Clousot: Static Contract Checking with Abstract Interpretation

Manuel Fähndrich and Francesco Logozzo

Microsoft Research, Redmond, WA (USA)
{maf, logozzo}@microsoft.com

Abstract. We present an overview of Clousot, our current tool to statically check CodeContracts. CodeContracts enable a compiler and language-independent specification of Contracts (precondition, postconditions and object invariants).
Clousot checks every method in isolation using an assume/guarantee reasoning: For each method under analysis Clousot assumes its precondition and asserts the postcondition. For each invoked method, Clousot asserts its precondition and assumes the postcondition. Clousot also checks the absence of common runtime errors, such as null-pointer errors, buffer or array overruns, divisions by zero, as well as less common ones such as checked integer overflows or floating point precision mismatches in comparisons. At the core of Clousot there is an abstract interpretation engine which infers program facts. Facts are used to discharge the assertions. The use of abstract interpretation (*vs* usual weakest precondition-based checkers) has two main advantages: (i) the checker automatically infers loop invariants letting the user focus only on boundary specifications; (ii) the checker is deterministic in its behavior (which abstractly mimics the flow of the program) and it can be tuned for precision and cost. Clousot embodies other techniques, such as iterative domain refinement, goal-directed backward propagation, precondition and postcondition inference, and message prioritization.

1 Introduction

A limiting factor to the adoption of formal methods in everyday programming practice is that tools do not integrate well into the existing programming workflow. Often, the price programmers have to pay to enjoy the benefits of formal methods include the use of non-mainstream languages or non-standard compilers.

The CodeContracts project [15] at Microsoft Research aims at bridging the gap between practice and formal specification and verification using the principle of least interference in the programmer's existing workflow. The main insight of CodeContracts is that program specifications can be authored as code [16]. Contracts take the form of method calls to a standard library. Therefore CodeContracts enable the programmer to write down specifications as Boolean expressions in their favorite .NET language (C#, F#, VB . . .). This has several advantages: the semantics of contracts is given by the IL produced by the compiler, no compiler modification is required, contracts are serialized and persisted as code (no need for separate parsing, type-checking ...), all the IDE support (intellisense, code refactoring ...) the programmer is used to is automatically leveraged.

CodeContracts provide a standard and uniform way to describe contracts which can then be consumed by several tools. At Microsoft Research, we have developed tools to automatically generate the documentation (ccdoc), to perform runtime checking (ccrewrite) and to perform static checking (cccheck, internally called Clousot). The tools are available for download at

$$\texttt{http://msdn.microsoft.com/es-ar/devlabs/dd491992(en-us).aspx}$$

A main difference of our static contract checker, with respect to similar and existing ones is that it is based on abstract interpretation [9] instead of solely relying on a theorem prover (automatic [19, 2, 18] or semiautomatic [3]). The use of abstract interpretation allows the checker to focus on some properties of interest, as for instance non-nullness, linear arithmetic or array invariants while forgetting more complex or unusual ones such as existentially quantified or arbitrarily universally quantified properties. An abstract interpretation-based static checker has the advantage of being more automatic and tunable than theorem prover-based ones. For instance, it can automatically compute loop invariants, which frees the programmer from the burden of specifying (often self-evident) loop invariants. The built-in abstract domains are optimized for the properties of interest, so that the precision/cost ratio can be finely set. Furthermore, the analysis is deterministic, in that it does not depend on internals of theorem provers such as random seeding, quantifier instantiation, or matching loops.

2 CodeContracts by Example

The class in Fig. 1 in an example of a C# class annotated with CodeContracts specifications. Contracts are defined by means of calls to static methods of a `Contract` class, part of .NET since $v4.0$. The class implements a simple stack of non-null objects. Externally, one can create a stack, can push or pop elements, can inquire about the number of stack elements and whether the stack is empty or not. Internally, the stack is backed-up by two fields: a growing array of objects containing the stack elements and a pointer to the next free position in the stack.

As a programmer, one would like to express some simple properties about those fields. The first property is that the array is never null and that the pointer can never be negative. Furthermore, the stack pointer can never be larger than the array length (it can be equal when the stack is full). Finally, all the elements in the interval $\texttt{a}[0] \ldots \texttt{a}[\texttt{nextFree} - 1]$ should be not-null.

```
public class NonNullStack<T> where T : class
  {
    private T[] arr;
    private int nextFree;

    [ContractInvariantMethod] /* Define the object invariant */
    void ObjectInvariant()
    {
      Contract.Invariant(arr !=null);
      Contract.Invariant(nextFree >= 0);
      Contract.Invariant(nextFree <= arr.Length);
      Contract.Invariant(Contract.ForAll(0, nextFree, i => arr[i] != null));
    }

    public NonNullStack(int len)
    {
      Contract.Requires(len >= 0); /* Method precondition */

      this.arr = new T[len];
      this.nextFree = 0;
    }

    public void Push(T x)
    {
      Contract.Requires(x != null);

      if (nextFree == arr.Length)
      {
        var newArr = new T[arr.Length * 2]; /* bug here */
        for (int i = 0; i < nextFree; i++) newArr[i] = arr[i];
        arr = newArr;
      }
      this.arr[nextFree++] = x;
    }

    public T Pop()
    {
      Contract.Requires(!this.IsEmpty);
      Contract.Ensures(Contract.Result<T>() != null); /* Method postcondition */

      return this.arr[--nextFree];
    }

    public bool IsEmpty { get { return this.nextFree == 0; } }
    public int Count { get { return this.nextFree; } }
  }
```

Fig. 1. A (buggy) implementation of a stack of non-null values annotated with Code-Contracts. `Contract.Requires` specifies the precondition, `Contract.Ensures` specify the postcondition, `Contract.Result` denotes the return value (not expressible in C#). The attribute `ContractInvariantMethod` tags the method containing the object invariant (specified with the `Contract.Invariant`).

2.1 Specification

The formal specification with CodeContracts of those invariants is given by the method `ObjectInvariant` if Fig. 1. CodeContracts require the object invariant to be specified in a void method annotated with the attribute `ContractInvariantMethod`. The object invariant method body can only contain calls to `Contract.Invariant`, which specify the object invariant. Valid conditions for contracts are language expressions, including those containing method calls (provided the callee is marked with the [`Pure`] attribute) augmented with dummy methods to specify limited universal (`Contract.ForAll`) and existential quantification (`Contract.Exists`).

Preconditions are expressed via `Contract.Requires`. In the example, the precondition of the `NonNullStack` constructor requires the caller to pass a non-negative initial size for the stack.

