On Abstraction in Software Verification *

Patrick Cousot[1] and Radhia Cousot[2]

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

[2] CNRS & École polytechnique, Laboratoire d'informatique,
91128 Palaiseau cedex, France
`Radhia.Cousot@polytechnique.fr` `lix.polytechnique.fr/~rcousot`

Abstract. We show that the precision of static abstract software checking algorithms can be enhanced by taking explicitly into account the abstractions that are involved in the design of the program model/abstract semantics. This is illustrated on reachability analysis and abstract testing.

1 Introduction

Most formal methods for reasoning about programs (such as deductive methods, software model checking, dataflow analysis) do not reason directly on the trace-based operational program semantics but on an approximate model of this semantics. The abstraction involved in building the model of the program semantics is usually left implicit and not discussed. The importance of this abstraction appears when it is made explicit for example in order to discuss the soundness and (in)completeness of temporal-logic based verification methods [1, 2].

The purpose of this paper is to discuss the practical importance of this abstraction when designing static software checking algorithms. This is illustrated on reachability analysis and abstract testing.

2 Transition Systems

We follow [3, 4] in formalizing a hardware or software computer system by a transition system $\langle S,\ t,\ I,\ F,\ E\rangle$ 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 $F \subseteq S$.

An example is that of automatic program manipulation techniques based on the operational semantics of a programming language $\mathcal{L}$. Then there is a computable function mapping any program $p \in \mathcal{L}$ to a (symbolic computer representation of the) transition relation $t[\![p]\!]$ (as well as $I[\![p]\!]$, $F[\![p]\!]$ $E[\![p]\!]$).

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

D. Brinksma and K. G. Larsen (Eds.): CAV 2002, LNCS 2404, pp. 37–56, 2002.

A program execution trace $\sigma \in S^{\infty}$ is a maximal non-empty finite ($\sigma \in S^{+}$) or infinite ($\sigma \in S^{\omega}$) 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 $\sigma = \sigma_0 \ldots \sigma_i \ldots \sigma_n$ is finite (of length $|\sigma| = n + 1$) 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 (formally $\forall s \in S \setminus (E \cup F) : \exists s' \in S : \langle s,\ s' \rangle \in t$) in which case execution goes on normally, may be for ever (for infinite traces σ of length $|\sigma| = \omega$).

3 Reachability

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: $\mathrm{post}[t]\, X \stackrel{\mathrm{def}}{=} \{s' \in S \mid \exists s \in X : \langle s,\ s' \rangle \in t\}$ [5,6]. Let $\mathrm{lfp}^{\sqsubseteq}\ \varphi$ be the least fixpoint of a monotone map φ on a poset $\langle L, \sqsubseteq \rangle$ when it exists (e.g. $\langle L, \sqsubseteq \rangle$ is a cpo or a complete lattice). We have $\mathrm{post}[t^{\star}]\, I = \mathrm{lfp}^{\subseteq}\ \mathcal{F}[t]I$ where $\mathcal{F}[t]I(X) \stackrel{\mathrm{def}}{=} I \cup \mathrm{post}[t]\, X$ [3,5]. Given a specification $Q \subseteq S$, the *reachability problem* considered in [5] consists in proving that $\mathrm{post}[t^{\star}]\, I \subseteq Q$.

Inverse problems consist in considering the inverse t^{-1} of the relation t [3]. We let $\mathrm{pre}[t]\, X \stackrel{\mathrm{def}}{=} \mathrm{post}[t^{-1}]\, X$ 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: $\mathrm{pre}[t]\, X = \{s \in S \mid \exists s' \in X : \langle s,\ s' \rangle \in t\}$. From $(t^{\star})^{-1} = (t^{-1})^{\star}$ we have $\mathrm{pre}[t^{\star}]\, F = \mathrm{post}[(t^{\star})^{-1}]\, F = \mathrm{lfp}^{\subseteq}\ \mathcal{F}[t^{-1}]F = \mathrm{lfp}^{\subseteq}\ \mathcal{B}[t]F$ where $\mathcal{B}[t]F(X) \stackrel{\mathrm{def}}{=} F \cup \mathrm{pre}[t]\, X$ [3,4].

Dual problems [3] consist in considering the dual $\neg \circ \varphi \circ \neg$ of monotone functions φ on complete boolean lattices where $\neg X \stackrel{\mathrm{def}}{=} S \backslash X$ and $f \circ g(x) = f(g(x))$. The dual notions are $\widetilde{\mathrm{post}}[r]\, X \stackrel{\mathrm{def}}{=} \neg\, \mathrm{post}[r](\neg X)$ so that $\widetilde{\mathrm{post}}[r]\, X = \{s' \mid \forall s : \langle s,\ s' \rangle \in r \implies s \in X\}$ and $\widetilde{\mathrm{pre}}[r]\, X \stackrel{\mathrm{def}}{=} \neg\, \mathrm{pre}[r](\neg X) = \{s \mid \forall s' : \langle s,\ s' \rangle \in r \implies s' \in X\}$. Dual fixpoint characterizations follow from Park's dual fixpoint theorem for monotone functions on complete boolean lattices $\mathrm{gfp}\, \varphi = \neg\, \mathrm{lfp}\, \neg \circ \varphi \circ \neg$ [6].

4 Program Testing

Program testing was extended beyond reachability analysis to liveness properties such as termination [3,6]. 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 [7] under the name "abstract debugging" to:

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

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, when P is finite, the user can specify local invariant assertions Iv_p attached to program points $p \in \mathrm{Pv} \subseteq P$ and local intermittent assertions It_p attached to program points $p \in \mathrm{Pt}$ so that

$$\begin{aligned} & \mathrm{Iv} = \{\langle p,\ m\rangle \mid p \in \mathrm{Pv} \implies \mathrm{Iv}_p(m)\} \\ \text{and} \qquad & \mathrm{It} = \{\langle p,\ m\rangle \mid p \in \mathrm{Pt} \wedge \mathrm{It}_p(m)\}\ . \end{aligned}$$

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).

