Abstract Interpretation Based Program Testing

Patrick Cousot and Radhia Cousot

Département d'informatique
École normale supérieure
45 rue d'Ulm
75230 Paris cedex 05, France
Patrick.Cousot@ens.fr
http://www.di.ens.fr/~cousot

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

Abstract— **Every one can daily experiment that programs are bugged. Software bugs can have severe if not catastrophic consequences in computer-based safety critical applications. This impels the development of formal methods, whether manual, computer-assisted or automatic, for verifying that a program satisfies a specification. Among the automatic formal methods, program static analysis can be used to check for the absence of run-time errors. In this case the specification is provided by the semantics of the programming language in which the program is written. Abstract interpretation provides a formal theory for approximating this semantics, which leads to completely automated tools where run-time bugs can be statically and safely classified as unreachable, certain, impossible or potential. We discuss the extension of these techniques to *abstract testing* where specifications are provided by the programmers. Abstract testing is compared to program debugging and model-checking.**

Keywords— **Abstract interpretation, Model-checking, Debugging, Abstraction, Testing, Abstract testing.**

I. Introduction

Software debugging represents a large proportion of the software development and maintenance cost (from 25% up to 70% for safety critical software). Beyond classical debugging methods, code review, simulation, etc., new formal methods have been investigated during this past decade such as:

deductive methods : to prove that a program semantics satisfies a user-provided specification (using automatic theorem provers or interactive proof checkers);
model-checking : to check that a (finite) model of the program satisfies a user-provided specification (using exhaustive or symbolic state exploration techniques);
static analysis : to verify that no program execution can lead to run-time errors as specified by the programming language semantics (by computation of an abstract interpretation of the semantics of the program).

We investigate *abstract testing*, an extension of program static analysis aiming at verifying user-provided specifications by abstract interpretation of the program semantics. The technique is a natural extension of program debugging using program properties instead of values. We compare abstract testing to program debugging and model checking.

II. An informal introduction to abstract testing

Abstract testing is the verification that the abstract semantics of a program satisfies an abstract specification. The origin is the abstract interpretation based static checking of safety properties [1], [2] such as array bound checking and the absence of run-time errors which was extended to liveness properties such as termination [3], [4].

Consider for example, the factorial program (the random assignment `?` is equivalent to the reading of an input value or the passing of an unknown but initialized parameter value):

```
# ITlra.analysis ();;
Reachability/ancestry analysis for initial/final states;
Type the program to analyze...
n := ?;
f := 1;
while (n <> 0) do
   f := (f * n);
   n := (n - 1)
od;;
```

The automatic analysis of this factorial program [5], [6] leads to the result below. Each program point has been numbered and a corresponding local invariant (given between parentheses) provides the possible values of the variables when reaching that program point. The value , typed `_0_`, denotes the uninitialized value. Otherwise an interval of possible values is given for integer variables (with $+\infty$ (respectively $-\infty$) typed `+oo` (resp. `-oo`) denoting the greatest (resp. smallest) machine representable integer). The result of the automatic analysis is the following:

```
0: { n:_0_; f:_0_ }
  n := ?;
1:!: { n:[0,+oo]; f:_0_ }
  f := 1;
2: { n:[0,+oo]; f:[1,+oo] }
  while ((n < 0) | (0 < n)) do
    3: { n:[1,+oo]; f:[1,+oo] }
      f := (f * n);
    4: { n:[1,+oo]; f:[1,+oo] }
      n := (n - 1)
    5: { n:[0,1073741822]; f:[1,+oo] }
  od {(n = 0)}
6: { n:[0,0]; f:[1,+oo] }
```

The analysis automatically discovers the condition $\mathtt{n} \geq 0$ which should be checked at program point `1` (as indicated

by :!:), since otherwise a runtime error or nontermination is inevitable. Then the computed invariants will always hold. For example the final value of n is 0 whereas f $\geq$ 1. The analysis is performed for the specific value $+\infty = 1073741823$ whence $n \in [0, 1073741822]$ at program point 5 since otherwise there would have been an overflow so that execution would have stopped after that runtime error.

III. A formalization of abstract testing

Let $\langle S,\ t,\ I,\ F,\ E\rangle$ be a transition system with set of states S, transition relation $t \subseteq (S \times S)$, initial states $I \subseteq S$, erroneous states $E \subseteq S$, and final states $F \subseteq S$. The transition system is assumed to be generated by a small step operational semantics [7].

Example 1 In the factorial example of Sec. II, the states are triples $\langle p, n, f\rangle$ where $p \in \{0, \ldots, 6\}$ is a program point, $n, f \in \mathbb{Z} \cup \{\}$ are the respective values of variables n and f. The initial states are $I = \{\langle 0, , \rangle\}$. The final states are $F = \{\langle 6,\ n,\ f\rangle \mid n, f \in \mathbb{Z} \cup \{\}\}$. The erroneous states are $E = \{\langle p,\ n,\ f\rangle \mid n \notin [-\infty, +\infty] \vee f \notin [-\infty, +\infty]\}$, that is the value of the integer variables is out of bounds. Writing $s \xrightarrow{t} s'$ for $\langle s,\ s'\rangle \in t$, we have:

$$
\begin{array}{lcll}
\langle 0,\ n,\ f\rangle & \xrightarrow{t} & \langle 1,\ z,\ f\rangle, & z \in \mathbb{Z} \\
\langle 1,\ n,\ f\rangle & \xrightarrow{t} & \langle 2,\ n,\ 1\rangle & \\
\langle 2,\ n,\ f\rangle & \xrightarrow{t} & \langle 3,\ n,\ f\rangle & \text{if n} \in \mathbb{Z} \setminus \{0\} \\
\langle 2,\ 0,\ f\rangle & \xrightarrow{t} & \langle 6,\ 0,\ f\rangle & \\
\langle 3,\ n,\ f\rangle & \xrightarrow{t} & \langle 4,\ n,\ f.n\rangle & \\
\langle 4,\ n,\ f\rangle & \xrightarrow{t} & \langle 5,\ n-1,\ f\rangle & \\
\langle 5,\ n,\ f\rangle & \xrightarrow{t} & \langle 2,\ n,\ f\rangle &
\end{array}
$$

A program execution is a finite or infinite sequence $\sigma_0 \ldots \sigma_i \ldots$ of states $\sigma_i \in S$. Execution starts with an initial state $\sigma_0 \in I$. Any state σ_i is related to its successor state σ_{i+1} as specified by the transition relation t so that $\langle \sigma_i,\ \sigma_{i+1}\rangle \in t$. The sequence is finite $\sigma_0 \ldots \sigma_i \ldots \sigma_n$ if and only if the last state is erroneous $\sigma_n \in E$ (because of an anomaly during execution) or final $\sigma_n \in F$ (because of normal termination). All other states have a successor in which case execution goes on normally, may be for ever (formally $\forall s \in S \setminus (E \cup F) : \exists s' \in S : \langle s,\ s'\rangle \in t$).

Let t^{-1} be the inverse of relation t. Let $t^\star$ be the reflexive transitive closure of the binary relation t. Let $\mathrm{post}[t]\, X$ be the post-image of X by t, that is the set of states which are reachable from a state of X by a transition t: $\mathrm{post}[t]\, X \stackrel{\text{def}}{=} \{s' \in S \mid \exists s \in X : \langle s, s'\rangle \in t\}$ [2], [4]. Inversely, let $\mathrm{pre}[t]\, X \stackrel{\text{def}}{=} \mathrm{post}[t^{-1}]\, X$ be the pre-image of X by t that is the set of states from which there exists a possible transition t to a state of X: $\mathrm{pre}[t]\, X = \{s \in S \mid \exists s' \in X : \langle s,\ s'\rangle \in t\}$. The specifications considered in [3] are of the form:

$$\mathrm{post}[t^\star]\, I \implies (\neg E) \wedge \mathrm{pre}[t^\star]\, F\ .$$

Informally such a specification states that the descendants of the initial states are never erroneous and can potentially lead to final states.

