Abstract Testing versus (Abstract) Model-Checking

Notes prepared for the initial panel of the Schloß Ringberg Seminar on *Model Checking and Program Analysis* organized by Andreas PODELSKI, Bernhard STEFFEN, Moshe VARDI on February 20-23, 2000.

Patrick COUSOT and Radhia COUSOT

Département d'informatique
École normale supérieure
45 rue d'Ulm
75230 Paris cedex 05, France
Patrick.Cousot@ens.fr

Laboratoire d'informatique
École polytechnique
91128 Palaiseau cedex, France
rcousot@lix.polytechnique.fr

February 19, 2000

1 Introduction

The panelists serving on the initial panel of the Schloß Ringberg Seminar on *Model Checking and Program Analysis* are asked to answer and discuss questions prepared by the organizers (Andreas PODELSKI, Bernhard STEFFEN, Moshe VARDI). Before answering these questions in Sec. 4, certainly with the controversial bias of a program analysis perspective, we compare automatic formal methods for verifying that a program satisfies a specification. In order to restrict the scope of an already large debate, we consider abstract testing and model-checking only (excluding manual or computer-assisted proof methods as e.g. [75, 115, 114, 116, 127], including their combination with abstract interpretation and model-checking [93, 113, 112, 125, 128, 87, 122, 25]) [1,2].

[1] As noted by Amir PNUELI, the organizers possibly wanted the debate to be restricted to "Model Checking vs. Program Analysis", and intended to exclude "Deductive Verification" from the confrontation.

[2] For short, the references are necessarily partial.

2 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 [36, 37] such as array bound checking and the absence of run-time errors which was extended to liveness properties such as termination [29, 30].

Let $\langle S, t, I, F, E\rangle$ be a transition system [97] 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 [117]. Let t^{-1} be the inverse of relation t. Let $t^\star$ be the reflexive transitive closure of the binary relation t. Let $\text{post}[t]\,X$ 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: $\text{post}[t]\,X \overset{\text{def}}{=} \{s' \in S \mid \exists s \in X : \langle s, s'\rangle \in t\}$ [37, 30]. Inversely, let $\text{pre}[t]\,X \overset{\text{def}}{=} \text{pre}[t^{-1}]\,X$ 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. The specifications considered in [29] are of the form $\text{post}[t^\star]\,I \implies (\neg E) \wedge \text{pre}[t^\star]\,F$. Informally such a specification states that the descendants of the initial states are never erroneous and can potentially lead to final states.

Consider for example, the factorial program (the random assignment `?` is equivalent to a read 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 [34, 33] below shows that the condition $\texttt{n} \geq 0$ 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 $\texttt{f} \geq 1$ (Ω, typed `_O_`, denotes the uninitialized value while `+oo` is the greatest machine integer):

```
0: { n:_O_; f:_O_ }
  n := ?;
1:!: { n:[0,+oo]; f:_O_ }
  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] }
```

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 [11] to $\mathrm{post}[t^\star]\, I \implies \mathrm{Iv} \wedge \mathrm{pre}[t^\star]\, \mathrm{It}$ under the name "Abstract debugging". 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 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 $\mathrm{Iv} = \{\langle p, m\rangle \mid p \in \mathrm{Pv} \implies \mathrm{Iv}_p(m)\}$ and $\mathrm{It} = \{\langle p, m\rangle \mid p \in \mathrm{Pt} \wedge \mathrm{Iv}_p(m)\}$. Otherwise stated, the descendants of the initial states always satisfy all local invariant assertions and can potentially lead to states satisfying some local intermittent assertion.

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.

3 Differences between abstract model-checking and abstract testing

At first sight, abstract testing is model-checking [19, 121] of the temporal formula $\Box(\bigwedge_{p\in\mathrm{Pv}} at_p \implies \mathrm{Iv}_p) \wedge \Diamond(\bigvee_{p\in\mathrm{Pt}} at_p \wedge \mathrm{It}_p)$ for a small-step operational semantics $\langle S,\, t,\, I\rangle$ of the program, or more precisely, abstract model-checking [20, 23, 96] 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 reasoning close to tracing program execution whence are easily understandable by programmers. However abstract testing is quite different from (abstract) model-checking for at least six technical reasons explained below.

3.1 Scope of application

3.1.1 Scope of abstract testing

First, the abstract interpreters are developed for programming languages that is infinitely many programs, with modular and recursive control and data infinitary structures which are difficult to abstract and are most often ignored in model-checking (with peculiar exceptions involving complete abstractions, such as e.g. [7]).

In order to apply abstract testing to a great variety of programming languages, abstract interpreter generators have been developed (see e.g. [60]).

The (generated) abstract interpreters are generic [35, 60, 101, 102, 120], that is parameterized by an abstract domain specifying the considered approximated properties.

The price to be paid for this generality is that there can be no fine tuning of the abstract interpreter for a particular specification and a particular program (abstract compilation, see e.g. [8], improving mainly the performance rather than the precision of the analyses).

3.1.2 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" [21]. 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. [84, 85, 86, 88, 89].

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 [42]). However, the proper abstraction may be quite difficult to find in practice [73, 104].

3.2 Abstract semantics

Second, the only abstractions considered in abstract model checking [110, 126] 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 [45, 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 [36, 37] or the polyhedral abstraction [47]) 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. [14, 59, 78, 79, 80, 82, 83, 90, 92]. 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. 2.

3.3 The need for infinite abstract domains

Third, infinite abstract domains are definitely needed in program analysis for precision (and sometimes efficiency or ease of programming). The argument given in [42] uses reachability analysis with the attribute-independent interval domain [36, 37] 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:_O_ }
  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:_O_ }
  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 $\{\Omega, [0, \mathtt{n}], [0, \mathtt{n}-1], [1, \mathtt{n}], [\mathtt{n}, \mathtt{n}]\}$) 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 [3].

[3]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 [77] based on heuristics for discovering invariants from the program test which leads to worse results than [36], as shown in [41].

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 [73, 104, 125, 74], 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 [54]). Note that this is different from using abstract interpretation or model-checking to help a prover/proof checker to infer invariants [93, 113, 112, 118, 123, 125, 128, 87, 122] or to guide the automatic prover in its proof search [65].

3.4 Precise checking in the presence of approximations

Fourth, and 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 [36, 37], can be simply ignored in model-checking of finite-state transition systems.