Consider for example, the factorial program below (the random assignment `?` is equivalent to the reading of an input value or the passing of an unknown but initialized parameter value and `<>` is $\neq$). A specification that this factorial program should always terminate normally states that any execution should always reach program point `6`. The termination requirement can be very simply specified as comments in the program text which specify the following local invariant and intermittent assertions:

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

$$\begin{aligned} &\mathrm{Iv}_p(\mathtt{n},\mathtt{f}) = \mathtt{n},\mathtt{f} \in [-\infty,+\infty], && p = 1,\ldots,6;\\ &\mathrm{It}_p(\mathtt{n},\mathtt{f}) = \mathit{false}, && p = 1,\ldots,5;\\ &\mathrm{It}_{\mathtt{6}}(\mathtt{n},\mathtt{f}) = \mathit{true}\ . \end{aligned}$$

5 Exact Formal Methods

Deductive methods were certainly the first considered to solve the reachability problem $\mathrm{post}[t^\star]\, I \subseteq Q$ that is $\mathrm{lfp}^{\subseteq}\, \mathcal{F}[t]I \subseteq Q$ exactly. By *exact*, we mean that one directly reason on the small-step operational semantics $\langle S[\![p]\!],\ t[\![p]\!],\ I[\![p]\!],\ F[\![p]\!],\ E[\![p]\!]\rangle$ of the considered program $p \in \mathcal{L}$.

5.1 Exact Deductive Methods

By Park fixpoint induction, we have $\mathrm{lfp}^{\subseteq}\, \mathcal{F}[t]I \subseteq Q$ if and only if $\exists J : \mathcal{F}[t]I(J) \subseteq J \wedge J \subseteq Q$ that is $\exists J : I \subseteq J \wedge \mathrm{post}[t]\, J \subseteq J \wedge J \subseteq Q$. This is Floyd's inductive proof method, subgoal induction for the inverse problem and contrapositive methods for dual problems [8].

Human interaction is necessary both to help discover the inductive argument J and to assist the prover to check the verification conditions $I \subseteq J$, $\mathrm{post}[t]\, J \subseteq J$ and $J \subseteq Q$ because the implication $\subseteq$ is not decidable. In general the transition relation $t[\![p]\!]$ is specified by a formal semantics of a programming language so that the formula $\mathrm{post}[t[\![p]\!]]$ can be computer-generated by structural induction on the syntax of program $p \in \mathcal{L}$, although human interaction is again needed since these formulae must be simplified. Moreover, the invariant J is program specific so the proof is not reusable and may have to be completely redone when the program is modified.

5.2 Exact Model Checking

When the set S of states is finite, model checking [9–11] consists in computing exactly $\mathrm{lfp}^{\subseteq}\ \mathcal{F}[t]I$ and checking that the fixpoint iterates are all included in Q. Efficient data structures and algorithms have been developped for boolean encodings such as BDDs [12]. To prove $\mathrm{lfp}^{\subseteq}\ \mathcal{F}[t]I \nsubseteq Q$, one can prove that there exist an iterate of $\mathcal{F}[t]I$ which is not included in Q using SAT [13]. Unfortunately this can only be used for debugging and does not presently scale up beyond a few thousands boolean variables.

In practice the transition system $\langle S,\, t,\, I,\, F,\, E\rangle$ is in general not that of the semantics $\langle S[\![p]\!], t[\![p]\!], I[\![p]\!], F[\![p]\!], E[\![p]\!]\rangle$ of the considered computer system $p \in \mathcal{L}$ but an abstraction of this semantics. This abstraction is often done by hand and its correctness is not discussed. The abstract interpretation framework [4, 5] is a formal basis for making the abstraction explicit and for proving its soundness and (in)completeness.

6 Abstract Interpretation

Abstract Interpretation [3] is a theory of abstraction of structures. Let us recall a few basic elements of this theory [4].

An abstraction is defined by a 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 $x \sqsubseteq x'$ (for example x is a property which implies a weaker one 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 upper approximation since $x \sqsubseteq \gamma \circ \alpha(x)$. It is the more precise since for any other abstract approximation y, $x \sqsubseteq \gamma(y) \implies \gamma \circ \alpha(x) \sqsubseteq \gamma(y)$.

In order to abstract fixpoints, let us recall the following classical results in abstract interpretation [3–5]:

Theorem 1 (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* $F \in L \xmapsto{mon} L$*, 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, the least fixpoints exist. So let $Q^\sharp \stackrel{\mathrm{def}}{=} \mathrm{lfp}^{\sqsubseteq^\sharp}\ \alpha \circ F \circ \gamma$. We have $\alpha \circ F \circ \gamma(Q^\sharp) = Q^\sharp$ whence $F \circ \gamma(Q^\sharp) \sqsubseteq \gamma(Q^\sharp)$ by definition of Galois connections. It follows that $\gamma(Q^\sharp)$ is a postfixpoint of F whence $\mathrm{lfp}^{\sqsubseteq}\ F \sqsubseteq \gamma(Q^\sharp)$ by Tarski's fixpoint theorem or equivalently $\alpha(\mathrm{lfp}^{\sqsubseteq}\ F) \sqsubseteq^\sharp Q^\sharp = \mathrm{lfp}^{\sqsubseteq^\sharp}\ \alpha \circ F \circ \gamma$. □

Theorem 2 (Fixpoint approximation). *If* $\langle L^\sharp,\, \sqsubseteq^\sharp,\, \bot^\sharp,\, \top^\sharp,\, \sqcup^\sharp,\, \sqcap^\sharp\rangle$ *is a complete lattice,* $F^\sharp, \bar{F}^\sharp \in L^\sharp \xmapsto{mon} L^\sharp$*, 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. □