By choosing different user specified invariant assertions Iv for $(\neg E)$ and intermittent assertions It for F, these forms of specification were slightly extended by [8] under the name "abstract debugging" to:

$$\mathrm{post}[t^\star]\, I \implies \mathrm{Iv} \wedge \mathrm{pre}[\mathrm{t}^\star]\, \mathrm{It}\ .$$

If the states $\langle p,\ m\rangle \in S$ consist of a program point $p \in P$ and a memory state $m \in M$ then the user can specify local invariant assertions $\mathrm{Iv_p}$ attached to program points $p \in \mathrm{Pv} \subseteq \mathrm{P}$ and local intermittent assertions $\mathrm{It_p}$ attached to program points $p \in \mathrm{Pt}$ so that

$$
\begin{array}{rcl}
\mathrm{Iv} & = & \{\langle p,\ m\rangle \mid p \in \mathrm{Pv} \implies \mathrm{Iv_p(m)}\} \\
\text{and} \quad \mathrm{It} & = & \{\langle p,\ m\rangle \mid p \in \mathrm{Pt} \wedge \mathrm{It_p(m)}\}\ .
\end{array}
$$

Otherwise stated, the descendants of the initial states always satisfy all local invariant assertions (which always holds) and can potentially lead to states satisfying some local intermittent assertion (which will sometime hold). For example, a specification that the above factorial program should always terminate normally states that any execution should always reach program point 6 hence would consist in choosing:

$$
\begin{array}{rcll}
\mathrm{Iv_p(n, f)} & = & \mathtt{n, f} \in [-\infty, +\infty], & p = \mathtt{1}, \ldots, \mathtt{6}; \\
\mathrm{It_p(n, f)} & = & \mathit{false}, & p = \mathtt{1}, \ldots, \mathtt{5}; \\
\mathrm{It_6(n, f)} & = & \mathit{true}\ . &
\end{array}
$$

The termination requirement can be very simply specified as comments in the program text:

```
0: n := ?;
1: f := 1;
2: while ((n < 0) | (0 < n)) do
    3: f := (f * n);
    4: n := (n - 1)
5: od;
6: sometime true;;
```

or by using an external temporal specification such as $\diamondsuit\, at\, 6$ (we use the temporal operators $\square\, P$ (read always P) to denote the set of sequences of states such that all states satisfy P, $\diamondsuit\, P$ (read sometime P) to denote the set of sequences containing at least one state satisfying P and the predicate $at\, p$ holds in all states which control point is p).

A program static analyzer can therefore be used for *abstract testing* which is similar to testing/debugging, with some essential differences such as the consideration of an abstract semantics instead of a concrete one, the ability to consider several (reversed) executions at a time (as specified by user initial and final state specifications), the use of forward and backward reasonings, the formal specification of what has to be tested, etc.

IV. Model-checking of temporal specifications

At first sight, abstract testing is model-checking [9], [10] of the temporal formula:

$$\square(\bigwedge_{p \in \mathrm{Pv}} at_p \implies \mathrm{Iv_p}) \wedge \diamondsuit(\bigvee_{\mathrm{p} \in \mathrm{Pt}} at_p \wedge \mathrm{It_p})$$

for a small-step operational semantics $\langle S,\ t,\ I\rangle$ of the program, or more precisely, abstract model-checking [11], [12] since abstract interpretation is involved.

Indeed model-checking and abstract testing are formal verification techniques which enjoy remarkable common advantages, the most important ones being that they are both fully automatic and both involve reasonings that are close to tracing program execution whence are easily understandable by programmers. However abstract testing is quite different both from program debugging and (abstract) model-checking for the technical reasons explained in the following sections.

V. Scope of application

A. Scope of abstract testing

The abstract interpreters are developed for programming languages that is infinitely many programs, with modular and infinitary recursive control and data structures which are difficult to abstract and are most often ignored in model-checking (with peculiar exceptions involving complete abstractions, such as e.g. [13]).

In order to apply abstract testing to a great variety of programming languages, abstract interpreter generators have been developed (see e.g. [14]).

The (generated) abstract interpreters are generic [14], [15], [16], [17], that is parameterized by an abstract domain specifying the considered approximated properties.

The advantage is that the user can choose the approximation of the semantics of the program which is considered for the abstract testing of the program among a selection of predesigned reusable choices. There is no need for the user to manually design the abstract interpreter. A consequence of this generality is that there is no easy fine tuning of the abstract interpreter for a particular specification and a particular program (abstract compilation, see e.g. [18], improving mainly the performance rather than the precision of the analyses).

B. Scope of (abstract) model checking

(Abstract) "model checking is a technique for verifying finite-state concurrent systems such as sequential circuit design and communication protocols" [19]. Indeed many model checking publications refer to the case study of a particular concurrent system which is often debugged and sometimes verified by using an existing model checker on an abstract model of the concurrent system. The particular, often implicit, abstraction which is used to design the model can be specifically developed for the considered concurrent system, see e.g. [20], [21], [22]. However these abstractions developed for a specific program and a specific specification of that program are not reusable hence extremely expansive to design.

In a sense this approach should always succeed since tuning the abstraction for a particular specification of a particular transition system is always complete (see [23]). However, the proper abstraction may be quite difficult to find in practice [24], [25] and may require a lot of efforts from the user sometimes amounting to a full manual proof.

VI. Abstract semantics

The only abstractions considered in abstract model checking [26] are state based abstractions $\wp(S) \mapsto \wp(S^\sharp)$ of the form $\alpha(X) = \{\alpha(s) \mid s \in X\}$ for a given state abstraction $\alpha \in S \mapsto S^\sharp$, see [27, sec. 14, p. 23]. This restriction follows from the requirement in abstract model-checking to model-check the abstract semantics which, in order to be able to reuse existing model-checkers, must have the form of a transition system on (abstract) states.

Contrary to a common believe not all abstractions are of that form. So some abstract semantics (using e.g. the interval abstraction [1], [2] or the polyhedral abstraction [28]) are beyond the scope of abstract model checking. Some model checking publications use these abstractions or similar ones which are not state based, e.g. [29], [30], [31], [32], [33], [34]. But then they use abstract interpretation based techniques such as fixpoint approximation, widening/narrowing, etc. to check safety (mainly reachability) properties as considered in Secs. II and III.

VII. The need for infinite abstract domains

Infinite abstract domains are definitely needed in program analysis for precision (and sometimes efficiency or ease of programming of the program analyzer). The argument given in [23] uses reachability analysis with the attribute-independent interval domain [1], [2] for the family of programs of the form:

```
x := 0;
while (x < n) do
    x := (x + 1)
od;;
```

where `n` is a given integer constant. For example, for `n` = 100, we get:

```
0: { x:_0_ }
  x := 0;
1: { x:[0,100] }
  while (x < 100) do
    2: { x:[0,99] }
      x := (x + 1)
    3: { x:[1,100] }
  od {((100 < x) | (x = 100))}
4: { x:[100,100] }
```

It is easy to prove that for any $n > 0$, the analyzer will discover:

```
0: { x:_0_ }
  x := 0;
1: { x:[0,n] }
  while (x < n) do
    2: { x:[0,n - 1] }
      x := (x + 1)
    3: { x:[1,n] }
  od {((n < x) | (x = n))}
4: { x:[n,n] }
```

The argument is then as follows:
1. for any given `n` it is possible to find an abstract domain (here $\{\ , [0,\mathtt{n}]\ , [0,\mathtt{n}-1]\ , [1,\mathtt{n}]\ , [\mathtt{n},\mathtt{n}]\}$) and to redesign a corresponding program analyzer (and its correctness proof) so

that the above result can be computed by this specific analyzer for the specific abstract domain corresponding to this particular `n`.
More generally, once a reachability proof has been done (e.g. by hand!), the abstract finite domain is the set of predicates involved in this proof and the abstract interpreter is nothing but the finite encoding of Floyd-Naur-Hoare verification conditions restricted to this peculiar finite domain. In general it is impossible to discover this best-fit abstract domain by simple inspection of the program text [1].
2. Any single program analyzer being able to analyze the entire infinite family of programs must use an abstract domain containing the $\subseteq$-strictly increasing chain $[1, n]$, $n > 0$, hence an infinite abstract domain, as well as a widening, to cope with:

```
0: { x:_0_ }
  x := 0;
1: { x:[0,+oo] }
  while (0 < 1) do
    2: { x:[0,+oo] }
      x := (x + 1)
    3: { x:[1,+oo] }
  od {((1 < 0) | (0 = 1))}
4: { x:_|_ }
```

The per-example redesign of the program analyzer has been proposed in model-checking, including with a proof-check of its correctness [24], [25], [37], [38], but is hardly conceivable for program analysis (but maybe for large very popular programs on which a huge human investment is conceivable, such as MS Word [39]). Note that this is different from using abstract interpretation or model-checking to help a prover/proof checker to infer invariants [40], [41], [37] or to guide the automatic prover in its proof search [42].