3.4.1 Fixpoint approximation check

A first illustration of this fourth point 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 $\mathrm{lfp}^{\subseteq} F = \mathrm{post}[t^\star]\, E$.

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 [44, p. 73] and Sec. 3.6 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 [30, 36, 37]. 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,* $F \in L \xmapsto{mon} L$ *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 [131, 38]. 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 [36, 37] 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 [33] is too approximate to statically check that the index `i` is within the array bounds 1 and 100 :

```
0: { i:_O_ }
  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 $0 < \texttt{i} < 100$ (the evaluation of the runtime check `always B` has the effect of blocking the program execution when the assertion `B` does not hold):

```
# 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:_O_ }
  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.

3.4.2 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 [3]) 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:_O_; f:_O_ }
  n := ?;
1: { n:[-oo,+oo]; f:_O_ }
  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:_O_; f:_O_ }
  n := ?;
1: { n:[-oo,+oo]; f:_O_ }
  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 [29, 41]. 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:_O_; f:_O_ }
  n := ?;
1:!: { n:[0,+oo]; f:_O_ }
  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 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 [29]. It is calculated as the limit of the *alternating fixpoint computation*:

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

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

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

For soundness, we assume:

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

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

$$= \quad \mathrm{lfp}^{\sqsubseteq}\, \boldsymbol{\lambda} X \cdot (\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B \sqcap B(X)) \tag{8}$$
$$= \quad \mathrm{lfp}^{\sqsubseteq}\, \boldsymbol{\lambda} X \cdot (\mathrm{lfp}^{\sqsubseteq} F \sqcap \mathrm{lfp}^{\sqsubseteq} B \sqcap F(X)) \tag{9}$$

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/up 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 [95] to infer types in flowchart programs.

Let us recall the following classical results in abstract interpretation [37, 39]:

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* $F \in L \xmapsto{mon} L$*, 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 [131], the least fixpoints exist. So let $P^\sharp \stackrel{\text{def}}{=} \mathrm{lfp}^{\sqsubseteq^\sharp} \alpha \circ F \circ \gamma$. 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,* $F^\sharp, \bar{F}^\sharp \in L^\sharp \xmapsto{mon} L^\sharp$*, 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 [131]. ■

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,* $F \in L \xmapsto{mon} L$ *and* $B \in L \xmapsto{mon} L$ *satisfy the hypotheses (8) and (9),* $F^\sharp \in L^\sharp \xmapsto{mon} L^\sharp$, $B^\sharp \in L^\sharp \xmapsto{mon} L^\sharp$, $\alpha \circ F \circ \gamma \sqsubseteq^\sharp F^\sharp$, $\alpha \circ B \circ \gamma \sqsubseteq^\sharp B^\sharp$ *and the sequence* $\langle \dot{X}^n, n \in \mathbb{N}\rangle$ *is defined by (3), (4) and (5) then* $\forall k \in \mathbb{N} :$ $\alpha(\mathrm{lfp}^{\subseteq} F \cap \mathrm{lfp}^{\subseteq} B) \sqsubseteq^\sharp \dot{X}^{k+1} \sqsubseteq^\sharp \dot{X}^k$. □

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 r™les of F and B. ■

It is interesting to note that the computed sequence (3), (5) and (8) is optimal (see [38]).

A similar result holds when replacing one least fixpoint by a greatest fixpoint[4].

If the abstract lattice does not satisfy the descending chain condition then [29] also suggests to use a narrowing operator $\triangle$ [36, 37] 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 $\boldsymbol{\lambda} X \cdot \dot{X}^{2n+1} \sqcap B^\sharp(X)$.

3.4.3 Fixpoint meet approximation

A third illustration of the difference between model-checking and abstract testing algorithms is the local iterations to handle tests, backward assignments, etc. For example, without local iterations:

```
# IT.analysis ();;
Forward analysis from initial states;
0: { x:_O_; y:_O_; z:_O_ }
```

[4]Damien Massé, private communication.

```
  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 {((((x < y) | (y < x)) | ((y < z) | (z < y))) | (((z + 1) < x) | (x < (z + 1))))}
    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] }
```

while the precision is greatly enhanced with local iterations:

```
# IT'.analysis ();;
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 {((((x < y) | (y < x)) | ((y < z) | (z < y))) | (((z + 1) < x) | (x < (z + 1))))}
    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] }
```

3.4.4 Fixpoint meet approximation check

The abstract testing strategy to check $\mathrm{post}[t^\star]\, I \implies \mathrm{Iv} \wedge \mathrm{pre}[t^\star]\, \mathrm{It}$ and more generally $\mathrm{lfp}^{\sqsubseteq} F \sqsubseteq I \sqcap \mathrm{lfp}^{\sqsubseteq} B$ combines the results of Sec. 3.4.1 and Sec. 3.4.3.

3.5 Counter-examples to erroneous designs

The fifth 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 [15]. 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 $\texttt{n} \geq 0$:

```
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 $\texttt{n} < 0$, 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 $\texttt{n} < 0$:

```
#  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-ime error ($\bot$, typed `_|_`, is the false invariant hence denotes irreachability in forward analysis and impossibility to reach the goal in backward analysis):

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

3.6 Contrapositive reasoning

For the sixth 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 [44, p. 73], the Galois connection $\langle \wp(S), \subseteq\rangle \underset{\mathrm{post}[r]}{\overset{\widetilde{\mathrm{pre}}[r]}{\leftrightarrows}} \langle \wp(S), \subseteq\rangle$ (where $r \subseteq S \times S$ and $\widetilde{\mathrm{pre}}[r]\, X \stackrel{\mathrm{def}}{=} \{s \mid \forall s' : \langle s, s'\rangle \in r \implies s' \in X\}$) implies that the invariance specification check $\mathrm{post}[t^\star]\, E \subseteq I$ is equivalent to $\widetilde{\mathrm{pre}}[t^\star]\, \neg I \subseteq \neg E$ (or $\mathrm{pre}[t^\star]\, \neg I \subseteq \neg E$ for total deterministic transition systems [30]). Otherwise stated a forward positive proof is equivalent to a backward contrapositive proof, as observed in [40]. So the difference between the abstract testing algorithm of [37, 39, 30] and the model-checking algorithm of [17, 18, 121] is that abstract testing checks $\mathrm{post}[t^\star]\, E \subseteq I$ while model-checking verifies $\widetilde{\mathrm{pre}}[t^\star]\, \neg I \subseteq \neg E$, which is equivalent for finite transition systems as considered in [17, 18, 121].