7 Abstract Interpretation Based Formal Methods

In order to reduce the need for human-interaction as in deductive methods some form of abstraction is needed. This consists in replacing program properties by abstract ones. For example a set of traces $X \subseteq S^\infty$ can be abstracted by sets of reachable states $\alpha(X)$ where $\alpha \in \wp(S^\infty) \mapsto \wp(S)$ is $\alpha(X) \stackrel{\mathrm{def}}{=} \{\sigma_i \mid \sigma \in X \wedge 0 \leqslant i < |\sigma|\}$ and $\gamma \in \wp(S) \mapsto \wp(S^\infty)$ is $\gamma(Y) \stackrel{\mathrm{def}}{=} \{\sigma \in S^\infty \mid \forall i < |\sigma| : \sigma_i \in Y\}$ so that $\langle\wp(S^\infty),\, \subseteq\rangle \xrightleftharpoons[\alpha]{\gamma} \langle\wp(S),\, \subseteq\rangle$. For reachability, a further abstraction $\langle\wp(S),\, \subseteq\rangle \xrightleftharpoons[\alpha]{\gamma} \langle L^\sharp,\, \sqsubseteq^\sharp\rangle$ leads, by Th. 1 & 2 to $\mathrm{post}[t^\star]\, I \subseteq \gamma(\mathrm{lfp}^{\sqsubseteq^\sharp} \mathcal{F}^\sharp)$ where $\alpha \circ \mathcal{F}[t]I \circ \gamma \sqsubseteq^\sharp \mathcal{F}^\sharp$ pointwise, so that $\mathrm{lfp}^{\sqsubseteq^\sharp} \mathcal{F}^\sharp \sqsubseteq^\sharp Q^\sharp$ implies $\mathrm{post}[t^\star]\, I \subseteq \gamma(Q^\sharp)$. Depending on the scope of the abstraction, there are essentially two approaches:

- In static program analysis, an abstraction $\alpha[\![p]\!]$ has to be conceived by the designer for all programs $p \in \mathcal{L}$ of a programming language $\mathcal{L}$;
- In abstract software model checking, a specific abstraction α is designed by the end-user for each particular program.

We now examine the consequences of these two possible choices.

7.1 Static Program Analysis

In static program analysis, the abstract interpreter is given any program $p \in \mathcal{L}$ of a programming language $\mathcal{L}$, establishes equations $X = \mathcal{F}^\sharp[\![p]\!](X)$ or constraints $\mathcal{F}^\sharp[\![p]\!](X) \sqsubseteq^\sharp X$ where $\alpha[\![p]\!] \circ \mathcal{F}[t[\![p]\!]]I[\![p]\!] \circ \gamma[\![p]\!] \sqsubseteq^\sharp \mathcal{F}^\sharp[\![p]\!]$ and computes or effectively upper approximates $\mathrm{lfp}^{\sqsubseteq^\sharp} \mathcal{F}^\sharp[\![p]\!]$.

There is no need for the user to manually design the abstract interpreter, which is done by specialists. Hence there is no easy fine tuning of the abstract interpreter for a particular specification and a particular program. A consequence of this generality is that there will always be some program on which the analyzer will produce false alarms.

To minimize false alarms, 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 [14] uses reachability analysis with the attribute-independent interval domain [5] 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. It is easy to prove that for any $n > 0$, an interval analyzer [5] will discover ranges of possible values for numerical variables as follows (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 uninitialized value Ω is denoted `_O_`.):

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

The argument is then as follows:

1. for any given n it is possible to find an abstract domain (here $\{\Omega,\ [0,n],\ [0,n-1],\ [1,n],\ [n,n]\}$) 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` [15].
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 non termination ($+\infty$ (respectively $-\infty$) typed `+oo` (resp. `-oo`) denotes the greatest (resp. smallest) machine representable integer):

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

   Program point 4 is not reachable which is denoted by the bottom value $\perp$ (typed `_|_`).

7.2 Abstract Model Checking

Most abstractions considered in abstract model checking [16, 17] are state to state 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 [1, sec. 14, p. 23]. Then we have $\langle \wp(S), \subseteq\rangle \xleftarrow[\alpha]{\gamma} \langle \wp(S^\sharp), \subseteq\rangle$ where $\gamma(Y) \stackrel{\text{def}}{=} \{x \in S \mid \alpha(x) \in Y\}$. This is of the form $\langle \wp(S), \subseteq\rangle \xleftarrow[\text{post}[\alpha]]{\widetilde{\text{pre}}[\alpha]} \langle \wp(S^\sharp), \subseteq\rangle$ which is a slight generalization when the state-to-state abstraction is relational ($\alpha \subseteq S \times S^\sharp$) and not simply functional ($\alpha \in S \mapsto S^\sharp$).

The need for restricting to state-to-state abstractions 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. Indeed $\alpha \circ \mathrm{post}[t] \circ \gamma$ is $\mathrm{post}[t^\sharp]$ by defining $t^\sharp \stackrel{\mathrm{def}}{=} \{\langle s^\sharp,\ s'^\sharp\rangle \mid \exists s, s' \in S : \langle s,\ s^\sharp\rangle \in \alpha \wedge \langle s,\ s'\rangle \in t \wedge \langle s',\ s'^\sharp\rangle \in \alpha\}$, $I^\sharp \stackrel{\mathrm{def}}{=} \{s^\sharp \mid \exists s \in I : \langle s,\ s^\sharp\rangle \in \alpha\}$, etc.

Contrary to a common believe not all abstractions are state-to state. So some abstract semantics (using e.g. the interval abstraction [5] or the polyhedral abstraction [18]) 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. [19]. But then they use abstract interpretation based techniques such as fixpoint approximation, widening/narrowing, etc. to check safety (mainly reachability) properties as considered in Sec. 7.1.

In (abstract) model-checking, all efforts are concentrated on the design of the (abstract) model $\langle S^\sharp,\ t^\sharp,\ I^\sharp,\ F^\sharp,\ E^\sharp\rangle$. By [15], the abstract model can always be chosen to be boolean, finite and even small enough so that a model-checker will always succeed. Not surprisingly, [15] shows that for reachability problems, the discovery of the adequate abstract model is logically equivalent to the discovery of an inductive invariant J while the soundness proof is logically equivalent to the inductive proof $\mathrm{post}[t]\,J \subseteq J$ (as considered in Sec. 5.1). So the human effort which was placed in the assistance of a prover for deductive methods is now placed in the design of an (abstract) model, which is a significant saving only when the abstract model is not proved correct. However these abstractions developed for a specific program and an explicit specification of that program are not reusable hence extremely expansive to design.