VIII. Precise checking in the presence of approximations

More importantly, the algorithms involved in abstract testing are more precise than model-checking ones *in the presence of approximations*. These approximations, such as widenings [1], [2], can be simply ignored in model-checking of finite-state transition systems.

A. Fixpoint approximation check

A first illustration of the enhanced precision of program testing algorithms consists in considering a fixpoint approximation check $\mathrm{lfp}^{\sqsubseteq} F \sqsubseteq I$ where $\langle L, \sqsubseteq, \bot, \top, \sqsubseteq, \sqsupseteq\rangle$ is a complete lattice, $F \in L \xrightarrow{\text{mon}} L$ is monotonic and $\mathrm{lfp}^{\sqsubseteq} F$ is the $\sqsubseteq$-least fixpoint of F.

For example an invariance check (such as array bound checking) $\Box I$ consists in verifying that $\mathrm{lfp}^{\sqsubseteq} F \sqsubseteq I$ where $\mathrm{lfp}^{\sqsubseteq} F$ characterizes the set of descendants of the entry states and I is the invariant to be checked (asserting for example that array indexes are within the declared bounds). In this example, L is $\langle \wp(S), \subseteq, \emptyset, S, \subseteq, \supseteq\rangle$, $F = \boldsymbol{\lambda}\, X \cdot E \cup \mathrm{post}[t]\, X$ so that $\mathrm{lfp}^{\sqsubseteq} F = \mathrm{post}[t^\star]\, E$.

In (abstract) model-checking, one computes iteratively $\mathrm{lfp}^{\sqsubseteq} F$ and then checks that $\mathrm{lfp}^{\sqsubseteq} F \sqsubseteq I$ (or uses a strictly equivalent check, see [43, p. 73] and Sec. X below).

In abstract testing, one computes iteratively an upper-approximation J of $\mathrm{lfp}^{\sqsubseteq} \boldsymbol{\lambda}\, X \cdot I \sqcap F(X)$ with acceleration of the convergence of the iterates by widening/narrowing [4], [1], [2]. The convergence criterion is:

$$(I \sqcap F(J)) \sqsubseteq J \ . \tag{1}$$

Then the invariance check has the form:

$$F(J) \sqsubseteq I \ . \tag{2}$$

This is sound, by the following theorem:

Theorem 1 *If $\langle L, \sqsubseteq, \bot, \top, \sqsubseteq, \sqsupseteq\rangle$ is a complete lattice, $F \in L \xrightarrow{\text{mon}} L$ is monotonic and $I, J \in L$, then:*

$$(I \sqcap F(J)) \sqsubseteq J \wedge F(J) \sqsubseteq I \implies \mathrm{lfp}^{\sqsubseteq} F \sqsubseteq I$$

Proof: We have $F(J) = F(J) \sqcap F(J) \sqsubseteq I \sqcap F(J)$ [by (2)] $\sqsubseteq J$ [by (1)] proving $F(J) \sqsubseteq J$ by transitivity whence $\mathrm{lfp}^{\sqsubseteq} F \sqsubseteq J$ by Tarski's fixpoint theorem [44], [45]. By definition of fixpoints and monotony, it follows that $\mathrm{lfp}^{\sqsubseteq} F = F(\mathrm{lfp}^{\sqsubseteq} F) \sqsubseteq F(J) \sqsubseteq I$ [by (2)]. By transitivity, we conclude $\mathrm{lfp}^{\sqsubseteq} F \sqsubseteq I$ as required. ∎

The reason why abstract testing uses more involved computations is that in the context of infinite state systems, and for a given abstraction, the approximation of the more complex expression is in general more precise than the abstraction of the trivial expression. Consider for example interval analysis [1], [2] of the simple loop accessing sequentially an array $A[1]$, ..., $A[100]$:

```
# IT.analysis ();;
Forward analysis from initial states;
Type the program to analyze...
i := 0;
while (i <> 100) do
    i := (i + 1);
    skip % array access %
od;;
```

The result of the analysis [6] is too approximate to statically check that the index `i` is within the array bounds 1 and 100 :

```
0: { i:_0_ }
  i := 0;
1: { i:[0,+oo] }
  while ((i < 100) | (100 < i)) do
    2: { i:[0,+oo] }
      i := (i + 1);
    3: { i:[1,+oo] }
      skip
    4: { i:[1,+oo] }
  od {(i = 100)}
5: { i:[100,100] }
```

However by explicit conjunction with the array access invariant $0 < \mathtt{i} \leq 100$ (the evaluation of the runtime check `always B` has the effect of blocking the program execution when the assertion `B` does not hold):

[1] Just as the invariants in Floyd-Naur-Hoare proof method are not trivial to discover. From a practical point of view, compare the empiric approach of [35] based on heuristics for discovering invariants from the program test which leads to worse results than [1], as shown in [36].

```
# IT.analysis ();;
Forward analysis from initial states;
Type the program to analyze...
i:=0;
while i <> 100 do
  i := i + 1;
  always (0 < i) & (i <= 100)
od;;
```

the static analysis now proves that the array out of bound error is impossible:

```
0: { i:_0_ }
  i := 0;
1: { i:[0,100] }
  while ((i < 100) | (100 < i)) do
    2: { i:[0,99] }
      i := (i + 1);
    3: { i:[1,100] }
      always ((0 < i) & ((i < 100) | (i = 100)))
    4: { i:[1,100] }
  od {(i = 100)}
5: { i:[100,100] }
```

Experimentally, acceleration of the convergence may even lead to a faster convergence of the more precise analysis.

B. Fixpoint meet approximation

A second illustration of the difference between model-checking and abstract testing algorithms is the upper-approximation of the descendants of the initial states which are ancestors of the final states. A model-checking algorithm (such as [46]) computes a conjunction of forward and backward fixpoints. The forward analysis of the factorial program:

```
# IT.analysis ();;
Forward analysis from initial states;
Type the program to analyze...
n := ?;
f := 1;
while (n <> 0) do
   f := (f * n);
   n := (n - 1)
od;;
```

yields:

```
0: { n:_0_; f:_0_ }
  n := ?;
1: { n:[-oo,+oo]; f:_0_ }
  f := 1;
2: { n:[-oo,+oo]; f:[-oo,+oo] }
  while ((n < 0) | (0 < n)) do
    3: { n:[-oo,+oo]; f:[-oo,+oo] }
      f := (f * n);
    4: { n:[-oo,+oo]; f:[-oo,+oo] }
      n := (n - 1)
    5: { n:[-oo,1073741822]; f:[-oo,+oo] }
  od {(n = 0)}
6: { n:[0,0]; f:[-oo,+oo] }
```