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 [36, 37] 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 [63]. To start with, the abstract domain might be enriched with complements [62], but this might not be sufficient and indeed the abstract domain might have to be enriched for each primitive operation [64], thus leading to an abstract algebra which might be quite difficult to implement if not totally inefficient.

4 Tentative and controversial answers to the questions to the panelists

Question 1:

Describe your view of the strength and weakness of the two communities (model checking and program analysis). Try in particular to reveal, from your point of view, similarities & distinctions in

- *philosophy,*
- *technique;*

The model-checking community is much larger and well-organized than the program analysis community. This does not completely explain why there are many more available tools and publications in model-checking than in program analyzsis in the academic community (e.g. compare the tool sections of the CAV [22, 99, 119, 28, 58, 136, 2, 76, 91, 81] and TACAS [13, 105, 12, 129, 129, 26] proceedings with those of the SAS proceedings [5, 6, 46, 100, 111, 48, 133, 103, 27]).

The techniques used in model-checking, such as fixpoint approximation, are often close to, if not directly borrowed, from program analysis. So one may wonder why they are immediately applied with success in model-checking while there are less direct applications in program analysis and a number of such analyzes simply fail.

One reason might be that it is easier (e.g. for a few PhD students) to design and implement a model-checker for e.g. linear hybrid systems which can be tested on impressive but limited examples, rather than a program analyzer for e.g. a full language such as ADA, C or Java [132] which have to be tested on very large programs (presently 100 000 [98] to 1 400 000 lines [54] not speaking of larger programs such as the 30 000 000 of MS Windows 2000 and its anticipated 63 000 bugs).

On the other hand and as noted in Sec. 3.1.1, program analyzers are often automatically generated, generic or for large languages so are more difficult to design and implement. However, it is very easy to exhibit the deficiencies of such program analysers. To check a program analyzer, say for C, you have thousands of public-domain programs available, no specification is needed and you are not allowed to change them for the purpose of the analysis (just may be to adapt the analyzer to the specificities of the C-dialect which is used).

It is much more very difficult to evaluate a model checker. You have to find an example, built a model, write a specification and then get out of memory (that's the easy part). Then you are told than your model is not the proper one and you go on (Klaus coined the term "Havelund's sledge-hammer").

Finally the scope of the abstraction techniques is rather wide (from program analysis, to typing [32] and hierarchies of semantics [31, 61]), researchers in program analysis might be too often tempted to increase their scientific productivity or publication rate by forgetting about long-term practical and usable implementations and turning to more theoretical subjects.

Question 2.a:

(Where) do you see a bridge between the respective fields?

A trivial answer is for program analysisto consider more properties (but is there anything useful beyonds reachability, invariance and ancestry analysis?) and for model-checking to consider really large, complex and parameterized systems which models are already existing or difficult to design (e.g. C programs over 100 000 lines versus boilers or water-tanks).

One way wonder whether model-checking techniques do scale up for such systems programmed in C. Do they ave a reasonable specification in temporal logic? including data? including interaction with the environment? Model checking would have to consider rich models of program semantics (much richer than e.g. only (finite) automata) in order to bridge the gap between the respective fields.

Since approximation is the only possibility in program analysis and thus idea is not (yet) well accepted by the model-checking community, cross-fertilization in the other direction is (still) unfruitful.

Question 2.b:

(What) are the obstacle to building such bridges? Try to give an example where you are convinced that the other community's method will fail to capture appropriately.

The main obstacle seem to be the difference of nature of the considered models. For example, partial order reduction [130] has hardly (can hardly?) be used in program analysis. Nevertheless geometric models, may be better adapted to describe true concurrency, although much more complex, have proved equally useful and efficient in the analysis of parallel functional programs [53, 49, 51, 50, 52, 72, 66, 68, 67, 69, 71, 70].

The necessity imposed by model checkers to map the language semantic model to a (finite) transition system may be a the cause of a considerable loss of precision as discussed in Sec. 3.3. See for another example the difference in precision between [24] and [134, 135] for the π-calculus. Other model-checkers for approximation of the infinite space of reachable states are mainly reuses or adaptation of classical abstract interpretation techniques, see Sec. 3.2, and compare for example [14] and [94].

Question 2.c:

(what) are your general worries and hopes concerning possible synergy?

One concern is the impossibility or difficulty to generalize the results of model-checking to program analysis. There is a gap between finite and infinite systems.

One aspect is sociological. After so much insistance on "sound and complete computer aided verification by formal methods", it is difficult to accept the idea of "sound and incomplete automatic partial checking by approximation methods".

Another aspect is technical. We have illustrated in Sec. 3.4.3 and 3.4.1 the problems with fixpoint approximation for infinite systems. We have also illustrated similar problems with negation in Sec. 3.6. For another example, consider the $\exists$ modalities which are trivial to handle in the finite case but more difficult in the infinite case. Many present solutions e.g. consists only in trying to execute a few examples (i.e. bread first testing), see e.g. [14]. More generally, the approximation of fixpoints from below is difficult (and mainly consist up to now in considering a few runs or reasoning variant functions trivially inferred from the program text (e.g. data types)).

Question 2.d:

What do you expect from this meeting?

A lot of contraversial discussions on these questions.

Question 3:

Is model checking mainly theory (the theory of temporal logics), whereas data flow analysis is "hands on"?

For the model-checking papers published in LICS certainly. Many complexity results in model-checking are also of poor help in practice. In the worst case everything is proportional to the input transition system and so will go out of memory for any reasonable program. The only way is therefore the interactive exploration of a few paths, that is program testing.

If data flow analysis is "hands on", is applied model-checking something more than boolean data flow analysis? or program testing?

A similar question is "Is abstract interpretation mainly theory (the theory of semantic approximation), whereas data flow analysis is 'hands on'?".

Question 4:

What has become applicable to other than toy examples just by coincidence is only due to the event of BDDs and fast computers and has nothing to do with the theory behind model checking ('all' practical success stories (errors found) are based on reachability, and not on sophisticated temporal properties).