7.3 Automatic Abstraction and Abstraction Refinement

Predicate abstraction [20] is the per example automatization of the design of specific program analyzers by computing $\alpha[\![p]\!] \circ \mathcal{F}[t[\![p]\!]]I[\![p]\!] \circ \gamma[\![p]\!]$ for a given program p with a theorem prover/proof assitant/algebraic simplifier. It is based on the use of abstract domains using finite sets of predicates in disjunctive normal form. The abstract domain is refined on a spurious counter example basis [21]. This is an instance of domain refinement in abstract interpretation [22] but for the fact that infinite-disjunctions are handled heuristically to cope with uncomputability.

Beyond the limitations on the automatic use of theorem provers, the huge cost of the refinement process, the precision of the method as analyzed by [21] may be insufficient. This is because widening by dropping conjuncts in disjuncts [21] is not equivalent to the extrapolations to limits involved in abstract domains. An example is:

```
x := 100; y := -200;
while x <> 0 do x := (x - 1); y := (y + 2) od;;
always { y = 0 }
```

which is easily and automatically handled by the linear equalities abstract domain [23] (which satisfies the ascending chain condition) since the loop invariant $2x - y = 0$ and $x = 0$ implies $y = 0$.

7.4 Abstract Software Model Checking

At first sight, abstract testing is model-checking [9–11] of the temporal formula[3]:

$$\Box\Big(\bigwedge_{p\in\mathrm{Pv}} at_p \implies \mathrm{Iv}_p\Big) \wedge \Diamond\Big(\bigvee_{p\in\mathrm{Pt}} at_p \wedge \mathrm{It}_p\Big) \tag{1}$$

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

Note that with state to state abstraction, the correctness of the formula in the abstract does not imply its correctness in the concrete (see [1, 2]). A simple counter-example would be the termination of:

```
n := ?; f := 1; b := true;
while ((n <> 0) | b) do f := (f * n); n := (n - 1) od;;
```

where memory states $\langle n, f, b\rangle$ are abstracted by $\langle n, f\rangle$. So the verification of (1) cannot be done with upper approximations only and would also require a lower approximation or the use of a variant function [1, 2].

7.5 Abstract Testing

Since lower approximations are hard to design and the handling of both lower and upper approximations is computationally complex, abstract testing uses upper approximations only. This consists in automatically computing local upper approximations A_p, $p \in P$, such that for $A \stackrel{\text{def}}{=} \{\langle p, m\rangle \mid A_p(m)\}$ we have:

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

The information provided by A is $(\neg A \wedge \mathrm{post}[t^\star]\, I) \implies (\neg\mathrm{Iv} \vee \widetilde{\mathrm{pre}}[t^\star]\, \neg\mathrm{It})$, that is reachable states not satisfying A either do not satisfy the invariant assertion Iv or must inevitably lead to a violation of the intermittent assertion It. So A should be checked at checking-time or run-time to forestall errors. This consists in considering the transition system $\langle S, t', I', F, E\rangle$ where $t' \stackrel{\text{def}}{=} t \cap (S \times A)$ and $I' \stackrel{\text{def}}{=} I \cap A$. If $S = P \times M$ then defining $t_{pp'} \stackrel{\text{def}}{=} \{\langle\langle p, m\rangle, \langle p', m'\rangle\rangle \in t\}$ and $\mathrm{succ}(p) \stackrel{\text{def}}{=} \{p' \in P \mid t_{pp'} \neq \varnothing\}$, we have $t = \bigcup_{p'\in\mathrm{succ}(p)} t_{pp'}$. An economical way to check A is to use local checks (denoted `:!:` in the examples below). This consists in considering $t' = \bigcup_{p'\in\mathrm{succ}(p)} t'_{pp'}$ where $t'_{pp'} \stackrel{\text{def}}{=} t_{pp'}$ if $\mathrm{post}[t_{pp'}]\, A_p \implies A_{p'}$ and $t'_{pp'} \stackrel{\text{def}}{=} t_{pp'} \cap (S \times A_{p'})$ otherwise. If A is taken in computer-language representable abstract domains, the transformed transition system $\langle S, t', I', F, E\rangle$ corresponds to a transformed program, which is a simple form of program monitoring [24].

The automatic analysis of the above factorial program leads to the following result [25, 26]:

[3] The temporal operator $\Box Q$ denotes the set of sequences of states such that all states satisfy Q, $\Diamond Q$ denotes the set of sequences containing at least one state satisfying Q and the predicate $at\,p$ holds in all states which control point is p.

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

The analysis automatically discovers the condition `n` $\geqslant 0$ 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` $\geqslant 1$.

8 Precise Fixpoint Checking in the Presence of Approximations

All approximate formal methods considered in Sec. 7 involve fixpoint approximations. These approximations, such as widenings [5], can be simply ignored in model-checking of finite-state transition systems. However, *in the presence of approximations*, fixpoint approximation check can be made more precise than the fixpoint computations involved in traditional abstract model-checking [16, 17].

8.1 Fixpoint Approximation Check

A first illustration is for the fixpoint approximation check $\mathrm{lfp}^{\sqsubseteq}\,\mathcal{F} \sqsubseteq Q$ where $\langle L, \sqsubseteq, \bot, \top, \sqsubseteq, \sqsupseteq\rangle$ is a complete lattice, $\mathcal{F} \in L \xmapsto{\text{mon}} L$ is monotonic and $\mathrm{lfp}^{\sqsubseteq}\,\mathcal{F}$ is the $\sqsubseteq$-least fixpoint of $\mathcal{F}$. An example is reachability analysis of Sec. 3.