Postconditions are expressed via `Contract.Ensures`. In the example, the postcondition of the method `Pop` ensures that the returned value is not-null. The void method call `Contract.Result` ⟨`T`⟩() is used to denote the return value of the method, which is not directly expressible in the source language.

CodeContracts, being simple method calls, are totally transparent to the compiler, and thanks to the shared type system in .NET, also to the different languages. Programmers can author Contracts in their favorite .NET language (C#, VB, F# . . .). The compiler compiles contracts to straight CIL (Common Intermediate Language [14]). Our tool extracts the contracts from the CIL and use them for multiple purposes: Documentation generation, Runtime checking and Static checking (Clousot).

2.2 Static checking

Clousot analyzes every method in isolation, using the usual assume/guarantee reasoning. The precondition of the method is turned into an assumption and the postcondition into an assertion. For public methods, the object invariant is assumed at the method entry and asserted at the exit point. For each method call, its precondition is asserted, and the postcondition assumed.

From a user-perspective, Clousot makes the distinction between explicit and implicit assertions (or proof obligations). *Explicit* proof obligations are those provided by the user as specifications or as an explicit assertion. In the running example, the object invariant and the postcondition of `Pop` are the assertions to be proved. *Implicit* proof obligations are those defined by the CIL language semantics, to avoid runtime errors such as null deference, index out of range for arrays or overflows for checked arithmetic expressions, but also buffer overruns which do not cause an exception to be thrown, but may compromise the stability (and security) of the program. In the default configuration, Clousot only checks the explicit proof obligations, to avoid overwhelming the user with too many warning messages. At first, we want the user to focus on boundary specifications. Once those are resolved (possibly going to zero warnings), the programmer can (selectively) enable the checking of the implicit proof obligations.

The analysis proceeds by performing some abstract interpretations of the method body, where all the contracts are turned into asserts or assumes. Clousot contains abstract domains tailored to specific properties of interest, such as heap location equalities, non-nullness, linear arithmetic, disjunctions and simple universally quantified facts. Those properties are enough to analyze and verify the example of Fig. 1.

Static contract checking To prove the object invariant for `NonNullStack`, one must be able to track nonnullness (to prove that $\mathtt{arr}! = \mathtt{null}$), linear arithmetic relationships (to prove that $\mathtt{0} \leq \mathtt{nextFree} \leq \mathtt{arr}.\mathtt{Length}$) and quantified facts (to prove that $\forall i \in [0, \mathtt{arr}.\mathtt{Length}).\mathtt{arr}[i]! = \mathtt{null}$). The most interesting case is the implementation of `Push`. First it checks if the backing array is full. If it is, it allocates an array twice as large and copies all the original elements into it. Finally it updates the array with `x` and increments the stack pointer.

The nonnull analysis infers that in both if-branches $\mathtt{arr}\ ! = \mathtt{null}$, so it concludes that the first conjunct of the invariant is satisfied.

The numerical analysis infers in one case (array full) that $\mathtt{0} \leq \mathtt{nextFree} \leq \mathtt{arr}.\mathtt{Length}$ and in the other that $\mathtt{0} \leq \mathtt{nextFree} < \mathtt{arr}.\mathtt{Length}$, so that $\mathtt{0} \leq \mathtt{nextFree} \leq \mathtt{arr}.\mathtt{Length}$ holds before the array store. The method exit point is reached only if the store was successful, *i.e.*, the index was inbounds, so that the abstract element can be refined to $\mathtt{0} \leq \mathtt{nextFree} < \mathtt{arr}.\mathtt{Length}$, and hence prove the other two conjuncts of the object invariant.

The universally quantified component of the object invariant is a little bit trickier. We know that the elements $\mathtt{arr}[0] \ldots \mathtt{arr}[\mathtt{nextFree} - 1]$ are not null (from the object invariant), and that the element to be pushed is not null (from the precondition). When there is still space, we can easily conclude that the elements $\mathtt{arr}[0] \ldots \mathtt{arr}[\mathtt{nextFree} - 1], \mathtt{arr}[\mathtt{nextFree}]$ are all not null. When there is no more space, a new array is allocated and all the elements are copied into it. Proving that $\mathtt{newArr}[0] \ldots \mathtt{newArr}[\mathtt{nextFree} - 1]$ are all not null requires inferring the quantified loop invariant $\forall j \in [0, \mathtt{i}].\ \mathtt{arr}[j]! = \mathtt{null}$. In Clousot we have new abstract domains to infer such invariants efficiently (Sect. 5.4).

Static runtime-error checking Once all the boundary contracts are proved, the user can opt-in to prove the absence of common runtime errors in the implementations. For instance, the user can turn on the non-null and array bounds checking. Then every time a field, an array, and in general a reference is accessed, Clousot will try to prove that such a reference is not null. In our example, Clousot will prove the absence of null references in the class. As for array bounds checking, every time an array is created, read or written, Clousot will try to prove that the access is in-bounds. For instance for an array store $\mathtt{a}[\mathtt{exp}]$ Clousot will emit the condition $0 \leq \mathtt{exp}$ (underflow) and $\mathtt{exp} < \mathtt{a}.\mathtt{Length}$ (overflow). In our example the most interesting case is `Push`. When the stack is full, then a new array is allocated and all the elements are copied into it. To prove the array accesses correct, Clousot infers the loop invariant $\mathtt{0} \leq \mathtt{i} \leq \mathtt{nextFree}$, which combined with the guard $\mathtt{nextFree} == \mathtt{arr}.\mathtt{Length}$, the array creation postcondition $\texttt{newArr.Length} = 2 * \texttt{arr.Length}$ and the loop guard, allows proving the safety of the `newArr` store and `arr` read inside the loop. At the end of the loop, one only knows that $\texttt{0} \leq \texttt{nextFree} \leq \texttt{newArr.Length}$, which is not enough to prove the safety of the next store instruction. In fact, when $\texttt{a.Length} = 0$, then $0 = \texttt{nextFree} = \texttt{newArr.Length}$ and the store is indeed causing an overrun. The programmer can fix it by changing the allocation expression to $\texttt{arr.Length} * 2 + 1$, in which case Clousot will discover that $\texttt{nextFree} < \texttt{newArr.Length}$, and hence validating the store.

The programmer can be more picky, and may want to prove more things about the program. He/she can turn on the arithmetic obligations switch in Clousot to check for common arithmetic errors such as division by zero or the overflow of checked expressions. In the particular example Clousot discovers that the array allocation $\texttt{newint}[\texttt{arr.Length} * 2 + 1]$ may cause an overflow exception. The expression $\texttt{arr.Length} * 2 + 1$ may overflow to a negative `Int32`, that when converted into a `UInt32` will cause an overflow. Inserting an explicit check against overflow will remove the warning.