The backward analysis of the factorial program:

```
# IT_1.analysis ();;
Backward analysis from final states;
Type the  program to analyze...
n := ?;
f := 1;
while (n <> 0) do
   f := (f * n);
   n := (n - 1)
od;;
```

yields:

```
0: { n:[-oo,+oo]?; f:[-oo,+oo]? }
  n := ?;
1: { n:[0,+oo]; f:[-oo,+oo]? }
  f := 1;
2: { n:[0,+oo]; f:[-oo,+oo]? }
  while ((n < 0) | (0 < n)) do
    3: { n:[1,+oo]; f:[-oo,+oo] }
      f := (f * n);
    4: { n:[1,+oo]; f:[-oo,+oo]? }
      n := (n - 1)
    5: { n:[0,+oo]; f:[-oo,+oo]? }
  od {(n = 0)}
6: { n:[-oo,+oo]?; f:[-oo,+oo]? }
```

The intersection is therefore:

```
0: { n:_0_; f:_0_ }
  n := ?;
1: { n:[-oo,+oo]; f:_0_ }
  f := 1;
2: { n:[0,+oo]; f:[-oo,+oo]? }
  while ((n < 0) | (0 < n)) do
    3: { n:[1,+oo]; f:[-oo,+oo] }
      f := (f * n);
    4: { n:[1,+oo]; f:[-oo,+oo] }
      n := (n - 1)
    5: { n:[0,1073741822]; f:[-oo,+oo] }
  od {(n = 0)}
6: { n:[0,0]; f:[-oo,+oo] }
```

Abstract testing iterates an alternation between forward and backward fixpoints [3], [36]. For the factorial program:

```
# ITlra.analysis ();;
Reachability/ancestry analysis for initial/final states;
Type the program to analyze...
n := ?;
f := 1;
while (n <> 0) do
   f := (f * n);
   n := (n - 1)
od;;
```

the analysis is more precise (since it can now derive that `f` is positive):

```
0: { n:_0_; f:_0_ }
  n := ?;
1:!: { n:[0,+oo]; f:_0_ }
  f := 1;
2: { n:[0,+oo]; f:[1,+oo] }
  while ((n < 0) | (0 < n)) do
    3: { n:[1,+oo]; f:[1,+oo] }
      f := (f * n);
    4: { n:[1,+oo]; f:[1,+oo] }
      n := (n - 1)
    5: { n:[0,1073741822]; f:[1,+oo] }
  od {(n = 0)}
6: { n:[0,0]; f:[1,+oo] }
```

Assume that we must approximate $\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B$ from above using an abstraction defined by the Galois connection $\langle L, \sqsubseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle L^\sharp, \sqsubseteq^\sharp\rangle$ that is by definition $\forall x \in L : \forall y \in L^\sharp : \alpha(x) \sqsubseteq^\sharp y \iff x \sqsubseteq \gamma(y)$. The intuition is that, in the concrete world L, any element $x \in L$ can be approximated by any x' such that $x \sqsubseteq x'$. In the abstract world $L^\sharp$, x can be approximated by any y such that $x \sqsubseteq \gamma(y)$. The best or more precise such abstract approximation is $y = \alpha(x)$. It is an abstract approximation since $x \sqsubseteq \gamma \circ \alpha(x)$. It is the more precise since for any other abstract approximation y, $x \sqsubseteq \gamma(y) \implies \gamma \circ \alpha(x) \sqsubseteq \gamma(y)$.

Now F and B can be approximated by their abstract interpretations $F^\sharp \sqsupseteq \alpha \circ F \circ \gamma$ of F and $B^\sharp \sqsupseteq \alpha \circ B \circ \gamma$ of B. A better approximation than $\mathrm{lfp}^{\sqsubseteq^\sharp} F^\sharp \sqcap^\sharp \mathrm{lfp}^{\sqsubseteq^\sharp} B^\sharp$ was suggested in [3]. It is calculated as the limit of the *alternating fixpoint computation*:

$$\dot{X}^0 = \mathrm{lfp}^{\sqsubseteq^\sharp} F^\sharp \text{ or } \mathrm{lfp}^{\sqsubseteq^\sharp} \mathrm{B}^\sharp \quad (3)$$

$$\dot{X}^{2n+1} = \mathrm{lfp}^{\sqsubseteq^\sharp} \boldsymbol{\lambda} X \cdot (\dot{X}^{2n} \sqcap^\sharp B^\sharp(X)), \quad n \in \mathbb{N} \quad (4)$$

$$\dot{X}^{2n+2} = \mathrm{lfp}^{\sqsubseteq^\sharp} \boldsymbol{\lambda} X \cdot (\dot{X}^{2n+1} \sqcap^\sharp F^\sharp(X)), \quad n \in \mathbb{N} \quad (5)$$

For soundness, we assume:

$$\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B = \mathrm{lfp}^{\sqsubseteq} \boldsymbol{\lambda} X \cdot (\mathrm{lfp}^{\sqsubseteq} F \sqcap B(X)) \quad (6)$$

$$= \mathrm{lfp}^{\sqsubseteq} \boldsymbol{\lambda} X \cdot (\mathrm{lfp}^{\sqsubseteq} B \sqcap F(X)) \quad (7)$$

$$= \mathrm{lfp}^{\sqsubseteq} \boldsymbol{\lambda} X \cdot (\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B \sqcap B(X)) \quad (8)$$

$$= \mathrm{lfp}^{\sqsubseteq} \boldsymbol{\lambda} X \cdot (\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B \sqcap F(X)) \quad (9)$$

so that there is no improvement when applying the alternating fixpoint computation to F and B (such as the exact collecting semantics). However, when considering approximations $F^\sharp$ of F and $B^\sharp$ of B, not all information can be collected in one pass. So the idea is to propagate the initial assertion forward so as to get a final assertion. This final assertion is then propagated backward to get stronger necessary conditions to be satisfied by the initial states for possible termination. This restricts the possible reachable states as indicated by the next forward pass. Going on this way, the available information on the descendant states of the initial states which are ascendant states of the final states can be improved on each successive pass, until convergence. A specific instance of this computation scheme was used independently by [47] to infer types in flowchart programs.

Let us recall the following classical results in abstract interpretation [2], [48]:

Theorem 2 (Fixpoint abstraction) *If* $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ *and* $\langle L^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp \rangle$ *are complete lattices,* $\langle L, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle L^\sharp, \sqsubseteq^\sharp \rangle$ *is a Galois connection, and* $F \in L \overset{\text{mon}}{\longmapsto} L$, *then* $\alpha(\mathrm{lfp}^{\sqsubseteq} F) \sqsubseteq \mathrm{lfp}^{\sqsubseteq^\sharp} \alpha \circ F \circ \gamma$. □

Proof: In a Galois connection, α and γ are monotonic, so by Tarski's fixpoint theorem [44], the least fixpoints exist. So let $P^\sharp \overset{\text{def}}{=} \mathrm{lfp}^{\sqsubseteq^\sharp} \alpha \circ F \circ \gamma$. We have $\alpha \circ F \circ \gamma(P^\sharp) = P^\sharp$ whence $F \circ \gamma(P^\sharp) \sqsubseteq \gamma(P^\sharp)$ by definition of Galois connections. It follows that $\gamma(P^\sharp)$ is a postfixpoint of F whence $\mathrm{lfp}^{\sqsubseteq} F \sqsubseteq \gamma(P^\sharp)$ by Tarski's fixpoint theorem or equivalently $\alpha(\mathrm{lfp}^{\sqsubseteq} F) \sqsubseteq^\sharp P^\sharp = \mathrm{lfp}^{\sqsubseteq^\sharp} \alpha \circ F \circ \gamma$. ■

Theorem 3 (Fixpoint approximation) *If* $\langle L^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp \rangle$ *is a complete lattice,* $F^\sharp, \bar{F}^\sharp \in L^\sharp \overset{\text{mon}}{\longmapsto} L^\sharp$, *and* $F^\sharp \sqsubseteq^\sharp \bar{F}^\sharp$ *pointwise, then* $\mathrm{lfp}^{\sqsubseteq^\sharp} F^\sharp \sqsubseteq^\sharp \mathrm{lfp}^{\sqsubseteq^\sharp} \bar{F}^\sharp$. □