In (abstract) model-checking, one computes iteratively $\mathrm{lfp}^{\sqsubseteq}\,\mathcal{F}$ and then checks that $\mathrm{lfp}^{\sqsubseteq}\,\mathcal{F} \sqsubseteq Q$ (or uses a strictly equivalent check, see [27, p. 73] and Sec. 10 below).

In abstract testing, one computes iteratively an upper-approximation J of $\mathrm{lfp}^{\sqsubseteq}\,\boldsymbol{\lambda} X \cdot Q \sqcap \mathcal{F}(X)$ with acceleration of the convergence of the iterates by widening/narrowing [5, 6]. The convergence criterion is:

$$(Q \sqcap \mathcal{F}(J)) \sqsubseteq J \;. \tag{2}$$

Then the invariance check has the form:

$$\mathcal{F}(J) \sqsubseteq Q \;. \tag{3}$$

This is sound, by the following theorem:

Theorem 3. *If* $\langle L, \sqsubseteq, \bot, \top, \sqsubseteq, \sqsupseteq\rangle$ *is a complete lattice,* $\mathcal{F} \in L \xrightarrow{mon} L$ *is monotonic and* $Q, J \in L$*, then:*

$$(Q \sqcap \mathcal{F}(J)) \sqsubseteq J \wedge \mathcal{F}(J) \sqsubseteq Q \implies \mathrm{lfp}^{\sqsubseteq}\, \mathcal{F} \sqsubseteq Q$$

Proof. We have $\mathcal{F}(J) = \mathcal{F}(J) \sqcap \mathcal{F}(J) \sqsubseteq Q \sqcap \mathcal{F}(J)$ [by (3)] $\sqsubseteq J$ [by (2)] proving $\mathcal{F}(J) \sqsubseteq J$ by transitivity whence $\mathrm{lfp}^{\sqsubseteq}\, \mathcal{F} \sqsubseteq J$ by Tarski's fixpoint theorem. By definition of fixpoints and monotony, it follows that $\mathrm{lfp}^{\sqsubseteq}\, \mathcal{F} = \mathcal{F}(\mathrm{lfp}^{\sqsubseteq}\, \mathcal{F}) \sqsubseteq \mathcal{F}(J)$ $\sqsubseteq Q$ [by (3)]. By transitivity, we conclude $\mathrm{lfp}^{\sqsubseteq}\, \mathcal{F} \sqsubseteq Q$ 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 [5] of the simple loop accessing sequentially an array $A[1], \ldots, A[100]$. The result of the analysis [26] is too approximate to statically check that the index `i` is within the array bounds 1 and 100 :

```
Reachability from initial states;
i := 0;
while (i <> 100) do
    i := (i + 1);
    skip % array access %
od;;
```

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

However by explicit conjunction with the array access invariant $0 < \mathtt{i} \leqslant 100$ (the evaluation of the runtime check `always B` has the effect of blocking the program execution when the assertion `B` does not hold), the static analysis now proves that the array out of bound error is impossible:

```
Reachability from initial states;
i:=0;
while i <> 100 do
  i := i + 1;
  always ((0 < i) & (i <= 100))
od;;
```

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

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

8.2 Fixpoint Meet Approximation

A second illustration of the possible refinement of fixpoint computation algorithms in the presence of abstraction is the upper-approximation of the descendants of the initial states which are ancestors of the final states. An (abstract) model-checking algorithm (such as [28]) computes a conjunction of forward and backward fixpoints. The forward and backward analyses of the factorial program, respectively yield:

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

```
Ancestors of final states;
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) 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
6: { n:[-oo,+oo]?; f:[-oo,+oo]? }
```

The intersection is therefore:

```
0: { n:_O_; f:_O_ }
  n := ?;
1: { n:[-oo,+oo]; f:_O_ }
  f := 1;
2: { n:[0,+oo]; f:[-oo,+oo]? }
  while (n <> 0) 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-1]; f:[-oo,+oo] }
  od
6: { n:[0,0]; f:[-oo,+oo] }
```

Abstract testing iterates an alternation between forward and backward fixpoints [3, 29]. For the factorial program, the analysis is more precise (since it can now derive that `f` is positive):

```
Reachability/ancestry analysis for initial/final states;
0: { n:_O_; f:_O_ }
  n := ?;
1:!: { n:[0,+oo]; f:_O_ }
  f := 1;
2: { n:[0,+oo]; f:[1,+oo] }
  while (n <> 0) do
```

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

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

$$\dot{X}^0 = \mathrm{lfp}^{\sqsubseteq^\sharp} \mathcal{F}^\sharp \text{ or } \mathrm{lfp}^{\sqsubseteq^\sharp} \mathcal{B}^\sharp \tag{4}$$

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

$$\dot{X}^{2n+2} = \mathrm{lfp}^{\sqsubseteq^\sharp} \boldsymbol{\lambda} X \cdot (\dot{X}^{2n+1} \sqcap^\sharp \mathcal{F}^\sharp(X)), \quad n \in \mathbb{N} \tag{6}$$

For soundness, we assume:

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

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

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

$$= \mathrm{lfp}^{\sqsubseteq} \boldsymbol{\lambda} X \cdot (\mathrm{lfp}^{\sqsubseteq} \mathcal{F} \sqcap \mathrm{lfp}^{\sqsubseteq} \mathcal{B} \sqcap \mathcal{F}(X)) \tag{10}$$

so that there is no improvement when applying the alternating fixpoint computation to $\mathcal{F}$ and $\mathcal{B}$ (such as the exact forward $\mathrm{post}[t^\star]\, I$ and backward $\mathrm{pre}[t^\star]\, F$ collecting semantics). However, when considering approximations $\mathcal{F}^\sharp$ of $\mathcal{F}$ and $\mathcal{B}^\sharp$ of $\mathcal{B}$, 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 [30] to infer types in flowchart programs.