Finally, Clousot helps to reduce the annotation burden by inferring some "easy" postconditions. In the default settings, Clousot infers postconditions only for: (i) properties and (ii) methods that return a non-null value. For the getter `IsEmpty` in our example, Clousot infers the postcondition $\texttt{Contract.Result}\langle\texttt{bool}\rangle() == (\texttt{this.nextFree} == 0)$. The postcondition is then propagated to all the call sites, so that for instance one can prove the safety of the array load in the `Pop` method.

3 The Analysis

Target language Clousot works at the bytecode level (CIL, Common Intermediate Language [14]). This is different from many other static analyzers, which work at the source level. There are several advantages of working at the bytecode level. First, the analysis is language independent: Clousot can analyze code produced by any compiler generating CIL (C#, VB, F# ...). Second, the analysis leverages the compiler to give semantics to complex constructs. For instance C# 3.0 introduced type inference for locals. The type inference algorithm is quite complicated, but once the compiler inferred all the types, then it generates straight IL. A source level analyzer for C# 3.0 would have to replicate the compiler type inference algorithm. A bytecode level analyzer can simply analyze the compiled IL. Third, the analysis is stable among different versions of the same language: languages change, CIL stays the same. For instance, C# 4.0 added many features over C# 3.0, such as the `dynamic` keyword or named parameters. A source level analyzer would have required (at least) a new parser to adapt to the new syntax. To the bytecode level analyzer the upgrade is totally transparent. Fourth, Contracts (serialized and persisted as CIL) do not need to be decompiled to some high level description.

Bytecode analysis has drawbacks too [30]. The main one is that high-level structure is lost, so that some additional analysis must be carried out to recover some of the information. Furthermore classical static analysis refinement techniques such as loop unrolling are harder to implement.

Phases Clousot has three main phases: Inference, Checking and Propagation. During the inference phase, the methods of the assemblies to analyze are sorted, so that callees are analyzed before their callers when possible. If there is a cyclic dependency between methods, it is broken by picking one method in the chain. For each method under analysis, its IL is read from the disk and its contracts are extracted. Then the method is analyzed. By analysis we mean a fixpoint computation with widening over a suitable abstract domain. First, aliasing is resolved (under some optimistic hypotheses) and the method code is abstracted into a *scalar* program. Then further analyses are run on the top of it to infer facts on the program. In the checking phase, the (explicit and implicit) proof obligations are collected, and the inferred facts are used to discharge them. If a proof obligation cannot be discharged, then the analysis is refined. If the more refined analysis fails, then a warning is reported to the user. Eventually, the inferred facts are used to materialize method postconditions that are attached to the method under analysis, and hence automatically propagated to the call sites.

4 Basic Framework

The inference phase is in its turn divided into two phases: (i) the scalar program construction and expression recovery; and (ii) the fact discovery. The first phase takes care of building the control flow graph (CFG), extracting the contracts and inserting them at the right spots, get rid of the stack, perform a heap analysis, and reconstruct larger expressions lost during compilation. The output of this phase is a program in scalar form. The second phase takes as input the scalar program, and performs a series of value analyses to infer facts for each program point in the method body.

Contract Extraction and CFG Construction The code to be analyzed is factored into subroutines: one subroutine per method body, one subroutine for a method's preconditions, and one subroutine for a method's postconditions. The actual code to be analyzed is then formed by inserting calls to appropriate contract subroutines in the method body. Additionally, at each method call-site, we insert a call to the precondition subroutine of the called method just prior to the actual call, and a call to the corresponding postcondition subroutine immediately following the call. The actual contract calls to `Contract.Requires` or `Contract.Ensures` turn into either `assert` or `assume` statements depending on their context. `Requires` on entry of a method turn into `assume` and `Ensures` on exit of a method turn into `assert`. Conversely, at call-sites, `Requires` turn into `assert`, and `Ensures` turn into `assume`. Conditional branches are expanded into non-deterministic branches with `assume` statements on the outgoing edges. In this manner, all conditions are simply sequences of CIL instructions, no different than ordinary method body code, and all assumptions are assume statements, and all explicit proof-obligations are assert statements.

Heap Abstraction The heap is abstracted by a graph, the Heap-graph, which maintains equalities between access paths (rooted in a local or a method parameter). Nodes in the graph denote symbolic values or heap locations, and edges denote containment or field selection. The intuitive meaning is that if two paths in the graph lead to the same node, then: (i) in the concrete executions they *always* represent the same value; and (ii) this value is *symbolically* denoted by the same symbolic value `sv`. The heap graph abstraction is optimistic in that it makes certain assumptions about non-aliasing of data structures that may not be correct in all executions. It is the only place in Clousot where such assumptions are made. Namely we assume that memory locations not explicitly aliased by the code under analysis are non-aliasing. This is clearly an optimistic assumption, but works very well in practice. Second, we guess the set of heap locations that are modified at call-sites (we do not require programmers to write heap modification clauses). Our guesses are often conservative, but may be optimistic if our non-aliasing assumptions are wrong. These assumptions allow us to compute a value numbering for all values accessed by the code, including heap accessing expressions. We also introduce names for uninterpreted functions marked as [`Pure`] by the programmer. This provides reasoning over abstract predicates. Finally, abstracting the heap also removes old-expressions in postconditions that refer to the state of an expression at the beginning of the method.

To compute the value numbering, we break the control flow of the analyzed code into maximal tree fragments. The root of each tree fragment is a join point (or the method entry point) and is connected by edges to predecessor leafs of other tree fragments. The set of names used by the value numbering is unique in each tree fragment. Edges connecting tree leafs to tree roots contain a set of assignments effectively rebinding value names from one fragment to the names of the next. The resulting code is in mostly passive form, where each instruction simply relates a set of value names. The assignments on rebinding edges between tree fragments provide a way to transform abstract domain knowledge prior to the join from one set of value names to the next, so that the join can operate on a common set of value names. The rebinding acts as a generalization of ϕ-nodes. In contrast to ϕ-nodes which provide a join for each value separately, our rebindings form a join for the entire state simultaneously, which is crucial to maintain relational properties.

Example 1. Consider the code snippet in Fig. Fig. 2. The heap analysis captures the fact that `p.b` and `a.b` are aliases starting from program point $(*)$.