Proof: We have $F^\sharp(\mathrm{lfp}^{\sqsubseteq^\sharp} \bar{F}^\sharp) \sqsubseteq^\sharp \bar{F}^\sharp(\mathrm{lfp}^{\sqsubseteq^\sharp} \bar{F}^\sharp) = \mathrm{lfp}^{\sqsubseteq^\sharp} \bar{F}^\sharp$ whence $\mathrm{lfp}^{\sqsubseteq^\sharp} F^\sharp \sqsubseteq^\sharp \mathrm{lfp}^{\sqsubseteq^\sharp} \bar{F}^\sharp$ since $\mathrm{lfp}^{\sqsubseteq^\sharp} F^\sharp = \bigsqcap^\sharp \{X \mid F^\sharp(X) \sqsubseteq^\sharp X\}$ by Tarski's fixpoint theorem [44]. ■

The correctness of the alternating fixpoint computation follows from the following:

Theorem 4 (Alternating fixpoint approximation) *If* $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ *and* $\langle L^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp \rangle$ *are complete lattices,* $\langle L, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle L^\sharp, \sqsubseteq^\sharp \rangle$ *is a Galois connection,* $F \in L \overset{\text{mon}}{\longmapsto} L$ *and* $B \in L \overset{\text{mon}}{\longmapsto} L$ *satisfy the hypotheses (8) and (9),* $F^\sharp \in L^\sharp \overset{\text{mon}}{\longmapsto} L^\sharp$, $B^\sharp \in L^\sharp \overset{\text{mon}}{\longmapsto} L^\sharp$, $\alpha \circ F \circ \gamma \sqsubseteq^\sharp F^\sharp$, $\alpha \circ B \circ \gamma \sqsubseteq^\sharp B^\sharp$ *and the sequence* $\langle \dot{X}^n, n \in \mathbb{N} \rangle$ *is defined by (3), (4) and (5) then* $\forall k \in \mathbb{N} : \alpha(\mathrm{lfp}^{\subseteq} F \sqcap \mathrm{lfp}^{\subseteq} B) \sqsubseteq^\sharp \dot{X}^{k+1} \sqsubseteq^\sharp \dot{X}^k$. □

Proof: Observe that by the fixpoint property, $\dot{X}^{2n+1} = \dot{X}^{2n} \sqcap^\sharp B^\sharp(\dot{X}^{2n+1})$ and $\dot{X}^{2n+2} = \dot{X}^{2n+1} \sqcap^\sharp F^\sharp(\dot{X}^{2n+2})$, hence $\dot{X}^{2n} \sqsubseteq^\sharp \dot{X}^{2n+1} \sqsubseteq^\sharp \dot{X}^{2n+2}$ since $\sqcap^\sharp$ is the greatest lower bound for $\sqsubseteq^\sharp$ so that $\dot{X}^k$, $k \in \mathbb{N}$ is a decreasing chain.

We have $\alpha(\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B) \sqsubseteq^\sharp \alpha(\mathrm{lfp}^{\sqsubseteq} F)$ since α is monotone and $\alpha(\mathrm{lfp}^{\sqsubseteq} F) \sqsubseteq^\sharp \mathrm{lfp}^{\sqsubseteq^\sharp} F^\sharp$ by 3, thus proving the proposition for $k = 0$.

Let us observe that $\alpha \circ F \circ \gamma \sqsubseteq^\sharp F^\sharp$ implies $F \circ \gamma \sqsubseteq \gamma \circ F^\sharp$ by definition of Galois connections so that in particular for an argument of the form $\alpha(X)$, $F \circ \gamma \circ \alpha \sqsubseteq \gamma \circ F^\sharp \circ \alpha$. In a Galois connection, $\gamma \circ \alpha$ is extensive so that by monotony and transitivity $F \sqsubseteq \gamma \circ F^\sharp \circ \alpha$.

Assume now by induction hypothesis that $\alpha(\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B) \sqsubseteq^\sharp \dot{X}^{2n}$, or equivalently, by definition of Galois connections, that $\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B \sqsubseteq \gamma(\dot{X}^{2n})$. Since $F \sqsubseteq \gamma \circ F^\sharp \circ \alpha$, it follows that $\boldsymbol{\lambda} X \cdot \mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B \sqcap F(X) \sqsubseteq \boldsymbol{\lambda} X \cdot \gamma(\dot{X}^{2n}) \sqcap \gamma \circ F^\sharp \circ \alpha(X) = \boldsymbol{\lambda} X \cdot \gamma(\dot{X}^{2n} \sqcap F^\sharp \circ \alpha(X))$ since, in a Galois connection, γ is a complete meet morphism. Now by hypothesis (8), we have $\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B = \mathrm{lfp} \boldsymbol{\lambda} X \cdot (\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B \sqcap F(X)) \sqsubseteq^\sharp \mathrm{lfp} \boldsymbol{\lambda} X \cdot \gamma(\dot{X}^{2n} \sqcap F^\sharp \circ \alpha(X))$ by Th. 3. Let G be $\boldsymbol{\lambda} X \cdot \dot{X}^{2n} \sqcap F^\sharp(X)$. In a Galois connection, $\alpha \circ \gamma$ is reductive so that by monotony $G \circ \alpha \circ \gamma \sqsubseteq^\sharp G$ and $\alpha \circ \gamma \circ G \circ \alpha \circ \gamma \sqsubseteq^\sharp G \circ \alpha \circ \gamma$, whence, by transitivity, $\alpha \circ \gamma \circ G \circ \alpha \circ \gamma \sqsubseteq^\sharp G$. By Th. 2, we have $\alpha(\mathrm{lfp} \gamma \circ G \circ \alpha) \sqsubseteq^\sharp \mathrm{lfp} \alpha \circ \gamma \circ G \circ \alpha \circ \gamma \sqsubseteq^\sharp \mathrm{lfp} G$ by Th. 3. Hence, $\mathrm{lfp} \boldsymbol{\lambda} X \cdot \gamma(\dot{X}^{2n} \sqcap F^\sharp \circ \alpha(X)) \sqsubseteq \gamma(\mathrm{lfp} \boldsymbol{\lambda} X \cdot \dot{X}^{2n} \sqcap F^\sharp(X))$ so that by transitivity we conclude that $\alpha(\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B) \sqsubseteq^\sharp \dot{X}^{2n+1}$.

The proof that $\alpha(\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B) \sqsubseteq^\sharp \dot{X}^{2n+2}$ is similar, using hypothesis (8) and by exchanging the rôles of F and B. ■

It is interesting to note that the computed sequence (3), (4) and (5) is optimal (see [45]).

If the abstract lattice does not satisfy the descending chain condition then [3] also suggests to use a narrowing operator Δ [1], [2] to enforce convergence of the downward iteration $\dot{X}^k, k \in \mathbb{N}$. The same way a widening/narrowing approach can be used to enforce convergence of the iterates for $\boldsymbol{\lambda} X \cdot \dot{X}^{2n} \sqcap F^\sharp(X)$ and $\boldsymbol{\lambda} X \cdot \dot{X}^{2n+1} \sqcap B^\sharp(X)$.