The correctness of the alternating fixpoint computation follows from the following [3]:

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,* $\mathcal{F} \in L \xrightarrow{mon} L$ *and* $\mathcal{B} \in L \xrightarrow{mon} L$ *satisfy the hypotheses (9) and (10),* $\mathcal{F}^\sharp \in L^\sharp \xrightarrow{mon} L^\sharp$*,* $\mathcal{B}^\sharp \in L^\sharp \xrightarrow{mon} L^\sharp$*,* $\alpha \circ \mathcal{F} \circ \gamma \sqsubseteq^\sharp \mathcal{F}^\sharp$*,* $\alpha \circ \mathcal{B} \circ \gamma \sqsubseteq^\sharp \mathcal{B}^\sharp$ *and the sequence* $\langle \dot{X}^n,\ n \in \mathbb{N} \rangle$ *is defined by (4), (5) and (6) then* $\forall k \in \mathbb{N} : \alpha(\mathrm{lfp}^{\subseteq} \mathcal{F} \cap \mathrm{lfp}^{\subseteq} \mathcal{B}) \sqsubseteq^\sharp \dot{X}^{k+1} \sqsubseteq^\sharp \dot{X}^k$.

Proof. Observe that by the fixpoint property, $\dot{X}^{2n+1} = \dot{X}^{2n} \sqcap^\sharp \mathcal{B}^\sharp(\dot{X}^{2n+1})$ and $\dot{X}^{2n+2} = \dot{X}^{2n+1} \sqcap^\sharp \mathcal{F}^\sharp(\dot{X}^{2n+2})$, 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 $\alpha(\operatorname{lfp}^{\sqsubseteq} \mathcal{F} \sqcap \operatorname{lfp}^{\sqsubseteq} \mathcal{B}) \sqsubseteq^\sharp \alpha(\operatorname{lfp}^{\sqsubseteq} \mathcal{F})$ since α is monotone and $\alpha(\operatorname{lfp}^{\sqsubseteq} \mathcal{F}) \sqsubseteq^\sharp \operatorname{lfp}^{\sqsubseteq^\sharp} \mathcal{F}^\sharp$ by Th. 2, thus proving the proposition for $k = 0$.

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

Assume now by induction hypothesis that $\alpha(\operatorname{lfp}^{\sqsubseteq} \mathcal{F} \sqcap \operatorname{lfp}^{\sqsubseteq} \mathcal{B}) \sqsubseteq^\sharp \dot{X}^{2n}$, or equivalently, by definition of Galois connections, that $\operatorname{lfp}^{\sqsubseteq} \mathcal{F} \sqcap \operatorname{lfp}^{\sqsubseteq} \mathcal{B} \sqsubseteq \gamma(\dot{X}^{2n})$. Since $\mathcal{F} \sqsubseteq \gamma \circ \mathcal{F}^\sharp \circ \alpha$, it follows that $\boldsymbol{\lambda} X \cdot \operatorname{lfp}^{\sqsubseteq} \mathcal{F} \sqcap \operatorname{lfp}^{\sqsubseteq} \mathcal{B} \sqcap \mathcal{F}(X) \sqsubseteq \boldsymbol{\lambda} X \cdot \gamma(\dot{X}^{2n}) \sqcap \gamma \circ \mathcal{F}^\sharp \circ \alpha(X) = \boldsymbol{\lambda} X \cdot \gamma(\dot{X}^{2n} \sqcap \mathcal{F}^\sharp \circ \alpha(X))$ since, in a Galois connection, γ is a complete meet morphism. Now by hypothesis (9), we have $\operatorname{lfp}^{\sqsubseteq} \mathcal{F} \sqcap \operatorname{lfp}^{\sqsubseteq} \mathcal{B} = \operatorname{lfp} \boldsymbol{\lambda} X \cdot (\operatorname{lfp}^{\sqsubseteq} \mathcal{F} \sqcap \operatorname{lfp}^{\sqsubseteq} \mathcal{B} \sqcap \mathcal{F}(X)) \sqsubseteq^\sharp \operatorname{lfp} \boldsymbol{\lambda} X \cdot \gamma(\dot{X}^{2n} \sqcap \mathcal{F}^\sharp \circ \alpha(X))$ by Th. 2. Let G be $\boldsymbol{\lambda} X \cdot \dot{X}^{2n} \sqcap \mathcal{F}^\sharp(X)$. 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. 1, we have $\alpha(\operatorname{lfp} \gamma \circ G \circ \alpha) \sqsubseteq^\sharp \operatorname{lfp} \alpha \circ \gamma \circ G \circ \alpha \circ \gamma \sqsubseteq^\sharp \operatorname{lfp} G$ by Th. 2. Hence, $\operatorname{lfp} \boldsymbol{\lambda} X \cdot \gamma(\dot{X}^{2n} \sqcap \mathcal{F}^\sharp \circ \alpha(X)) \sqsubseteq \gamma(\operatorname{lfp} \boldsymbol{\lambda} X \cdot \dot{X}^{2n} \sqcap \mathcal{F}^\sharp(X))$ so that by transitivity we conclude that $\alpha(\operatorname{lfp}^{\sqsubseteq} \mathcal{F} \sqcap \operatorname{lfp}^{\sqsubseteq} \mathcal{B}) \sqsubseteq^\sharp \dot{X}^{2n+1}$.

The proof that $\alpha(\operatorname{lfp}^{\sqsubseteq} \mathcal{F} \sqcap \operatorname{lfp}^{\sqsubseteq} \mathcal{B}) \sqsubseteq^\sharp \dot{X}^{2n+2}$ is similar, using hypothesis (9) and by exchanging the rôles of $\mathcal{F}$ and $\mathcal{B}$. □

It is interesting to note that the computed sequence (4), (5) and (6) is optimal (see [3]). A similar result holds when replacing one least fixpoint by a greatest fixpoint [31].

If the abstract lattice does not satisfy the descending chain condition then [3] also suggests to use a narrowing operator $\triangle$ [5] 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 $\boldsymbol{\lambda} X \cdot \dot{X}^{2n} \sqcap \mathcal{F}^\sharp(X)$ and $\boldsymbol{\lambda} X \cdot \dot{X}^{2n+1} \sqcap \mathcal{B}^\sharp(X)$.