Model-checking has always been a success even before the advent of BDDs. Indeed it is very difficult to find a failure story in the CAV [22, 99, 119, 28, 58, 136, 2, 76, 91, 81] or TACAS [13, 105, 12, 129, 129, 26] proceedings. So BDDs and now SAT [4] are part of the progress, since the theory behind model checking does not change too much. Indeed one may wonder, in the case of infinite systems, if there are model-checking success stories including a full verification of a full system. Is finding errors really significant? Program testing would be a great collection success stories if finding errors in programs were considered an achievement.

For program analysis, the situation is more difficult since the verification is partial. However their are success stories with error found, invariants verified and very few uncertainty cases left out [57, 98]. However one program is not enough, since the analyzer must work rather well on all writenable programs.

Question 5:

Are the techniques, heuristics and algorithms developed for model checking just on eliminating the redundancy inherent to the interleaving semantics of concurrent systems? And hence, not applicable to 'functional' software systems, where concurrency is mostly deferred to the operating system?

Not all concurrency is mostly deferred to the operating system. For example in real-time critinal systems (in avionics, automotive, healthcare, etc.), it is important to check timing constraints. Can the numerous model checking methods which have been developped for timed systems be applied to analyze a real-time program writen in C running on a pipelined processor with caches? Then this could then be compared with what is done and commercialized in program analysis [1].

Question 6:

Does it make sense to view a software system like MS Word as a transition system, which is the only thing that model checking can be applied to?

It is remarkable that both abstract interpretation [29, 39] and then model-checking [17, 121] used transition systems (originating from [97] to model the systems to be analyzed. For program analysis, they work very well for languages such as Prolog (whence may the large use of abstract interpretation in Prolog [56, 55]) and C. MS Word is certainly a transition system, even a procedural flow chart. We do not know on any attempt to model-check it. To get a temporal specification of what happens when one types command would be great (and may be more costly to develop and maintain than MS Word itself). However, MS Word is definitely within the scope of automatic point-to analysis, see [54].

Question 7:

Should it not be viewed as a functional program? If yes, can model checking methods be reasonably transferred to functional programs? Can control flow analysis *methods be used to enhance model checking methods (traditionally applied to transition systems)?*

The limits of transitions sytems do not really appear in model checking, since the systems to be checked (e.g. a protocol) do not involve higher-order recursion (as in functional programs for which, e.g. to handle strictness analysis, the theory of abstract interpretation has to use both an approximation and a computational ordering [43]). Note that MS Word is (certainly) not higher-order and the handling of first-order procedures is rather well-understood and can be very precise, see among many others [9, 10]. It is clear also that the simple modelisation of operations on heap-allocated recursive data structures as transition systems or abstract machines are not well suited for program analysis [124]. So model checking methods can hardly be reasonably transferred to higher-order functional programs with say denotational or SOS semantics (as needed in e.g. strictness analysis).

Reciprocally, in order to enhance model checking by program analysis methods for higher-order control and data recursion, it seems that the specification methods used by model checking such as temporal logic are not very well-adapted to describe such control and data structures. Moreover programming languages have also specific problems not appearing in model-checking such as naming (recall [16]), etc. So temporal specification languages//logics, which are well-adapted to programs and scale-up for large ones have to be found.

Question 8:

Are BDDs useful for software, or is viewing software as a finite-state system a "looser right from the start"?

We have no hope in viewing software as a finite-state system and neither in model-checking software with success stories different from successful tests. The same question could have been asked for the automata theoretic approach to model-checking for which we know no report of failure by immediate size explosion in the model-checking literature, which seems nevertheless quite frequent in program analysis.

BDDs have already shown useful for software, e.g. for strictness analysis (after proper encoding of higher-order boolean functions), see [106, 107]. Must be exploitable in case of explosion (widening). However the generalization beyond trivial boolean properties so as to include also, e.g. tree automata in an expressive (e.g. fairness) and efficient way turned out to be a non-trivial task [108, 109].

References

[1] M. Alt, Ferdinand C., F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In R. Cousot and D.A. Schmidt, editors, *Proceedings*

of the Third International Symposium on Static Analysis, SAS '96, Aachen, Germany, 20–22 september 1996, Lecture Notes in Computer Science 1145, pages 52–66. Springer-Verlag, Berlin, Germany, 1996. 21

[2] R. Alur and T.A. Henzinger, editors. *Proceedings of the Eight International Conference on Computer Aided Verification, CAV '96*, New Brunswick, New Jersey, United States, Lecture Notes in Computer Science 1102. Springer-Verlag, Berlin, Germany, 31 July – 3 August 1996. 18, 21

[3] 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. 8

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '99*, Amsterdam, Netherlands, 22-25 March 1999, Lecture Notes in Computer Science 1579. Springer-Verlag, Berlin, Germany, 1999. 21

[5] M. Billaud, P. Castéran, M.-M. Corsini, K. Musumbu, and A. Rauzy, editors. *Actes JTASPEFL '91, LaBRI, Bordeaux, France*, volume 74 of *BIGRE*. IRISA, Campus de Beaulieu, Rennes, France, October 1991. 18

[6] M. Billaud, P. Castéran, M.-M. Corsini, K. Musumbu, and A. Rauzy, editors. *Proceedings of the Second International Workshop on Static Analysis, WSA '92, LaBRI, Bordeaux, France*, volume 81–82 of *BIGRE*. IRISA, Campus de Beaulieu, Rennes, France, 23–25 september 1992. 18

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

[8] D. Boucher and M. Feeley. Abstract compilation: A new implementation paradigm for static analysis. In T. Gyimothy, editor, *Proceedings of the Sixth International Conference on Compiler Construction, CC '96*, Linköping, Sweden, Lecture Notes in Computer Science 1060, pages 192–207. Springer-Verlag, Berlin, Germany, 24–26 April 1996. 4

[9] F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In P. Deransart and J. Małuszyński, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '90*, Linköping, Sweden, Lecture Notes in Computer Science 456, pages 307–323. Springer-Verlag, Berlin, Germany, 20–22 August 1990. 22

[10] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992. 22

[11] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 46–55. ACM Press, New York, New York, United States, 1993. 3

[12] E. Brinksma, editor. *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '97*, Enschede, Netherlands, Lecture Notes in Computer Science 1217. Springer-Verlag, Berlin, Germany, 2–4 April 1997. 18, 21

[13] E. Brinksma, R. Cleaveland, K.G. Larsen, , T. Margaria, and B. Steffen, editors. *Proceedings of the First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '95*, Aarhus, Denmark, Lecture Notes in Computer Science 1019. Springer-Verlag, Berlin, Germany, 19–20 May 1995. 18, 21

