract domains," in *Proceedings of the Twentyfourth International Colloquium International Colloquium on Automata, Languages and Programming '97*, P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, Eds. 1997, vol. 1256 of *Lecture Notes in Computer Science*, pp. 771–781, Springer-Verlag, Berlin, Germany.

[63] R. Giacobazzi, F. Ranzato, and F. Scozzari, "Making abstract interpretations complete," *Journal of the Association for Computing Machinary*, vol. 47, no. 2, pp. 361–416, 2000.

[64] M.J. Harrold and M.L. Soffa, "Efficient computation of interprocedural definition-use chains," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 2, pp. 175–204, March 1994.

[65] R.W. Floyd, "Assigning meaning to programs," in *Proceedings of the Symposium in Applied Mathematics*, J.T. Schwartz, Ed., vol. 19, pp. 19–32. American Mathematical Society, Providence, Rhode Island, United States, 1967.

[66] Z. Xu, T. Reps, and B.P. Miller, "Typestate checking of machine code," in *Proceedings of the Tenth European Symposium on Programming Languages and Systems, ESOP '2001*, D. Sands, Ed. 2001, Genova, Italy, 2–6 April 2001, Lecture Notes in Computer Science 2028, pp. 335–351, Springer-Verlag, Berlin, Germany.

[67] O. Shivers, "The semantics of scheme control-flow analysis," in *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '91*, P. Hudak and N.D. Jones, Eds. september 1991, Yale U., New Haven, Connecticut, United States, 17–19 June 1991, ACM SIGPLAN Notices 26(9), pp. 190–198, ACM Press, New York, New York, United States.

[68] J. Dean, Grove D., and G. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the Ninth European Conference on Object-Oriented Programming, ECOOP '95*, W.G. Olthoff, Ed., rhus, Denmark, 7–11 August 1995, Lecture Notes in Computer Science 952, pp. 77–101. Springer-Verlag, Berlin, Germany, 1995.

[69] M. Das, B. Liblit, M. F hndrich, and J. Rehof, "Estimating the impact of scalable pointer analysis on optimization," in *Proceedings of the Eight International Symposium on Static Analysis, SAS '01*, P. Cousot, Ed., Paris, France, Lecture Notes in Computer Science 2126, pp. 259–277. Springer-Verlag, Berlin, Germany, 16–18 July 2001.

[70] F. Randimbivololona, J. Souyris, and A. Deutsch, "Improving

avionics software verification cost-effectiveness: Abstract interpretation based technology contribution," in *Proceedings DASIA 2000 – DAta Systems In Aerospace*. 22–26 May 2000, Montreal, Canada, ESA Publications.