8.3 Local Iterations

A third illustration that the precision of static abstract software checking algorithms can be enhanced by taking explicitly into account the abstractions is the local iterations [32] to handle tests, backward assignments, etc. Below is an example of program static analysis, without local iterations:

```
Reachability from initial states;
0: { x:_O_; y:_O_; z:_O_ }
  x := 0;
1: { x:[0,0]; y:_O_; z:_O_ }
  y := ?;
```

```
2: { x:[0,0]; y:[-oo,+oo]; z:_O_ }
  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:

```
Forward reductive analysis from initial states;
0: { x:_O_; y:_O_; z:_O_ }
  x := 0;
1: { x:[0,0]; y:_O_; z:_O_ }
  y := ?;
2: { x:[0,0]; y:[-oo,+oo]; z:_O_ }
  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.

9 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 [12]. This is not directly possible in the abstract since the necessary over-approximation leads to the consideration of inexisting program executions which should not be proposed as counter-examples. Nevertheless, in abstract model checking, counterexamples can be found in the concrete [33, 34], provided concrete program transformers can be effectively computed (e.g. when the concrete transition system is finite). Because of the uncomputability of the programming language semantics, this is not always possible with abstract testing (e.g. for non-termination).

However, abstract testing can provide necessary conditions for the specification to be (un-)satisfied. These automatically calculated conditions can serve to abstract program slicing 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 leads to the necessary pre-condition $\mathtt{n} \geqslant 0$:

```
Ancestors of final states;
n := ?;
f := 1;
while (n <> 0) do
   f := (f * n);
   n := (n - 1)
od;;
```

```
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) 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
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 $\mathtt{n} < 0$ shows that the program execution never terminates properly so that the only remaining possible case is an incorrect termination with a run-time error ($\perp$, typed `_|_`, is the false invariant hence denotes unreachability in forward analysis and impossibility to reach the goal in backward analysis):

```
Reachability from initial states;
initial n < 0;
f := 1;
while (n <> 0) do
   f := (f * n);
   n := (n - 1)
od;;
```

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

```
                                6: { n:_|_; f:_|_ }
```

Otherwise stated, infinitely many counter-examples are simultaneously provided by this counter-analysis. Except in the case of bounded cyclicity, concrete nontermination counterexamples would be hard to exhibit for infinite state transition systems.

10 Contrapositive Reasoning

For the last element of comparison between concrete and abstract software verification, 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 [27, p. 73], the Galois connection $\langle \wp(S),\ \subseteq\rangle \underset{\text{post}[r]}{\overset{\widetilde{\text{pre}}[r]}{\longleftrightarrow}} \langle \wp(S),\ \subseteq\rangle$ (where $r \subseteq S \times S$ and $\widetilde{\text{pre}}[r]\, X \stackrel{\text{def}}{=} \{s \mid \forall s' : \langle s,\ s'\rangle \in r \implies s' \in X\}$) implies that the invariance specification check $\text{post}[t^\star]\, E \subseteq Q$ is equivalent to $\widetilde{\text{pre}}[t^\star]\, \neg Q \subseteq \neg E$ (or $\text{pre}[t^\star]\, \neg Q \subseteq \neg E$ for total deterministic transition systems [6]). Otherwise stated a forward positive proof is equivalent to a backward contrapositive proof, as observed in [8]. So the difference between the abstract testing algorithm of [4–6] and the model-checking algorithm of [9–11] is that abstract testing checks $\text{post}[t^\star]\, I \subseteq Q$ while model-checking verifies $\widetilde{\text{pre}}[t^\star]\, \neg Q \subseteq \neg I$, which is equivalent for finite transition systems as considered in [9–11].

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 [5] 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:

```
Ancestors of final states;
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,+oo-1] }
  while (i <> 100) do
```

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

```
Ancestors of final states;
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 and similarly for the upper bound:

```
0: { i:_|_ }
  i := 0;
1: { i:[-oo,-1] }
  while (i <> 100) 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)}
        6: { i:[-oo,0] }
          final (i <= 0)
        7: { i:[-oo,-1] }
      fi
    8: { i:[-oo,-1] }
  od
```