[14] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In O. Grumberg, editor, *Proceedings of the Ninth International Conference on Computer Aided Verification, CAV '97*, Haifa, Israel,Lecture Notes in Computer Science 1254, pages 400–411. Springer-Verlag, Berlin, Germany, 22–25 July 1997. 4, 19, 20

[15] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992. 13

[16] E.M. Clarke. Programming language constructs for which it is impossible to obtain "good" hoare-like axiom systems. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 10–20. ACM Press, New York, New York, United States, January 1977. 22

[17] E.M. Clarke and E.A. 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. 15, 21

[18] E.M. Clarke, E.A. Emerson, and A.P. 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*, pages 117–126. ACM Press, New York, New York, United States, January 1983. 15

[19] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986. 3

[20] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, september 1994. 3

[21] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, United States, 1999. 4

[22] E.M. Clarke and R.P. Kurshan, editors. *on Computer Aided Verification, CAV '90*, number 3 in New Brunswick, New Jersey, United States, DIMACS Volume Series. American Mathematical Society, Providence, Rhode Island, United States, June 1990. 18, 21

[23] R. Cleaveland, P. Iyer, and D. Yankelevitch. Optimality in abstractions of model checking. In A. Mycroft, editor, *Proceedings of the Second International Symposium on Static Analysis, SAS '95*, Glasgow, United Kindom, 25–27 september

1995, Lecture Notes in Computer Science 983, pages 51–63. Springer-Verlag, Berlin, Germany, 1995. 3

[24] W.R. Cleaveland, editor. *Finite State Verification for the Asynchronous π-Calculus*, Amsterdam, Netherlands, Lecture Notes in Computer Science 1579. Springer-Verlag, Berlin, Germany, 22–28 March 1999. 19

[25] W.R. Cleaveland, editor. *On Proving Safety Properties by Integrating Static Analysis, Theorem Proving and Abstraction*, Amsterdam, Netherlands, Lecture Notes in Computer Science 1579. Springer-Verlag, Berlin, Germany, 22–28 March 1999. 1

[26] W.R. Cleaveland, editor. *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '99*, Amsterdam, Netherlands, Lecture Notes in Computer Science 1579. Springer-Verlag, Berlin, Germany, 22–28 March 1999. 18, 21

[27] A. Cortesi and G. Filé, editors. *Proceedings of the Sixth International Symposium on Static Analysis, SAS '99*, Venice, Italy, 22–24 september 1999, Lecture Notes in Computer Science 1694. Springer-Verlag, Berlin, Germany, 22–24 september 1999. 18

[28] C. Courcoubetis, editor. *Proceedings of the Fifth International Conference on Computer Aided Verification, CAV '93*, Elounda, Greece, Lecture Notes in Computer Science 697. Springer-Verlag, Berlin, Germany, 28 June – 1 July 1993. 18, 21

[29] 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. 2, 10, 12, 21

[30] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, United States, 1981. 2, 7, 15

[31] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997. URL: http://www.elsevier.nl/locate/entcs/volume6.html, 25 pages. 18

[32] P. Cousot. Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press, New York, New York, United States. 18

[33] 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. 2, 7

[34] P. Cousot. The Marktoberdorf'98 generic abstract interpreter. http://www.di.ens.fr/~cousot/Marktoberdorf98.shtml, November 1998. 2

[35] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, volume 173, pages 421–505. NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, Amsterdam, The Netherlands, 1999. 4

[36] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976. 2, 4, 5, 6, 7, 12, 15

[37] 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*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, New York, United States. 2, 4, 5, 6, 7, 11, 12, 15

[38] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979. 7, 12

[39] 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*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, New York, United States. 11, 15, 21

[40] P. Cousot and R. Cousot. Induction principles for proving invariance properties of programs. In D. Néel, editor, *Tools & Notions for Program Construction*, pages 43–119. Cambridge University Press, Cambridge, United Kindom, 1982. 15

[41] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):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.dmi.ens.fr/~cousot`.). 5, 10

[42] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992. 4, 5

[43] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 95–112, Toulouse, France, 16–19 May 1994. IEEE Computer Society Press, Los Alamitos, California, United States. 22

[44] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6:69–95, 1999. 6, 15

[45] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Massachusetts, January 2000. ACM Press, New York, New York, United States. 4

[46] P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors. *Proceedings of the Third International Workshop on Static Analysis, WSA '93*, Padova, Italy, Lecture Notes in Computer Science 724. Springer-Verlag, Berlin, Germany, 22–24 september 1993. 18

[47] 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*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, New York, United States. 4

[48] R. Cousot and D.A. Schmidt, editors. *Proceedings of the Third International Symposium on Static Analysis, SAS '96*, Aachen, Germany Lecture Notes in Computer Science 1145. Springer-Verlag, Berlin, Germany, 24–26 september 1996. 18

[49] R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '95*, La Jolla, California, 21–23 June 1995. ACM Press, New York, New York, United States. 19

[50] R. Cridlig. Implementing a static analyzer of concurrent programs: Problems and perspectives. In M. Dam, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, Stockhlom, Sweden, 24–26 June 1996, Lecture Notes in Computer Science 1192, pages 244–259. Springer-Verlag, Berlin, Germany, 1996. 19

[51] R. Cridlig. Semantic analysis of Concurrent ML by abstract model-checking. In B. Steffen and T. Margaria, editors, *Proceedings of the International Workshop on Verification of Infinite State Systems*. vol. MIP-9614, Universität Passau, Germany, August 1996. To be published in Electronic Notes on Theoretical Computer Science, 1997. 19

[52] R. Cridlig. Semantic analysis of Concurrent ML by abstract model-checking. *Electronic Notes in Theoretical Computer Science*, 5, 1996. URL: `http://www.elsevier.nl/locate/entcs/volume5.html`, nn pages. 19

[53] R. Cridlig and É. Goubault. Semantics and analysis of Linda-based languages. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis, WSA '93*, Padova, Italy, Lecture Notes in Computer Science 724, pages 72–86. Springer-Verlag, Berlin, Germany, 22–24 september 1993. 19

[54] 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*, page 1, Boston, Massachusetts, United States, 22–23 January 2000. ACM Press, New York, New York, United States. 6, 18, 21