The heap graph looks like the one in Fig. 2 (intermediate address nodes for locals and fields have been omitted for brevity) where symbols on edges denote the fields being selected, and $\mathtt{sv}_1$ is the symbolic value of `a.b`, and $\mathtt{sv}_2$ is the symbolic value of `a.b.x`. □

```
void HeapExample(bool b, A a, P p)
{
  p.b = a.b; // (*)

  if (b)
    a.b.x = 12;
  else
    p.b.x = 4;

  Contract.Assert(a.b.x >= 4);
}
```

$$\begin{tikzcd}
\mathtt{a} \arrow[dr, "\mathtt{b}"] & & \mathtt{p} \arrow[dl, "\mathtt{b}"'] \\
& \mathsf{sv}_1 \arrow[d, "\mathtt{x}"] & \\
& \mathsf{sv}_2 &
\end{tikzcd}$$

Fig. 2. A simple program and the corresponding Heap abstraction.

In the following we let $\mathsf{sv}(\mathsf{p})$ denote the symbolic value assigned by the heap analysis to the path p.

Expression Reconstruction The expression reconstruction analysis allows to recover some of the structure of Boolean and Arithmetic expressions that may have been lost during the compilation. The analysis is similar in many aspects to the symbolic abstract domain of [35]. A main difference is that the depth of exploration for the expression reconstruction is dynamically chosen by the particular analysis (essentially performing a widening). A comprehensive discussion of the pros and the cons of a bytecode level analysis is in [30].

5 Fact Inference

5.1 NonNull Analysis

The NonNull analysis discovers those references which are definitely not-null or definitely null. Given a reference r, the analysis assigns r a value in the flat lattice $\bot \sqsubseteq \mathtt{N}, \mathtt{NN} \sqsubseteq \top$, with `N` meaning that the reference is *always* `null` and `NN` meaning that the reference is *never* `null`.

5.2 Numerical Analysis

The numerical analysis discovers ranges and linear arithmetic relationships between symbolic values. Those relationships are then used to discharge proof obligations containing numerical conditions. The numerical analysis is a usual forward fixpoint computation with widening [7] parametrized by a numerical abstract domain.

Transfer functions corresponding to CIL instructions are parametrized by the underlying abstract domain. For instance, when an array store `ldelem` $\mathsf{a}[\mathsf{exp}]$ is encountered, two numerical constraints are pushed to the numerical abstract domain: $0 \leq \mathsf{sv}(\mathsf{exp})$ and $\mathsf{sv}(\mathsf{exp}) < \mathsf{sv}(\mathsf{a.Length})$.

Example 2. Let us consider the example in Fig. 2. A simple numerical domain infers that $\mathsf{sv}_2 = 12$ at the end of the *true* branch of the conditional, and $\mathsf{sv}_2 = 4$ at the end of the *false* branch. As a consequence, at the exit point of the conditional $4 \leq \mathsf{sv}_2 \leq 12$, which is sufficient to prove the assertion. $\square$

Thresholds are used to improve the precision of the widening (as in [4]). The thresholds are collected from the constants appearing in assumptions and assertions in the method. The numerical analysis *assumes* the common case that arithmetic expressions do not overflow, but it explicitly *checks* it in presence of checked operations [1]. Therefore our assumption can be easily checked by instructing the compiler to threat all the operations as checked. Clousot will then try to prove that they do not overflow.

Numerical abstract domains They abstract the values of numerical program variables. In the literature many numerical abstract domains have been developed with different precision/cost tradeoffs. Intervals [9] infer properties in the form $\mathsf{x} \in [a, b]$, where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$. Intervals are very efficient yet unsuitable for symbolic reasoning as they do not keep track of relations among different variables. At the opposite end of the precision spectrum Polyhedra [13] capture arbitrary linear inequalities in the form of $\sum a_i \cdot \mathsf{x}_i \leq b$. Polyhedra are very precise yet expensive (the worst case, easily attained in practice is exponential). In between these two domains, other domains (*weakly* relational) have been developed to tune the precision/cost ratio. Examples include Octagons [34] ($\pm\mathsf{x} \pm \mathsf{y} \leq b$), TVPI [38] ($a_1\mathsf{x} + a_2\mathsf{y} \leq b$) or Octahedra [6] ($\sum \pm\mathsf{x}_i \leq b$). In Clousot, we first tried using some of these domains, but we found them unfit for our purposes. For instance, Octagons introduce a non-negligible slowdown (the complexity is cubic in the number of variables, with a large multiplicative constant). A known technique to have Octagons scale up is bucketing (or packing), where buckets are restricted to a certain fixed number, and some weak relations are kept by using pivot variables. We rejected buckets, as they make the analysis result dependent on the order in which the heap analysis generates the variables, introducing a degree of non-determinism in our analysis which we prefer to avoid. We also tried Polyhedra, but early results turned out to be very bad [17]. As a consequence we developed a series of new numerical abstract domains, refining and combining existing ones. They are mainly validated by empirical experimenting and tuning.

DisIntervals DisIntervals are a simple extensions of Intervals to a finite disjunction. Formally they are an abstraction of the disjunctive completion of Intervals [8]. Elements of Disintervals are normalized sequences of non-overlapping intervals: $[a_0, b_0], \ldots [a_i, b_i], [a_{i+1}, b_{i+1}] \ldots [a_n, b_n]$ with the property that only a_0 can be $-\infty$; only b_n can be $+\infty$ and that $\forall i \in [0, n-1]. b_i < a_{i+1}$. Usual

[1] The CIL instruction set has *checked* counterparts for all the arithmetic operations which cause an exception to be thrown if an overflow has occurred.

operations on Intervals can be easily lifted to Disintervals (only the widening needs some care). DisIntervals present a very cheap way to represent non-relational disjunction as well as common "negative" information. For instance $\mathsf{x} \in [-\infty, -1], [1, 5], [50, +\infty]$ is a compact representation for $\mathsf{x} \neq 0 \wedge \mathsf{x} \neq 6 \wedge \ldots \mathsf{x} \neq 49$. This kind of information is needed for instance when dealing with enumerations.

In early versions of Clousot we had one abstract domain for Intervals and one for simple disequalities. It turned out that combining the two into the Disinterval abstract domain improves the precision, simplifies the implementation, and produces no observable slow-down in our tests and experiments.

Zones DisIntervals, or Intervals are non-relational domains which are useful in many situations. However, in modular static analysis one needs to perform some form of symbolic reasoning. The easiest one involves simple upper bounds.