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

```
9: { i:_|_ }                        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 [22]. To start with, the abstract domain might be enriched with complements, but this might not be sufficient and indeed the abstract domain might have to be enriched for each primitive operation, thus leading to an abstract algebra which might be quite difficult to implement if not totally inefficient. Moreover limit abstract values in the abstract domains require infinite iterations so that the exact abstract domain refinement may not be computable (see Sec. 7.3).

11 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. Abstraction was first considered in contexts in which false alarms are quite acceptable (e.g. static program analysis in compilation [5], overestimation of worst-case execution time [35], etc) to applications in which false alarms are unsatisfactory (e.g. software verification). For example in compile-time boundedness checking, a selectivity of 90% will lead to significant performance improvements in execution times whereas such a selectivity rate is not acceptable to prove that no unexpected interrupts can be raised in large embedded critical real-time software. By concentrating on models of programs rather than on their semantics, these formal methods have had more successes for finding bugs than for actual correctness proofs of full programs. Beyond the design and maintenance cost of models of complex programs and their unreliability, the basic idea of complete program verification underlying the deductive and model checking methods has been abandoned in favor of debugging. Because of theoretical and practical limitations, these approaches will be hard to scale up for complex programs as considered in the DAEDALUS project (over 250 000 lines of C) e.g. for boundedness checking or liveness analysis.

Abstract interpretation based 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. There are essentially two approaches:

- *General-purpose static analyzers* automatically provide a program model by an approximation of its semantics chosen to offer a good average cost/precision compromize. Such analyzers are reusable and so their development cost can be shared among many users;
- *Specializable static analyzers* provide the user with the capability to tune the abstractions to achieve high-precision by chosing among a predefined set of wide-spectrum parameterized approximations. The refinement is on the local choice of abstract domains which automatically induces a more precise abstract semantics. This should ensure the soundness of the abstraction,

and at least for specific classes of programs lead to very precise and efficient analyzes going much beyond examples modelled by hand.

References

1. Cousot, P., Cousot, R.: Temporal abstract interpretation. In: 27^{th} POPL, Boston, USA, ACM Press (2000) 12–25
2. Schmidt, D.: From trace sets to modal-transition systems by stepwise abstract interpretation. Submitted for publication (2001)
3. Cousot, P.: 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 (1978)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6^{th} POPL, San Antonio, USA, ACM Press (1979) 269–282
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4^{th} POPL, Los Angeles, USA, ACM Press (1977) 238–252
6. Cousot, P.: Semantic foundations of program analysis. In Muchnick, S., Jones, N., eds.: Program Flow Analysis: Theory and Applications. Prentice-Hall (1981) 303–342
7. Bourdoncle, F.: Abstract debugging of higher-order imperative languages. In: Proc. ACM SIGPLAN '93 Conf. PLDI. ACM SIGPLAN Not. 28(6), Albuquerque, USA, ACM Press (1993) 46–55
8. Cousot, P., Cousot, R.: Induction principles for proving invariance properties of programs. In Néel, D., ed.: Tools & Notions for Program Construction. Cambridge U. Press (1982) 43–119
9. Clarke, E., Emerson, E.: Synthesis of synchronization skeletons for branching time temporal logic. In: IBM Workshop on Logics of Programs. Yorktown Heights, USA, LNCS 131, Springer-Verlag (1981)
10. Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. TOPLAS **8** (1986) 244–263
11. Queille, J.P., Sifakis, J.: Verification of concurrent systems in Cesar. In: Proc. Int. Symp. on Programming. LNCS 137. Springer-Verlag (1982) 337–351
12. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. Inform. and Comput. **98** (1992) 142–170
13. Biere, A., Cimatti, A., Clarke, E., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Proc. 36^{th} Conf. DAC '99. New Orleans, USA. ACM Press (21–25 June 1999) 317–320
14. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In Bruynooghe, M., Wirsing, M., eds.: Proc. 4^{th} Int. Symp. PLILP '92. Leuven, Belgium, 26–28 Aug. 1992, LNCS 631, Springer-Verlag (1992) 269–295
15. Cousot, P.: Partial completeness of abstract fixpoint checking, invited paper. In Choueiry, B., Walsh, T., eds.: Proc. 4^{th} Int. Symp. SARA '2000. Horseshoe Bay, USA, LNAI 1864. Springer-Verlag (2000) 1–25
16. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. TOPLAS **16** (1994) 1512–1542

17. Cleaveland, R., Iyer, P., Yankelevitch, D.: Optimality in abstractions of model checking. In Mycroft, A., ed.: Proc. 2^{nd} Int. Symp. SAS '95. Glasgow, UK, 25–27 Sep. 1995, LNCS 983. Springer-Verlag (1995) 51–63
18. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: 5^{th} POPL, Tucson, USA, ACM Press (1978) 84–97
19. Halbwachs, N.: About synchronous programming and abstract interpretation. Sci. Comput. Programming **31** (1998) 75–89
20. Graf, S., Loiseaux, C.: A tool for symbolic program verification and abstraction. In Courcoubetis, C., ed.: Proc. 5^{th} Int. Conf. CAV '93. Elounda, Grece, LNCS 697, Springer-Verlag (1993) 71–84
21. Ball, T., Podelski, A., Rajamani, S.K.: Relative Completeness of Abstraction Refinement for Software Model Checking. In Katoen, J.P., Stevens, P., eds.: Proc. 8^{th} Int. Conf. TACAS '2002. Grenoble, France, LNCS 2280, Springer-Verlag (2002)
22. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM **47** (2000) 361–416
23. Karr, M.: Affine relationships among variables of a program. Acta Informat. **6** (1976) 133–151
24. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks. In: 29^{th} POPL, Portland, USA, ACM Press (2002) 178–190
25. Cousot, P.: The Marktoberdorf'98 generic abstract interpreter. `http://www.di.ens.fr/~cousot/Marktoberdorf98.shtml` (1998)
26. Cousot, P.: Calculational design of semantics and static analyzers by abstract interpretation. NATO Int. Summer School 1998 on Calculational System Design. Marktoberdorf, DE. Organized by F.L. Bauer, M. Broy, E.W. Dijkstra, D. Gries and C.A.R. Hoare. (1998)
27. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Aut . Soft . Eng. **6** (1999) 69–95
28. Berezin, S., Clarke, E., Jha, S., Marrero, W.: Model checking algorithms for the μ-calculus. Tech. rep. tr-cmu-cs-96-180, Carnegie Mellon University, USA, (1996)
29. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. J. Logic Programming **13** (1992) 103–179 (The editor of J. Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).
30. Kaplan, M., Ullman, J.: A general scheme for the automatic inference of variable types. J. ACM **27** (1980) 128–145
31. Massé, D.: Combining forward and backward analyzes of temporal properties. In Danvy, ., Filinski, A., eds.: Proc. 2^{nd} Symp. PADO '2001. Århus, Danmark, 21–23 May 2001, LNCS 2053, Springer-Verlag (2001) 155–172
32. Granger, P.: Improving the results of static analyses of programs by local decreasing iterations. In Shyamasundar, R., ed.: Proc. 12^{th} FST & TCS. New Delhi, India, 18–20 Dec. 1992, LNCS 652, Springer-Verlag (1992) 68–79
33. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In Emerson, E., Sistla, A., eds.: Proc. 12^{th} Int. Conf. CAV '00. Chicago, USA, LNCS 1855, SPRINGER (2000) 154–169
34. Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples and refinements in abstract model-checking. In Cousot, P., ed.: Proc. 8^{th} Int. Symp. SAS '01. Paris, France, LNCS 2126, Springer-Verlag (2001) 356–373
35. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In Henzinger, T., Kirsch, C., eds.: Proc. 1^{st} Int. Workshop ESOP '2001. Volume 2211 of LNCS. Springer-Verlag (2001) 469Ñ–485