[55] S.K. Debray. Formal bases for dataflow analysis of logic programs. In G. Levi, editor, *Advances in Logic Programming Theory*, International Schools for Computer Scientists, section 3, pages 115–182. Clarendon Press, Oxford, United Kindom, 1994. 21

[56] S. Debray, editor. Special issue on abstract interpretation. *Journal of Logic Programming*, 13(2–3), 1992. 21

[57] A. Deutsch, G. Gonthier, and M. Turin. La vérification des programmes d'ariane. *Pour la Science*, 243:21–22, January 1998. (in French). 21

[58] D.L. Dill, editor. *Proceedings of the Sixth International Conference on Computer Aided Verification, CAV '94*, Stanford, California, United States, Lecture Notes in Computer Science 818. Springer-Verlag, Berlin, Germany, 21–23 June 1994. 18, 21

[59] D.L. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In P. Wolper, editor, *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95*, Liège, Belgium, Lecture Notes in Computer Science 939, pages 409–422. Springer-Verlag, Berlin, Germany, 3–5 July 1995. 4

[60] C. Ferdinand. *Generating Program Analyzers*. Verfasser – Pirrot Verlag, Saarbrücken, Germany, 1999. 4

[61] R. Giacobazzi. "optimal" collecting semantics for analysis in a hierarchy of logic program semantics. In C. Puech and R. Reischuk, editors, *Proceedings of the STACS '96*, Lecture Notes in Computer Science 1046, pages 503–514. Springer-Verlag, Berlin, Germany, 1996. 18

[62] R. Giacobazzi, C. Palamidessi, and F. Ranzato. Weak relative pseudo-complements of closure operators. *Algebra Universalis*, 36(3):405–412, 1996. 17

[63] R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: A domain perspective. In M. Johnson, editor, *Proc. of the Sixth International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 231–245. Springer-Verlag, Berlin, Germany, 1997. 17

[64] R. Giacobazzi, F. Ranzato, and F. Scozzari. Complete abstract interpretations made constructive. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Proceedings of the Twentythird International Symposium on Mathematical Foundations of Computer Science, MFCS'98*, volume 1450 of *Lecture Notes in Computer Science*, pages 366–377. Springer-Verlag, Berlin, Germany, 1998. 17

[65] F. Giunchiglia and A. Villafiorita. ABSFOL: A proof checker with abstraction. In M.A. McRobbie and J.K. Slaney, editors, *Proceedings of the Thirteenth International Conference on Automated Deduction, CADE '962*, New Brunswick, New Jersey, United States, Lecture Notes in Computer Science 1104, pages 136–140. Springer-Verlag, Berlin, Germany, 30 July – 3 August 1996. 6

[66] É. Goubault. Domains of higher-dimensional automata. In E. Best, editor, *CONCUR '93, Proceedings of the Fourth International Conference on Concurrency Theory*, Hildesheim, Germany, August 1993, Lecture Notes in Computer Science 715, pages 293–307. Springer-Verlag, Berlin, Germany, August 1993. 19

[67] É. Goubault. *Geometry of Concurrency*. Thèse de doctorat de l'École Polytechnique en informatique, École polytechnique, Palaiseau, France, 13 November 1995. 19

[68] É. Goubault. Schedulers as abstract interpretations of higher-dimensional automata. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '95*, La Jolla, California, pages 134–145. ACM Press, New York, New York, United States, 21–23 June 1995. 19

[69] É. Goubault. Duration for truly-concurrent transitions. In H. Riis Nielson, editor, *Proceedings of the Sixth European Symposium on Programming, ESOP '96*, Lecture Notes in Computer Science 105, pages 173–187, Linköping, Sweden, 22–26 April 1996. Springer-Verlag, Berlin, Germany. 19

[70] É. Goubault. Optimal implementation of wait-free binary relations. In *Proceedings of the Twentysecond on Trees in Algebra and Programming, CAAP*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 1997. 19

[71] É. Goubault. A semantic view on distributed computability and complexity. In *Proceedings of the Third Theory and Formal Methods Section Workshop*. Imperial College Press, London, United Kindom, 1997. 19

[72] É. Goubault and T.P. Jensen. Homology of higher dimensional automata. In W.R. Cleaveland, editor, *CONCUR '92, Proceedings of the Third International Conference on Concurrency Theory*, Stony Brook, , New York, August 1992, Lecture Notes in Computer Science 630, pages 254–268. Springer-Verlag, Berlin, Germany, August 1992. 19

[73] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In C. Courcoubetis, editor, *Proceedings of the Fifth International Conference on Computer Aided Verification, CAV '93*, Elounda, Greece, Lecture Notes in Computer Science 697, pages 71–84. SPRINGER, 28 June –1 July 1993. 4, 6

[74] S. Graf and H. Saï di. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proceedings of the Ninth International Conference on Computer Aided Verification, CAV '97*, Haifa, Israel,Lecture Notes in Computer Science 1254, pages 72–83. Springer-Verlag, Berlin, Germany, 22–25 July 1997. 6

[75] E.P. Gribomont and D. Rossetto. CAVEAT: Technique and tool for computer aided verification and transformation. In P. Wolper, editor, *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95*, Liège, Belgium, Lecture Notes in Computer Science 939, pages 70–83. Springer-Verlag, Berlin, Germany, 3–5 July 1995. 1

[76] O. Grumberg, editor. *Proceedings of the Ninth International Conference on Computer Aided Verification, CAV '97*, Haifa, Israel,Lecture Notes in Computer Science 1254. Springer-Verlag, Berlin, Germany, 22–25 July 1997. 18, 21

[77] R. Gupta. A fresh look at optimizing array bound checking. In *ACM-SIGPLAN Conference on Programming Language Design and Implementation '90*, pages 272–282, June 1990. 5

[78] N. Halbwachs. Delays analysis in synchronous programs. In C. Courcoubatis, editor, *Proceedings of the Fifth International Conference on Computer Aided Verification, CAV '93*, Elounda, Greece, Lecture Notes in Computer Science 697, pages 333–346. Springer-Verlag, Berlin, Germany, 28 June –1 July 1993. 4

[79] N. Halbwachs. About synchronous programming and abstract interpretation. In B. Le Charlier, editor, *Proceedings of the First International Symposium on Static Analysis, SAS '94*, Namur, Belgium, 20–22 september 1994, Lecture Notes in Computer Science 864, pages 179–192. Springer-Verlag, Berlin, Germany, 1994. 4