Example 3. Let us consider the method `AllZero` in Fig. 3. (Dis)Intervals infer the loop invariant $\mathsf{sv}(\mathsf{i}) \in [0, +\infty]$, which is enough to prove that the array store will not cause an underflow. To prove no overflow will ever occur, one needs to propagate the constraint $\mathsf{sv}(\mathsf{i}) < \mathsf{sv}(\mathsf{a.Length})$. To prove the assertion at the end of the loop, one needs to infer the loop invariant $\mathsf{sv}(\mathsf{i}) \leq \mathsf{sv}(\mathsf{a.Length})$, which together with the loop exit condition is exactly the assertion. □

```
void AllZero(int[] a)
{
  Contract.Requires(a != null);
  int i;
  for(i = 0; i < a.Length; i++) a[i] = 0;
  Contract.Assert(i == a.Length);
}
```

Fig. 3. Example requiring a numerical abstract domain able to perform symbolical reasoning.

In Clousot, WeakUpperBounds capture simple strict upper-bounds $\mathsf{x} < \mathsf{y}_0, \ldots \mathsf{y}_i$ and WeakUpperBoundsEqual capture upper-bounds $\mathsf{x} \leq \mathsf{y}_0, \ldots \mathsf{y}_i$. They enable very efficient implementations in terms of maps. We call Disintervals combined with WeakUpperBounds and WeakUpperBoundsEqual Pentagons [31]. Pentagons are essentially a weak form of the zones abstract domains [32]. The major difference is that Pentagons avoid performing the costly closure operation, relying instead on *hint* operators to keep acceptable precision at join points [27].

Linear Equalities We use the abstract domain of linear equalities [24] to infer and propagate relations in the form $\sum a_i \cdot \mathsf{x}_i = b$. The linear equalities domain enables a very efficient implementation in terms of sparse arrays which largely compensates for the cubic cost. When combined with Pentagons, Linear Equalities can produce very powerful analyses at a moderate cost.

Example 4. Let us consider the example in Fig. 4 (taken from [37]): At loop exit, (Dis)intervals infer $\mathsf{sv}(\mathsf{i}) \in [1, +\infty], \mathsf{sv}(\mathsf{j}) \in [-\infty, +\infty], \mathsf{sv}(\mathsf{x}) \in [0, 0], \mathsf{sv}(\mathsf{y}) \in$

$[-\infty, +\infty]$ and Linear Equalities infer $\mathsf{sv}(\mathsf{x})-\mathsf{sv}(\mathsf{y}) = \mathsf{sv}(\mathsf{i})-\mathsf{sv}(\mathsf{j})$. At the assertion we can then conclude that $\mathsf{sv}(\mathsf{i}) = \mathsf{sv}(\mathsf{j})$. □

```
void Foo(int i, int j)
{
  var x = i, y = j;

  if(x <= 0) return;
  while(x > 0)
  { x--; y--; }

  if(y == 0)
   Contract.Assert(i == j);
}
```

Fig. 4. Example needing the inference of the loop invariant $\langle \mathtt{x} - \mathtt{y} = \mathtt{i} - \mathtt{j}, \mathtt{x} \in [0, +\infty]\rangle$, easily obtained by combining Linear Equalities and Intervals.

Please note that even if the assertion has a shape that would fit other weak relational domains, proving it require inferring a relation involving four variables, which is out of reach of those domains. This an extremely common case that we found over and over.

Combination of domains Every single abstract domain sketched above is weak by itself, but their combination can produce very powerful analyses [10]. The basis of the combination of numerical abstract domains in Clousot is the *reduced* product [9]. Given two abstract domains A_1 and A_2, the *cartesian* product $\mathsf{A}_1 \times \mathsf{A}_2$ is equivalent to running the two analyses separately, so that no precision gain is obtained by the composition (worse, in general it can slow down the analysis). If the two domains are allowed to communicate, by either pulling or pushing information, then the analysis precision can be dramatically improved. The example of the previous section is an example of pushing: By *pushing* the information that $\mathsf{sv}(\mathsf{x}) = 0$ at the end of the loop, the abstract state for linear equalities is refined to $\mathsf{sv}(\mathsf{x}) - \mathsf{sv}(\mathsf{y}) = \mathsf{sv}(\mathsf{i}) - \mathsf{sv}(\mathsf{j}) \land \mathsf{sv}(\mathsf{x}) = 0$. Please note that linear equalities alone cannot infer that $\mathsf{sv}(\mathsf{x}) = 0$, as this is a consequence of the loop invariant $\mathsf{sv}(\mathsf{x}) \geq 0$, which is not a linear equality. Pulling is mainly used during the fixpoint computation when transfer functions may *explictly* ask other domains to refine some information, or if some relation holds. For instance suppose that we have to evaluate the expression $\mathsf{sv}(\mathsf{u}) - \mathsf{sv}(\mathsf{w})$ in an interval environment where $\mathsf{sv}(\mathsf{u}) \in [0, +\infty], \mathsf{sv}(\mathsf{w}) \in [0, +\infty]$. With no additional information the result can be any `Int32`. Intervals can *pull* information from other domains (*oracles*), for instance asking if $\mathsf{sv}(\mathsf{w}) < \mathsf{sv}(\mathsf{u})$. The oracle can return four possible outcomes: $\top$, meaning *"I do not know"*; $\bot$ meaning this program point is unreachable, so the evaluation simply returns $\bot$; true so that the result can be refined to $[1, +\infty]$; false meaning that $\mathsf{sv}(\mathsf{w}) \geq \mathsf{sv}(\mathsf{u})$ holds, so that the result can be refined to $[-\infty, 0]$. To avoid computing a fixpoint computation among the different abstract domains at every single step of the analysis, the domains are ordered according to a tree structure (as in [10]) where the most precise yet expensive domains are at the root, and the less precise yet cheaper are towards the leafs. Every domain is allowed to pull information from every domain, but only higher-rank domains can push information to lower-rank ones.

```
void ArrayCopy(int[] input, int[] output, int index)
{
  Contract.Requires(index >= 0);
  Contract.Requires(output.Length - index >= input.Length);

  for (var i = 0; i < input.Length; i++) output[i+index] = input[i];
}
```

Fig. 5. Simple example where fully fledged relational numerical domains are needed.

Subpolyhedra In the general setting of contract checking, arbitrary linear inequalities are needed for effective symbolic reasoning. For instance in the example in Fig. 5, one needs to infer the loop invariant $0 \leq \mathsf{sv}(\mathsf{i}) + \mathsf{sv}(\mathsf{index}) \leq \mathsf{sv}(\mathsf{output.Length})$.