C. Local iterations

A third illustration of the difference between model-checking and abstract testing algorithms in the context of

approximation is the local iterations [49] to handle tests, backward assignments, etc. Below is an example of program static analysis, without local iterations:

```
# IT.analysis ();;
Forward analysis from initial states;
0: { x:_0_; y:_0_; z:_0_ }
  x := 0;
1: { x:[0,0]; y:_0_; z:_0_ }
  y := ?;
2: { x:[0,0]; y:[-oo,+oo]; z:_0_ }
  z := ?;
3: { x:[0,0]; y:[-oo,+oo]; z:[-oo,+oo] }
  if (((x = y) & (y = z)) & ((z + 1) = x)) then
    4: { x:[0,0]; y:[0,0]; z:[-1,-1] }
      skip
    5: { x:[0,0]; y:[0,0]; z:[-1,-1] }
  else
    6: { x:[0,0]; y:[-oo,+oo]; z:[-oo,+oo] }
      skip
    7: { x:[0,0]; y:[-oo,+oo]; z:[-oo,+oo] }
  fi
8: { x:[0,0]; y:[-oo,+oo]; z:[-oo,+oo] }
```

The precision of the same program with the same abstract domain is greatly enhanced with local iterations:

```
# IT'.analysis ();;
Forward reductive analysis from initial states;
0: { x:_0_; y:_0_; z:_0_ }
  x := 0;
1: { x:[0,0]; y:_0_; z:_0_ }
  y := ?;
2: { x:[0,0]; y:[-oo,+oo]; z:_0_ }
  z := ?;
3: { x:[0,0]; y:[-oo,+oo]; z:[-oo,+oo] }
  if (((x = y) & (y = z)) & ((z + 1) = x)) then
    4: { x:_|_; y:_|_; z:_|_ }
      skip
    5: { x:_|_; y:_|_; z:_|_ }
  else
    6: { x:[0,0]; y:[-oo,+oo]; z:[-oo,+oo] }
      skip
    7: { x:[0,0]; y:[-oo,+oo]; z:[-oo,+oo] }
  fi
8: { x:[0,0]; y:[-oo,+oo]; z:[-oo,+oo] }
```

When applied to tests without side-effects, the idea of the local iterations is to iterate the abstract evaluation of the test. From `{ x:[0,0]; y:[-oo,+oo]; z:[-oo,+oo] }`, the abstract interpretation of the test `(x = y)` yields `y:[0,0]`, the test `(y = z)` provides no information on `y` and `z` while `((z + 1) = x)` yields `z:[-1,-1]`. Iterating once more, the tests `(x = y)` and `((z + 1) = x)` provide no new information while `(y = z)` is false and so is the conjunction `(((x = y) & (y = z)) & ((z + 1) = x))`. It follows that program point 4 is not reachable which is denoted by assigning the bottom value $\bot$ (typed `_|_`) to variables.

D. Fixpoint meet approximation check

The abstract testing strategy to check $\text{post}[t^\star]\, I \implies \text{Iv} \wedge \text{pre}[t^\star]\, \text{It}$ and more generally $\text{lfp}^{\sqsubseteq} F \sqsubseteq I \sqcap \text{lfp}^{\sqsubseteq} B$ combines the results of Sec. VIII-A and Sec. VIII-B.

IX. Counter-examples to erroneous designs

Another important element of comparison between model-checking and abstract testing concerns the conclusions that can be drawn in case of failure of the automatic verification process. The model checking algorithms usually provide a counter-example [50]. This is not always possible with abstract testing (e.g. for non-termination) since the necessary over-approximation leads to the consideration of inexisting program executions which should not be proposed as counter-examples. This is the price to pay for undecidability.

However, abstract testing can provide necessary conditions for the specification to be (un-)satisfied. These automatically calculated conditions can serve as a guideline to discover the errors. They can also be checked at run-time to start the debugging mode before the error actually happens. For example the analysis of the following factorial program with a termination requirement:

```
# IT_1.analysis ();;
Backward analysis from final states;
Type the  program to analyze...
n := ?;
f := 1;
while (n <> 0) do
   f := (f * n);
   n := (n - 1)
od;;
```

leads to the necessary pre-condition textttn ≥ 0:

```
0: { n:[-oo,+oo]?; f:[-oo,+oo]? }
  n := ?;
1: { n:[0,+oo]; f:[-oo,+oo]? }
  f := 1;
2: { n:[0,+oo]; f:[-oo,+oo]? }
  while ((n < 0) | (0 < n)) do
    3: { n:[1,+oo]; f:[-oo,+oo] }
      f := (f * n);
    4: { n:[1,+oo]; f:[-oo,+oo]? }
      n := (n - 1)
    5: { n:[0,+oo]; f:[-oo,+oo]? }
  od {(n = 0)}
6: { n:[-oo,+oo]?; f:[-oo,+oo]? }
```

Indeed when this condition is not satisfied, i.e. when initially $\mathtt{n} < 0$, the program execution may not terminate or may terminate with a run-time error (arithmetic overflow in the above example). The following static analysis with this erroneous initial condition $\mathtt{n} < 0$:

```
#  IT.analysis ();;
Forward analysis from initial states;
Type the program to analyze...
initial n < 0;
f := 1;
while (n <> 0) do
   f := (f * n);
   n := (n - 1)
od;;
```

shows that the program execution never terminates properly so that the only remaining possible case is an incorrect termination with a run-time error ($\bot$, typed `_|_`, is the false invariant hence denotes unreachability in forward analysis and impossibility to reach the goal in backward analysis):

```
0: { n:_|_; f:_|_ }
  initial (n < 0);
1: { n:[-oo,-1]; f:_0_ }
  f := 1;
2: { n:[-oo,-1]; f:[-oo,1] }
  while ((n < 0) | (0 < n)) do
```

```
    3: { n:[-oo,-1]; f:[-oo,1] }
      f := (f * n);
    4: { n:[-oo,-1]; f:[-oo,0] }
      n := (n - 1)
    5: { n:[-oo,-2]; f:[-oo,0] }
  od {(n = 0)}
6: { n:_|_; f:_|_ }
```

Otherwise stated, infinitely many counter-examples are simultaneously provided by this counter-analysis.

X. Contrapositive reasoning

For the last element of comparison between abstract testing and model-checking, observe that in model-checking, using a set of states or its complement is equivalent as far as the precision of the result is concerned (but may be not its efficiency). For example, as observed in [43, p. 73], the Galois connection $\langle \wp(S), \subseteq \rangle \underset{\text{post}[r]}{\overset{\widetilde{\text{pre}}[r]}{\leftrightarrows}} \langle \wp(S), \subseteq \rangle$ (where $r \subseteq S \times S$ and $\widetilde{\text{pre}}[r]\, X \overset{\text{def}}{=} \{s \mid \forall s' : \langle s, s' \rangle \in r \implies s' \in X\}$) implies that the invariance specification check $\text{post}[t^\star]\, E \subseteq I$ is equivalent to $\widetilde{\text{pre}}[t^\star]\, \neg I \subseteq \neg E$ (or $\text{pre}[t^\star]\, \neg I \subseteq \neg E$ for total deterministic transition systems [4]). Otherwise stated a forward positive proof is equivalent to a backward contrapositive proof, as observed in [51]. So the difference between the abstract testing algorithm of [2], [48], [4] and the model-checking algorithm of [52], [53], [10] is that abstract testing checks $\text{post}[t^\star]\, E \subseteq I$ while model-checking verifies $\widetilde{\text{pre}}[t^\star]\, \neg I \subseteq \neg E$, which is equivalent for finite transition systems as considered in [52], [53], [10].

However, when considering infinite state systems the negation may be approximate in the abstract domain. For example the complement of an interval as considered in [1], [2] is not an interval in general. So the backward contrapositive checking may not yield the same conclusion as the forward positive checking. For example when looking for a pre-condition of an out of bounds error for the following program:

```
# IT_1.analysis ();;
Backward analysis from final states;
Type the  program to analyze...
i:=0;
while i <> 100 do
  i := i + 1;
  if (0 < i) & (i <= 100) then
    skip % array access %
  else
    final (i <= 0) | (100 < i) % out of bounds error %
  fi
od;;
```

the predicate `(i <= 0) | (100 < i)` cannot be precisely approximated with intervals, so the analysis is inconclusive:

```
0: { i:[-oo,+oo]? }
  i := 0;
1: { i:[-oo,1073741822] }
  while ((i < 100) | (100 < i)) do
    2: { i:[-oo,1073741822] }
      i := (i + 1);
    3: { i:[-oo,+oo] }
      if ((0 < i) & ((i < 100) | (i = 100))) then
        4: { i:[-oo,1073741822] }
          skip
        5: { i:[-oo,1073741822] }
      else {(((i < 0) | (0 = i)) | (100 < i))}
        6: { i:[-oo,+oo] }
          final (((i < 0) | (i = 0)) | (100 < i))
        7: { i:[-oo,1073741822] }
      fi
    8: { i:[-oo,1073741822] }
  od {(i = 100)}
9: { i:_|_ }
```