[80] N. Halbwachs. About synchronous programming and abstract interpretation. *Science of Computer Programming*, 31(1):75–89, May 1998. 4

[81] N. Halbwachs and D. Peled, editors. *Proceedings of the Eleventh International Conference on Computer Aided Verification, CAV '99*, Trento, Italy, Lecture Notes in Computer Science 1633. Springer-Verlag, Berlin, Germany, 6–10 July 1999. 18, 21

[82] N. Halbwachs, J.-É. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. Le Charlier, editor, *Proceedings of the First International Symposium on Static Analysis, SAS '94*, Namur, Belgium, 20–22 september 1994, Lecture Notes in Computer Science 864, pages 223–237. Springer-Verlag, Berlin, Germany, 1994. 4

[83] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997. 4

[84] K. Havelund, K.G. Larsen, and A. Skou. Formal verification of an audio/video power controller using the real-time model checker UPPAAL. In *ARTS'99*, 1999. 4

[85] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space craft controller using SPIN. In *SPIN'98*, 1998. 4

[86] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2000. toappear. 4

[87] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M.C. Gaudel and J. Woodcock, editors, *Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, FME '96: Industrial Benefit of Formal Methods*, Oxford, United Kindom, Lecture Notes in Computer Science 1051, pages 662–681. SPRINGER, 18–22 March 1996. 1, 6

[88] K. Havelund and J. Skakkebaek. Practical application of model checking in software verification – a case study using Java PathFinder. In *FM'99*, 1999. 4

[89] K. Havelund, A. Skou, K.G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *RTSS'97*, 1997. 4

[90] P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In P. Wolper, editor, *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95*, Liège, Belgium,Lecture Notes in Computer Science 939, pages 381–394. Springer-Verlag, Berlin, Germany, 3–5 July 1995. 4

[91] A.J. Hu and M.Y. Vardi, editors. *Proceedings of the Tenth International Conference on Computer Aided Verification, CAV '98*, Vancouver, British Columbia, Canada,Lecture Notes in Computer Science 1427. Springer-Verlag, Berlin, Germany, 28 June – 2 July 1998. 18, 21

[92] 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*, pages 261–272, Paris, France, 27–29 september 1999. ACM Press, New York, New York, United States. 4

[93] H. Hungar. Combining model checking and theorem proving to verify parallel processes. In C. Courcoubetis, editor, *Proceedings of the Fifth International Conference on Computer Aided Verification, CAV '93*, Elounda, Greece, Lecture

Notes in Computer Science 697, pages 154–165. SPRINGER, 28 June –1 July 1993. 1, 6

[94] P. Jouvelot. Semantic parallelization: a practical exercise in abstract interpretation. In *Conference Record of the Fourteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–48, Munich, Germany, 21–23 January 1987. ACM Press, New York, New York, United States. 19

[95] M. Kaplan and J.D. Ullman. A general scheme for the automatic inference of variable types. *Journal of the Association for Computing Machinary*, 27(1):128–145, 1980. 11

[96] P. Kelb. Model checking and abstraction: A framework approximating both truth and failure information. Technical report, University of Oldenburg, 1994. 3

[97] R.M. Keller. Formal verification of parallel programs. *Communications of the Association for Computing Machinary*, 19(7):371–384, July 1976. 2, 21

[98] P. Lacan, J.N. Monfort, Le Vinh Quy Ribal, A. Deutsch, and G. Gonthier. The software reliability verification process: The ARIANE 5 example. In *Proceedings DASIA 98 – DAta Systems IN Aerospace*, Athens, Greece. ESA Publications, SP-422, 25–28 May 1998. 18, 21

[99] K.G. Larsen and A. Skou, editors. *Proceedings of the Third International Workshop on Computer Aided Verification, CAV '91*, Aalborg, Denmark, Lecture Notes in Computer Science 575. Springer-Verlag, Berlin, Germany, 1–4 July 1991, 1992. 18, 21

[100] B. Le Charlier, editor. *Proceedings of the First International Symposium on Static Analysis, SAS '94*, Namur, Belgium, Lecture Notes in Computer Science 864. Springer-Verlag, Berlin, Germany, 28–30 september 1994. 18

[101] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis. In K. Furukawa, editor, *Proceedings of the International Conference on Logic Programming*, Paris, France, pages 64–78. MIT Press, Cambridge, Massachusetts, United States, 24–28 June 1991. 4

[102] 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, pages 137–146. IEEE Computer Society Press, Los Alamitos, California, United States, 20–23 April 1992. 4

[103] G. Levi, editor. *Proceedings of the Fifth International Symposium on Static Analysis, SAS '98*, Pisa, Italy, Lecture Notes in Computer Science 1503. Springer-Verlag, Berlin, Germany, 14–16 september 1998. 18

[104] 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*, 6(1), 1995. 4, 6

[105] T. Margaria and B. Steffen, editors. *Proceedings of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '96*, Passau, Germany, Lecture Notes in Computer Science 1055. Springer-Verlag, Berlin, Germany, 27–29 March 1996. 18, 21

[106] L. Mauborgne. Abstract interpretation using TDGs. In B. Le Charlier, editor, *Proceedings of the First International Symposium on Static Analysis, SAS '94*, Namur, Belgium, 20–22 september 1994, Lecture Notes in Computer Science 864, pages 363–379. Springer-Verlag, Berlin, Germany, 1994. 22

[107] L. Mauborgne. Abstract interpretation using typed decision graphs. *Science of Computer Programming*, 31(1):91–112, May 1998. 22

[108] L. Mauborgne. Binary decision graphs. In A. Cortesi and G. Filé, editors, *Proceedings of the Sixth International Symposium on Static Analysis, SAS '99*, Venice, Italy, 22–24 september 1999, Lecture Notes in Computer Science 1694, pages 101–116. Springer-Verlag, Berlin, Germany, 1999. 22

[109] L. Mauborgne. *Représentation d'ensembles d'arbres pour l'interprétation abstraite*. Thèse de l'école polytechnique en informatique, École polytechnique, Palaiseau, France, 25 November 1999. 22

[110] M. Müller-Olm, D.A. Schmidt, and B. Steffen. Model checking: a tutorial introduction. In A. Cortesi and G. Filé, editors, *Proceedings of the Sixth International Symposium on Static Analysis, SAS '99*, Venice, Italy, 22–24 september 1999, Lecture Notes in Computer Science 1694, pages 330–354. Springer-Verlag, Berlin, Germany, 1999. 4