Using the classical Polyhedra turned out to be far too expensive [17]. We are aware that many advances have been made to optimize them [1, 22], but we are still skeptical that they can scale up to the needs of Clousot's customers. Classical Polyhedra have a double representation for an abstract state: geometrical (where the Polyhedra is expressed as a set of points and generators) and algebraic (maintaining the tableau of equations defining the polyhedron). Some abstract operations are very efficient in one form, some in another. Converting from one form to its dual is very expensive (exponential) and it has been shown that it cannot be done faster [25]. Hence we developed a new abstract domain, Supolyhedra, which is as *expressive* as Polyhedra, but which gives away some of the inference power. The main, simple idea, is to split a linear inequality $\sum a_i \cdot \mathsf{x}_i \leq b$ into an equality and an interval via a slack variable β: $\sum a_i \cdot \mathsf{x}_i = \beta \wedge \beta \in [-\infty, b]$. Each of the two conjuncts is handled by a separate abstract domain, i.e., linear equalities and intervals. There are two main challenges here. The first one is to have a precise enough join, the pairwise join being simply to rough. The second one is to have an effective reduction algorithm to get the tightest bounds on the intervals. We have defined in [28] a join (and widening) operator which allow fine tuning the two points above, *de facto* defining a family of abstract domains, where the precision/cost ratio can be adjusted: more precise domains are obtained by improving the *hints* [26] at join/widening points and the reduction subroutine. In our tests Subpolyhedra scales to hundreds of variables, going well beyond the current state of the art of Polyhedra implementations.

5.3 Floating point values

We have an implementation of Intervals supporting the IEEE 754 standard. We have not yet extended this support to relational domains, as for instance [33, 5], so that the amount of reasoning that can be done on floats is very limited. We have an analysis to figure out possible precision mismatches in double comparisons caused by implicit conversions between 80 and 64 bits of precision. Such

```
private double balance;
public void Deposit(double amount)
{
  Contract.Requires(amount >= 0.0);
  Contract.Ensures(this.balance == Contract.OldValue(balance) + amount);

  balance = balance + amount;
}
```

Fig. 6. Example showing problems induced by the extra-precision for `double`s allowed by the ECMA standard. The field `balance` is stored into a 64 bits memory location whereas the result of `balance` + `amount` is stored into a 80 register.

conversions may introduce subtle bugs. This is best illustrated by the example in Fig. 6.

One may expect the postcondition to trivially hold. However, using an automatic test generation tool as *e.g.* PEX [39] one can easily find counterexamples to the postconditions! The ECMA standard [14] allows locals (including parameters) to be passed with the full precision of the architecture, whereas fields should *always* be truncated to 64 bit doubles. In an x86 architecture, double registers are 80 bits long. As a consequence, `amount` is passed as an 80 bit value, the result of `this.balance` + `amount` is stored in a CPU register (80 bits), but when written back to memory, it gets truncated to 64 bits. As a consequence the postcondition may be violated at runtime for specific values of `amount`. Clousot tracks floating point types of a symbolic values according to the flat lattice $\bot \sqsubseteq \texttt{Float}, \texttt{CPUFloat} \sqsubseteq \top$, $\texttt{Float} \neq \texttt{CPUFloat}$. In the example, Clousot infers $\texttt{balance} + \texttt{amount} : \texttt{CPUFloat}$ and $\texttt{balance} : \texttt{Float}$, and hence issues a warning for a possible precision mismatch. An explicit cast forces the truncation: the correct postcondition is hence `balance` == (`double`) (`Contract.OldValue`(`balance`)+`amount`).

5.4 Arrays and Collections

The abstract domains for scalar values are lifted to sequences (like arrays or collections) via a parametric segmentation functor [12]. The functor automatically and semantically divides (*e.g.*) arrays into sequences of consecutive non-overlapping possibly empty segments. Segments are delimited by sets of boundary expressions and abstracted uniformly. The overhead of the analysis is very low (around 1% on large framework libraries). Once again we developed a new (functor) abstract domain as existing solutions turned out either to require too much extra-assistance from the user [23] or to be inherently not-scalable [20, 21].

Example 5. At the end of the `for` loop of the (incorrect version of the) method `Push`, the array analysis associates the following two abstract elements to the arrays:

$$\begin{array}{ll} \texttt{arr} & \mapsto \{\texttt{0}\}\texttt{NN}\{\mathsf{sv(i)}, \mathsf{sv(nextFree)}, \mathsf{sv(arrLen)}\}? \\ \texttt{newArr} & \mapsto \{\texttt{0}\}\texttt{NN}\{\mathsf{sv(i)}, \mathsf{sv(nextFree)}\}?\texttt{N}\{\mathsf{sv(newArrLen)}?\} \end{array}$$

stating that all the elements of `newArr` up to `nextFree` are not-null, but also that $\mathsf{sv(i)} = \mathsf{sv(nextFree)}$ (expressions in bounds are equal) and that it may be the case $0 = \mathsf{sv(nextFree)} = \mathsf{sv(newArrLen)}$, in which case the `newArr` is empty (? denotes the fact that successive segments may be equal). □

Example 6. For the method `AllZero` of Fig. 3, at loop exit the analysis discover the invariant $\mathsf{a} \mapsto \{\mathsf{0}\}[\mathsf{0},\mathsf{0}]\{\mathsf{sv(i)}, \mathsf{sv(a.Length)}\}?$ which compactly represents $\forall j \in [0, \mathsf{a.Length}).\mathsf{a}[j] = 0 \land \mathsf{sv(i)} = \mathsf{sv(a.Length)} \land 0 \leq \mathsf{sv(i)}$.

6 Checking

Assertion Crawling The code of the method under analysis is crawled to collect a set of proof obligations $\mathcal{P}$. Proof obligations are either explicit or implicit. Explicit proof obligations are either: (i) preconditions at call sites; (ii) explicit assertions; or (iii) postcondition for the current method. Checking of explicit proof obligations is always on. Implicit proof obligations are induced by the CIL semantics. For a reference access r, a non null proof obligation $r \neq$ `null` is emitted. For an array creation with size `exp`, a proof obligation $0 \leq$ `exp` is emitted. For an array load or store with index `exp`, the two proof obligations $0 \leq$ `exp` and `exp` $<$ `svLen` are emitted. Similarly for buffer accesses, divisions, negation of minint, overflow checking and floating type mismatches. The checks for implicit proof obligations (such as non-null dereferencing and array bound checks) can be individually activated by the user. The rationale is to avoid drowning the user with too many warnings and instead have him/her first focus on the contracts.