However both the forward positive and backward contrapositive checking may be conclusive. This is the case if we check for the lower bound only:

```
# IT_1.analysis ();;
Backward analysis from final states;
Type the  program to analyze...
i:=0;
while i <> 100 do
  i := i + 1;
  if (0 < i) then
    skip % array access %
  else
    final (i <= 0) % out of lower bound error %
  fi
od;;
```

This is shown below since the initial invariant is false so the out of lower bound error is unreachable:

```
0: { i:_|_ }
  i := 0;
1: { i:[-oo,-1] }
  while ((i < 100) | (100 < i)) do
    2: { i:[-oo,-1] }
      i := (i + 1);
    3: { i:[-oo,0] }
      if (0 < i) then
        4: { i:[-oo,-1] }
          skip
        5: { i:[-oo,-1] }
      else {((i < 0) | (0 = i))}
        6: { i:[-oo,0] }
          final ((i < 0) | (i = 0))
        7: { i:[-oo,-1] }
      fi
    8: { i:[-oo,-1] }
  od {(i = 100)}
9: { i:_|_ }
```

Similarly for the upper bound:

```
0: { i:_|_ }
  i := 0;
1: { i:[101,1073741822] }
  while ((i < 100) | (100 < i)) do
    2: { i:[100,1073741822] }
      i := (i + 1);
    3: { i:[101,+oo] }
      if ((i < 100) | (i = 100)) then
        4: { i:[101,1073741822] }
          skip
        5: { i:[101,1073741822] }
      else {(100 < i)}
        6: { i:[101,+oo] }
          final (100 < i)
        7: { i:[101,1073741822] }
      fi
    8: { i:[101,1073741822] }
  od {(i = 100)}
9: { i:_|_ }
```

Both analyzes could be done simultaneously by considering both intervals and their dual, or more generally finite disjunctions of intervals. More generally, completeness may

always be achieved by enriching the abstract domain [54]. To start with, the abstract domain might be enriched with complements [55], but this might not be sufficient and indeed the abstract domain might have to be enriched for each primitive operation [56], thus leading to an abstract algebra which might be quite difficult to implement if not totally inefficient.

XI. Conclusion

As an alternative to program debugging, formal methods have been developed to prove that a semantics or a model of the program satisfies a given specification. Because of theoretical and practical limitations, these formal methods have had more successes for finding bugs than for actual correctness proofs of full programs. For complex programs, the basic idea of complete program verification underlying the deductive and model checking methods must be abandoned in favor of debugging. In the context of debugging, we have shown that abstract interpretation based program static analysis can be extended to program testing. Abstract interpretation methods offer techniques which, in the presence of approximation, can be viable and powerful alternatives to both the exhaustive search of model-checking and the partial exploration methods of classical debugging. The main advantage is that no tuning of the abstract is needed since the program model is provided by approximation of its semantics chosen among a predefined set of wide-spectrum approximations hence avoiding the need to be designed by the user which ultimately amounts to a full proof.

References

[1] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the Second International Symposium on Programming*, pp. 106–130, Dunod, Paris, France, 1976. II, VI, VII, VIII, 1, VIII-A, VIII-A, VIII-B, X

[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), pp. 238–252, ACM Press, New York, New York, United States, 1977. II, III, VI, VII, VIII, VIII-A, VIII-A, VIII-B, VIII-B, X

[3] P. Cousot, *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*. Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, France, 21 March 1978. II, III, VIII-B, VIII-B

[4] P. Cousot, "Semantic foundations of program analysis," in *Program Flow Analysis: Theory and Applications* (S. Muchnick and N. Jones, eds.), ch. 10, pp. 303–342, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, United States, 1981. II, III, VIII-A, X

[5] P. Cousot, "The Marktoberdorf'98 generic abstract interpreter." http://www.di.ens.fr/~cousot/Marktoberdorf98.shtml, November 1998. II

[6] P. Cousot, "Calculational design of semantics and static analyzers by abstract interpretation." NATO International Summer School 1998 on Calculational System Design. Marktoberdorf, Germany. Organized by F.L. Bauer, M. Broy, E.W. Dijkstra, D. Gries and C.A.R. Hoare., 28 July – 9 August 1998. II, VIII-A

[7] G. Plotkin, "A structural approach to operational semantics," Tech. Rep. DAIMI FN-19, Aarhus University, Denmark, september 1981. III

[8] F. Bourdoncle, "Abstract debugging of higher-order imperative languages," in *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pp. 46–55, ACM Press, New York, New York, United States, 1993. III

[9] E. Clarke, E. Emerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 244–263, January 1986. IV

[10] J.-P. Queille and J. Sifakis, "Verification of concurrent systems in Cesar," in *Proceedings of the International Symposium on Programming*, Lecture Notes in Computer Science 137, pp. 337–351, Springer-Verlag, Berlin, Germany, 1982. IV, X

[11] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1512–1542, september 1994. IV

[12] R. Cleaveland, P. Iyer, and D. Yankelevitch, "Optimality in abstractions of model checking," in *Proceedings of the Second International Symposium on Static Analysis, SAS '95* (A. Mycroft, ed.), Glasgow, United Kindom, 25–27 september 1995, Lecture Notes in Computer Science 983, pp. 51–63, Springer-Verlag, Berlin, Germany, 1995. IV

[13] B. Boigelot and P. Godefroid, "Symbolic verification of communication protocols with infinite state spaces using QDDs (extended abstract)," in *Proceedings of the Eight International Conference on Computer Aided Verification, CAV '96* (R. Alur and T. Henzinger, eds.), New Brunswick, New Jersey, United States, Lecture Notes in Computer Science 1102, pp. 1–12, Springer-Verlag, Berlin, Germany, 31 July –3 August 1996. V-A

[14] C. Ferdinand, *Generating Program Analyzers*. Verfasser – Pirrot Verlag, Saarbrücken, Germany, 1999. V-A

[15] P. Cousot, "The calculational design of a generic abstract interpreter," in *Calculational System Design* (M. Broy and R. Steinbrüggen, eds.), vol. 173, pp. 421–505, NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, Amsterdam, The Netherlands, 1999. V-A

[16] B. Le Charlier and P. Van Hentenryck, "Experimental evaluation of a generic abstract interpretation algorithm for Prolog," in *Proceedings of the 1992 International Conference on Computer Languages*, Oakland, California, pp. 137–146, IEEE Computer Society Press, Los Alamitos, California, United States, 20–23 April 1992. V-A

[17] G. Puebla, M. Hermenegildo, and J. P. Gallagher, "An integration of partial evaluation in a generic abstract interpretation framework," in *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio, January 1999.*, pp. 75–84, University of Aarhus, Dept. of Computer Science, January 1999. V-A

[18] 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.), Linköping, Sweden, Lecture Notes in Computer Science 1060, pp. 192–207, Springer-Verlag, Berlin, Germany, 24–26 April 1996. V-A

[19] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, Cambridge, Massachusetts, United States, 1999. V-B

[20] K. Havelund, K. Larsen, and A. Skou, "Formal verification of an audio/video power controller using the real-time model checker UPPAAL," in *ARTS'99*, 1999. V-B

[21] K. Havelund, M. Lowry, and J. Penix, "Formal analysis of a space craft controller using SPIN," in *SPIN'98*, 1998. V-B

[22] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer (STTT)*, 2000. toappear. V-B

[23] P. Cousot and R. Cousot, "Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper," in *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92* (M. Bruynooghe and M. Wirsing, eds.), Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pp. 269–295, Springer-Verlag, Berlin, Germany, 1992. V-B, VII

[24] S. Graf and C. Loiseaux, "A tool for symbolic program verification and abstraction," in *Proceedings of the Fifth International Conference on Computer Aided Verification, CAV '93* (C. Courcoubetis, ed.), Elounda, Greece, Lecture Notes in Computer Sci-

ence 697, pp. 71–84, Springer-Verlag, Berlin, Germany, 28 June –1 July 1993. V-B, VII