[111] A. Mycroft, editor. *Proceedings of the Second International Symposium on Static Analysis, SAS '95*, Glasgow, United Kindom, Lecture Notes in Computer Science 983. Springer-Verlag, Berlin, Germany, 25–27 september 1995. 18

[112] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Proceedings of the Eight International Conference on Computer Aided Verification, CAV '96*, New Brunswick, New Jersey, United States, Lecture Notes in Computer Science 1102, pages 411–414. Springer-Verlag, Berlin, Germany, 31 July –3 August 1996. 1, 6

[113] S. Owre, J. Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Third International Workshop , TACAS '97*, number 1217 in Lecture Notes in Computer Science, pages 366–383, Enschede, Netherlands, 2–4 April 1997. Springer-Verlag, Berlin, Germany. 1, 6

[114] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995. 1

[115] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Proceedings of the Eleventh International Conference on Automated Deduction, CADE '92*, Saratoga Springs, New York, United States, Lecture Notes in Computer Science 607, pages 748–752. Springer-Verlag, Berlin, Germany, 15–18 June 1992. 1

[116] S. Owre, N. Shankar, and D.W.J. Stringer-Calvert. PVS: An experience report. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *PROC Applied Formal Methods - FM-Trends'98, International Workshop on Current Trends in Applied Formal Method*, Boppard, Germany, Lecture Notes in Computer Science 1641, pages 338–345. Springer-Verlag, Berlin, Germany, 7–9 October 1999. 1

[117] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, september 1981. 2

[118] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In R. Alur and T.A. Henzinger, editors, *Proceedings of the Eight International Conference on Computer Aided Verification, CAV '96*, New Brunswick, New Jersey, United States, Lecture Notes in Computer Science 1102, pages 184–195. Springer-Verlag, Berlin, Germany, 31 July –3 August 1996. 6

[119] D.K. Probst and G.V. Bochmann, editors. *Proceedings of the Fourth International Workshop on Computer Aided Verification, CAV '92*, Montreal, Canada, Lecture Notes in Computer Science 663. Springer-Verlag, Berlin, Germany, 29 June – 1 July 1992. 18, 21

[120] 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.*, pages 75–84. University of Aarhus, Dept. of Computer Science, January 1999. 4

[121] 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, pages 337–351. Springer-Verlag, Berlin, Germany, 1982. 3, 15, 21

[122] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking and automated proof checking. In P. Wolper, editor, *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95*, Liège, Belgium, Lecture Notes in Computer Science 939, pages 84–97. SPRINGER, 3–5 July 1995. 1, 6

[123] J.M. Rushby. Automated deduction and formal methods. In R. Alur and T.A. Henzinger, editors, *Proceedings of the Eight International Conference on Computer Aided Verification, CAV '96*, New Brunswick, New Jersey, United States, Lecture Notes in Computer Science 1102, pages 169–183. Springer-Verlag, Berlin, Germany, 31 July –3 August 1996. 6

[124] M. Sagiv, T. Reps, and R. Wilhelm. Shape analysis. In *Proceedings of the International Conference on Compiler Construction, CC '00*, 2000. To appear. 22

[125] H. Saï di and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Proceedings of the Eleventh International Conference on Computer Aided Verification, CAV '99*, Trento, Italy, Lecture Notes in Computer Science 1633, pages 443–454. Springer-Verlag, Berlin, Germany, 6–10 July 1999. 1, 6

[126] D.A. Schmidt and B. Steffen. Program analysis *as* model checking of abstract interpretations. In G. Levi, editor, *Proceedings of the Fifth International Symposium on Static Analysis, SAS '98*, Pisa, Italy, 14–16 september 1998, Lecture Notes in Computer Science 1503, pages 351–380. Springer-Verlag, Berlin, Germany, 1998. 4

[127] N. Shankar. Mechanical verification of real-time systems using PVS. In C. Courcoubetis, editor, *Proceedings of the Fifth International Conference on Computer Aided Verification, CAV '93*, Elounda, Greece, Lecture Notes in Computer Science 697, pages 280–291. Springer-Verlag, Berlin, Germany, 28 June –1 July 1993. 1

[128] N. Shankar. PVS: Combining specification, proof checking, and model checking. In M.S. Srivas and A.J. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design, FMCAD '96*, number 1166 in LNCS, pages 257–264, Palo Alto, California, United States, 6–8 November 1996. Springer-Verlag, Berlin, Germany. 1, 6

[129] B. Steffen, editor. *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '98*, Lisbon, Portugal, Lecture Notes in Computer Science 1384. Springer-Verlag, Berlin, Germany, 28 March – 4 April 1998. 18, 21

[130] B. Steffen, editor. *Ten Years of Partial Order Reduction*, Lisbon, Portugal, Lecture Notes in Computer Science 1384. Springer-Verlag, Berlin, Germany, 28 March – 4 April 1998. 19

[131] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955. 7, 11

[132] R. Vallée-Rai, H. Hendren, P. Lam, É Gagnon, and P. Co. Soot - a Java$^{\text{tm}}$ optimization framework. In *CASCON '99*, september 1999. 18

[133] P. van Hentenryck, editor. *Proceedings of the Fourth International Symposium on Static Analysis, SAS '97*, Paris, France, Lecture Notes in Computer Science 1302. Springer-Verlag, Berlin, Germany, 8–10 september 1998. 18

[134] A. Venet. Abstract interpretation of the π-calculus. In M. Dam, editor, *Analysis and Verification of Multiple-Agent Languages, LOMAPS Workshop*, Stockhlom, Sweden, 24–26 June 1996, Lecture Notes in Computer Science 1192, pages 51–75. Springer-Verlag, Berlin, Germany, 1996. 19

[135] A. Venet. Automatic determination of communication topologies in mobile systems. In G. Levi, editor, *Proceedings of the Fifth International Symposium on Static Analysis, SAS '98*, Pisa, Italy, 14–16 september 1998, Lecture Notes in Computer Science 1503, pages 152–167. Springer-Verlag, Berlin, Germany, 1998. 19

[136] P. Wolper, editor. *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95*, Liège, Belgium, Lecture Notes in Computer Science 939. Springer-Verlag, Berlin, Germany, 3–5 July 1995. 18, 21