Direct Checking For each proof obligation $\langle \mathtt{pc}, c\rangle \in \mathcal{P}$ Clousot individually asks each of the analyses if at program point `pc` the condition c holds. Each analysis implements a specialized decision procedure (in the numerical and the array analysis those specialized decision procedures are also invoked during the fixpoint computation to refine the analysis itself). The analysis fetches the abstract state at program point `pc`, and checks if it implies c. Fetching may cause a re-run of a part of the analysis, as for performance and memory considerations we only save abstract states at some specific program points (*e.g.* loop heads as in [4]). There are four possible check outcomes: true, meaning that c holds for all the possible executions *reaching* `pc`; false, meaning that there is no execution reaching `pc` such that c holds; $\bot$, meaning that the program point `pc` is unreached (dead code); and $\top$ meaning that the analysis does not have a definite answer. Direct checking $\langle \mathtt{pc}, c\rangle$ is aborted as soon as an outcome different from $\top$ is reported. This approach may fail to report the most precise answer, produced by the meet of all the analyses outcomes. We do so mainly for performance reasons (projects typically contain tenths of thousands of proof obligations to discharge).

```
string Nums(int a)
{
  Contract.Requires(a > 0);

  string s = null;
  var i = 0
  /* 1 */
  for (; i < a; i++) { s += i.ToString(); /* 2 */}

  /* 3 */
  Contract.Assert(s != null);

  return s;
}
```

Fig. 7. Example showing the combination of analyses via backward goal propagation. The NonNull analysis discovers that $\texttt{s}! = \texttt{null}$ at `2`, and the Numerical analysis discovers that the path $\texttt{1} \to \texttt{3}$ is unfeasible.

Domain Refinement If all the analyses had $\top$ as outcome, then Clousot refines the analysis. One first way of refining the analysis is to re-analyze the method body with a more precise abstract domain. Clousot implements an iterative strategy in which first less precise abstract domains are used (*e.g.* the numerical analysis instantiated with Pentagons) then moving to more precise yet expensive domains. In the worst case, one may always resort to the most expensive domain (*e.g* Subpolyhedra with all the hints on and the Simplex-based reduction [27]). Empirically we noticed that refinement pays off since the number of cases where one needs the most expensive domains is relatively small.

Goal directed backwards analysis If domain refinement is not good enough to discharge a proof obligation, we propagate the condition backwards. Essentially, the condition c is turned into an obligation for all the predecessor program points using weakest preconditions. We attempt to use the abstract state at those points to discharge the condition. This approach is good at handling disjunctive invariants which our abstract domains typically do not represent precisely. E.g., an assert after a join point may not be provable due to loss of precision at the join. However, the abstract states at the program points just prior to the join may be strong enough to discharge the obligation. This backwards analysis discharges an obligation if it can be discharged on *all* the paths leading to the assertion. It thus acts as a form of on-demand trace partitioning [36]. Furthermore, it also provide: (i) another way of modularly combining different analyses, as for instance one branch may be discharged by the non-null analysis and the other by the numerical analysis (the common case for implication-like conditions such as $\texttt{a} == \texttt{null} \mid\mid \texttt{a.Length} > 0$); and (ii) to lazily perform loop unrolling.

Example 7. Let us consider the code in Fig.7. Intuitively the assertion holds because the loop is executed at least once. At program point 3, the NonNull analysis infers $\mathsf{sv}(\mathsf{s}) = \mathtt{N} \sqcup \mathtt{NN} = \top$, and the numerical analysis infers $\mathsf{sv}(\mathsf{i}) = \mathsf{sv}(\mathsf{a}) \wedge \mathsf{sv}(\mathsf{i}) \in [1, +\infty]$. So the direct check cannot prove the assertion. The condition is pushed back to the predecessor program points, 1 corresponding to 0 executions of the loop, and 2 corresponding to > 0 loop iterations. At 2, we know that $\mathsf{sv}(\mathsf{s}) = \mathtt{NN}$ from the forward analysis, so this path can be discharged. At 1, we know that $\mathsf{sv}(\mathsf{i}) = 0$, but $\mathsf{sv}(\mathsf{i}) > 0$ at 3 from the forward analysis, hence a contradiction, so the path $1 \to 3$ is unfeasible, and the condition can be discharged. □

7 Contract Inference

To help the programmer get started with the CodeContracts, Clousot performs some amount of inference, which is either suggested to the user as missing contracts or silently propagated.

Precondition inference When a proof obligation cannot be discharged with any of the methods sketched above, Clousot checks if *all* the variables appearing in the condition: (i) existed in the pre-state of the method; and (ii) are unmodified. In this case it suggests a possible precondition. For instance in the example of Fig. 5, Clousot will suggest the two preconditions `input` $! =$ `null` and `output` $! =$ `null`. The precondition is only *suggested* and not inferred as it may be wrong. In the same example, suppose that the code

```
if(input == null) return;
```

was added before the loop, then Clousot would still have suggested `output` $! =$ `null` as precondition, but it would be incorrect, as `output` can perfectly be `null` when `input` is `null`. We have a better and correct solution for the precondition inference problem [11], but have yet to implement it at the time of writing.

Postcondition inference Theoretically the postcondition inference problem is simply the projection of the abstract state(s) at the method return point. In practice one must also consider two facts: (i) avoid repeating postconditions already provided by the user; and (ii) produce a minimal set of postconditions. Our postcondition inference algorithm works as follows. First, ask all the analyses to provide known facts at the method return point. Facts should be serialized as Boolean expressions. Second, sort the Boolean expressions according to some heuristic (*e.g.* equalities are more interesting than inequalities). Call the result S. Third, create a product abstract state R abstracting the method postcondition. Fourth, for each fact $s \in S$, check if it is implied by R. If it is not, output s as a postcondition, and assume s in R. The algorithm produces a set of postconditions which fulfills the two requirements above.

Readonly field invariant inference We have prototyped a static analysis to infer object invariants on readonly fields based on [29].

8 Practical considerations

To make Clousot practical, we have engineered several solution to improve the user experience.

Adaptive analysis, timeouts We spent a considerable amount of time profiling and optimizing Clousot. However, there are corner cases in which a method analysis can take too long. Single methods can present complex control flow with a lot of join points (several thousands for a single method) or several nested loops causing the fixpoint computation to converge too slowly, in particular with relational domains. We have implemented an adaptive analysis, which tries to figure out if the method to analyze is too complex, in which case it analyzes it with cheaper abstract domains. Orthogonally, the fixpoint computation can be aborted when a certain timeout is reached (by default 10 seconds).