[25] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem, "Property preserving abstractions for the verification of concurrent systems," *Formal Methods in System Design*, vol. 6, no. 1, 1995. V-B, VII

[26] M. Müller-Olm, D. Schmidt, and B. Steffen, "Model checking: a tutorial introduction," in *Proceedings of the Sixth International Symposium on Static Analysis, SAS '99* (A. Cortesi and G. Filé, eds.), Venice, Italy, 22–24 september 1999, Lecture Notes in Computer Science 1694, pp. 330–354, Springer-Verlag, Berlin, Germany, 1999. VI

[27] P. Cousot and R. Cousot, "Temporal abstract interpretation," in *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Boston, Massachusetts), pp. 12–25, ACM Press, New York, New York, United States, January 2000. VI

[28] 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), pp. 84–97, ACM Press, New York, New York, United States, 1978. VI

[29] T. Bultan, R. Gerber, and W. Pugh, "Model checking concurrent systems with unbounded variables, symbolic representations, approximations and experimental results," *ACM Transactions on Programming Languages and Systems*, vol. 21, pp. 747–789, July 1999. VI

[30] N. Halbwachs, "Delays analysis in synchronous programs," in *Proceedings of the Fifth International Conference on Computer Aided Verification, CAV '93* (C. Courcoubatis, ed.), Elounda, Greece, Lecture Notes in Computer Science 697, pp. 333–346, Springer-Verlag, Berlin, Germany, 28 June –1 July 1993. VI

[31] N. Halbwachs, "About synchronous programming and abstract interpretation," *Science of Computer Programming*, vol. 31, pp. 75–89, May 1998. VI

[32] N. Halbwachs, Y. Proy, and P. Roumanoff, "Verification of real-time systems using linear relation analysis," *Formal Methods in System Design*, vol. 11, pp. 157–185, August 1997. VI

[33] P.-H. Ho and H. Wong-Toi, "Automated analysis of an audio control protocol," in *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95* (P. Wolper, ed.), Liège, Belgium,Lecture Notes in Computer Science 939, pp. 381–394, Springer-Verlag, Berlin, Germany, 3–5 July 1995. VI

[34] F. Huch, "Verification of Erlang programs using abstract interpretation and model checking," in *Proceedings of the 1999 ACM SIGPLANInternational Conference on Logic Programming, ICFP '99*, (Paris, France), pp. 261–272, ACM Press, New York, New York, United States, 27–29 september 1999. VI

[35] R. Gupta, "A fresh look at optimizing array bound checking," in *ACM-SIGPLAN Conference on Programming Language Design and Implementation '90*, pp. 272–282, June 1990. 1

[36] 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.). 1, VIII-B

[37] H. Saïdi and N. Shankar, "Abstract and model check while you prove," in *Proceedings of the Eleventh International Conference on Computer Aided Verification, CAV '99* (N. Halbwachs and D. Peled, eds.), Trento, Italy, Lecture Notes in Computer Science 1633, pp. 443–454, Springer-Verlag, Berlin, Germany, 6–10 July 1999. VII

[38] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *Proceedings of the Ninth International Conference on Computer Aided Verification, CAV '97* (O. Grumberg, ed.), Haifa, Israel,Lecture Notes in Computer Science 1254, pp. 72–83, Springer-Verlag, Berlin, Germany, 22–25 July 1997. VII

[39] M. Das, "Static analysis of large programs: Some experiences (invited talk)," in *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '00*, (Boston, Massachusetts, United States), p. 1, ACM Press, New York, New York, United States, 22–23 January 2000. VII

[40] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas, "PVS: Combining specification, proof checking, and model checking," in *Proceedings of the Eight International Conference on Computer Aided Verification, CAV '96* (R. Alur and T. Henzinger, eds.), New Brunswick, New Jersey, United States, Lecture Notes in Computer Science 1102, pp. 411–414, Springer-Verlag, Berlin, Germany, 31 July –3 August 1996. VII

[41] A. Pnueli and E. Shahar, "A platform for combining deductive with algorithmic verification," in *Proceedings of the Eight International Conference on Computer Aided Verification, CAV '96* (R. Alur and T. Henzinger, eds.), New Brunswick, New Jersey, United States, Lecture Notes in Computer Science 1102, pp. 184–195, Springer-Verlag, Berlin, Germany, 31 July –3 August 1996. VII

[42] F. Giunchiglia and A. Villafiorita, "ABSFOL: A proof checker with abstraction," in *Proceedings of the Thirteenth International Conference on Automated Deduction, CADE '962* (M. McRobbie and J. Slaney, eds.), New Brunswick, New Jersey, United States, Lecture Notes in Computer Science 1104, pp. 136–140, Springer-Verlag, Berlin, Germany, 30 July – 3 August 1996. VII

[43] P. Cousot and R. Cousot, "Refining model checking by abstract interpretation," *Automated Software Engineering*, vol. 6, pp. 69–95, 1999. VIII-A, X

[44] A. Tarski, "A lattice theoretical fixpoint theorem and its applications," *Pacific Journal of Mathematics*, vol. 5, pp. 285–310, 1955. VIII-A, VIII-B, VIII-B

[45] P. Cousot and R. Cousot, "Constructive versions of Tarski's fixed point theorems," *Pacific Journal of Mathematics*, vol. 82, no. 1, pp. 43–57, 1979. VIII-A, VIII-B

[46] S. Berezin, E. Clarke, S. Jha, and W. Marrero, "Model checking algorithms for the μ-calculus," Technical report tr-cmu-cs-96-180, Carnegie Mellon University, september 1996. VIII-B

[47] M. Kaplan and J. Ullman, "A general scheme for the automatic inference of variable types," *Journal of the Association for Computing Machinary*, vol. 27, no. 1, pp. 128–145, 1980. VIII-B

[48] 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), pp. 269–282, ACM Press, New York, New York, United States, 1979. VIII-B, X

[49] P. Granger, "Improving the results of static analyses of programs by local decreasing iterations," in *Proceedings of the Twelfth Foundations of Software Technology and Theoretical Computer Science Conference* (R. Shyamasundar, ed.), New Delhi, India, 18–20 December 1992, Lecture Notes in Computer Science 652, pp. 68–79, Springer-Verlag, Berlin, Germany, 1992. VIII-C

[50] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992. IX

[51] P. Cousot and R. Cousot, "Induction principles for proving invariance properties of programs," in *Tools & Notions for Program Construction* (D. Néel, ed.), pp. 43–119, Cambridge University Press, Cambridge, United Kindom, 1982. X

[52] E. Clarke and E. Emerson, "Synthesis of synchronization skeletons for branching time temporal logic," in *IBM Workshop on Logics of Programs*, Lecture Notes in Computer Science 131, Springer-Verlag, Berlin, Germany, May 1981. X

[53] E. Clarke, E. Emerson, and A. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach," in *Conference Record of the Tenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 117–126, ACM Press, New York, New York, United States, January 1983. X

[54] R. Giacobazzi and F. Ranzato, "Completeness in abstract interpretation: A domain perspective," in *Proc. of the Sixth International Conference on Algebraic Methodology and Software Technology (AMAST'97)* (M. Johnson, ed.), vol. 1349 of *Lecture Notes in Computer Science*, pp. 231–245, Springer-Verlag, Berlin, Germany, 1997. X

[55] R. Giacobazzi, C. Palamidessi, and F. Ranzato, "Weak relative pseudo-complements of closure operators," *Algebra Universalis*, vol. 36, no. 3, pp. 405–412, 1996. X

[56] R. Giacobazzi, F. Ranzato, and F. Scozzari, "Complete abstract interpretations made constructive," in *Proceedings of the Twentythird International Symposium on Mathematical Foundations of Computer Science, MFCS'98* (L. Brim, J. Gruska, and J. Zlatuska, eds.), vol. 1450 of *Lecture Notes in Computer Science*, pp. 366–377, Springer-Verlag, Berlin, Germany, 1998. X

Published in the *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, CD Rom paper 248, L'Aquila, Italy, July 31 – August 6 2000. Scuola Superiore G. Reiss Romoli.