Message prioritization Clousot has heuristics for sorting the warning messages, trying to report the more relevant ones first. The heuristics assign an initial score I_P to each warning depending on the proof obligation ($P \in \{$`Precondition`, `Postcondition`, `Invariant`, `Assert`, `NonNullobligation`$\dots)$. The initial score is corrected with a reward ρ for the outcome ($\rho(\texttt{False}) > \rho(\bot) > \rho(\top) \geq 1$, and a penalty δ on the variables in the condition ($\delta(\texttt{Param}) > \delta(\texttt{Field}) > \delta(\texttt{Local}) \geq 1$). Intuitively, a warning on a condition with only locals (where all the information should be known to Clousot) is more likely to be a bug than one on a condition containing only references to parameters (for instance, the code may be missing a precondition). Eventually, a proof obligation of type P, with condition `C` and outcome O is prioritized according the formula $I_P \cdot \rho(O)/(\sum_{\texttt{v}\in\texttt{Vars(C)}} \delta(\texttt{v}))$.

Dealing with false positives There are two main reasons for which Clousot reports a false warning: (i) it does not know some external fact (for instance some third-party library methods returns a non-null value); (ii) it is incomplete (as all the static analyses). The user can help Clousot by adding an explicit assumption via `Contract.Assume`.

Clousot will simply believe the condition, and it will not try to check it statically. The condition can be checked at runtime (it behaves as a normal assertion). With the time, assumptions may grow very large in the codebase. Clousot can be instructed to find duplicated assumptions (essentially Clousot tries to prove the assumption, and if it succeeds reports it to the user, otherwise it silently moves on).

Example 8. Let us consider the code snippet if Fig. 8, abstracting the common case of an application using a third-party library without contracts (yet). Without any contract on `GetSomeString`, Clousot will issue a warning for a possible

```
var str = ThirdPartyLibrary.GetSomeString();

Contract.Assume(str != null);

/* Without the assumption, Clousot complains str may be null */
if(str.Length > 10) { ... }
```

Fig. 8. Example of using `Assume` to shut off a warning caused by a missing postcondition on third-party code.

null deference. The programmer, after reading the documentation, convinced himself that the method will never return `null`, and hence decided to add the assumption, hence documenting the fact that the warning has been reviewed, and classified as a false warning. Clousot will then assume it, and it will not issue the warning anymore. When the author of `ThirdPartyLibrary` releases a new version of its library with contracts, then Clousot will inform the user that the assumption is no longer needed. □

If the assumption is not enough to shut off the warning, then the user can mask it via the `SuppressMessage` attribute. This is normally the case when a contract is far beyond what Clousot can understand (for instance it involves several quantifiers). Furthermore, the user can focus the analysis on a particular type or method via the `ContractVerification` attribute.

Visual Studio integration and Analysis Caching Clousot is fully integrated into Visual Studio. In a normal run, it runs as a post-build step. Running synchronously the whole analysis at every build may decrease the user experience. As a consequence we have implemented a caching mechanism to re-analyze a small subset of the code that changed between two builds. Orthogonally, the user can make the verification process more interactive by using the "analyze this" feature, which runs the analysis only on the particular method or class under the mouse pointer.

9 Conclusions

We presented an overview of Clousot, a static checked for CodeContracts. Clousot analyzes annotated programs to infer facts (including loop invariants), and it uses this information to discharge proof obligations. Unlike similar tools, it is based on abstract interpretation and focused on specific properties of interest. Advantages include more determinism, (tunable) performance and automation. Clousot is distributed with the CodeContracts tools, available for downloading with academic license at `http://research.microsoft.com/en-us/projects/contracts/`. So far, we have had positive feedback from our users. Still there is much work to do, like increasing the expressivity of the heap analysis, adding abstract domains for strings and bit vectors, improving the inter-method inference, and facilitating the annotation process of legacy codebases.

Acknowledgments We'd like to thank Mike Barnett and all those who contributed to Clousot: Patrick and Radhia Cousot, Pietro Ferrara, Jacques-Henri Jourdan, Vincent Laviron and Michäel Monerau.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. Applications of polyhedral computations to the analysis and verification of hardware and software systems. *Theor. Comput. Sci.*, 410(46), 2009.
2. M. Barnett, B.-Y.E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05*.
3. G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. Jack - a tool for validation of security and behaviour of java applications. In *FMCO'06*.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI'03*.
5. L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *APLAS'08*, 2008.
6. R. Clarisó and J. Cortadella. The octahedron abstract domain. In *SAS'04*.
7. P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP'92*.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *ACM POPL'79*.
9. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252. ACM Press, 1977.
10. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the astrée static analyzer. In *ASIAN'06*.
11. P. Cousot, R. Cousot, and F. Logozzo. Contract precondition inference from intermittent assertions on collections. Technical Report MSR-TR-2010-117, Microsoft Research, 2010.
12. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. Technical Report MSR-TR-2009-194, Microsoft Research, 2010.
13. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM POPL '78*.
14. ECMA. Standard ECMA-355, Common Language Infrastructure, June 2006.
15. M. Fähndrich, M. Barnett, and F. Logozzo. Code Contracts, March 2009.
16. M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *ACM SAC'10*, 2010.
17. P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code in .NET. In *OOPSLA'08*. ACM Press, 2008.
18. J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV'07*.
19. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI'02*.

20. D. Gopan, T.W.. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *32nd POPL*, pages 338–350. ACM Press, 2005.
21. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *35th POPL*, pages 235–246. ACM Press, 2008.
22. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV'09*.
23. R. Jhala and K.L. McMillan. Array abstractions from proofs. In *CAV'07*, LNCS 4590, pages 193–206. Springer, 2007.
24. Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6, 1976.
25. L. Khachiyan, E. Boros, E. Borys, K. M. Elbassioni, and V. Gurvich. Generating all vertices of a polyhedron is hard. *Discrete & Computational Geometry*, 39(1-3):174–190, 2008.
26. V. Laviron and F. Logozzo. Refining abstract interpretation-based static analyses with hints. In *APLAS'09*.
27. V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI '09*.
28. V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI'09*.
29. F. Logozzo. *Modular static analysis of object-oriented languages*. Thèse de doctorat en informatique, École polytechnique, 2004.
30. F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *CC'08*.
31. F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *ACM SAC'08*.
32. A. Miné. A few graph-based relational numerical abstract domains. In *SAS'02*.
33. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*.
34. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.
35. A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI'06*, 2006.
36. X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.
37. S. Sankaranarayanan, f. Ivancic, and A. Gupta. Program analysis using symbolic ranges. In *SAS'07*.
38. A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In *LOPSTR'02*.
39. N. Tillmann and J. de Halleux. Pex-white box test generation for .net. In *TAP'08*.