Data Flow Analysis is Model Checking of Abstract Interpretations

David A. Schmidt
Computing and Information Sciences Department
Kansas State University[0]

Abstract

This expository paper simplifies and clarifies Steffen's depiction of data flow analysis (*d.f.a.*) as model checking: By employing abstract interpretation (*a.i.*) to generate program traces and by utilizing Kozen's modal mu-calculus to express trace properties, we express in simplest possible terms that a *d.f.a.* is a model check of a program's *a.i.* trace. In particular, the classic flow equations for bit-vector-based *d.f.a.*s reformat trivially into modal mu-calculus formulas. A surprising consequence is that two of the classical *d.f.a.*s are exposed as *unsound*; this problem is analyzed and simply repaired. In the process of making the above discoveries, we clarify the relationship between *a.i.* and *d.f.a.* in terms of the often-misunderstood notion of *collecting semantics* and we highlight how the research areas of flow analysis, abstract interpretation, and model checking have grown together.

1 Introduction

Folklore tells us that abstract interpretation (*a.i.*) is the "theory" of static analysis and data flow analysis (*d.f.a.*) is the "practice." This isn't quite so, and this paper clarifies the relation between *a.i.* and *d.f.a.* in terms of a third concept, model checking.

Utilizing Steffen's crucial observation that one can define a flow analysis by a modal logic formula [59, 60], we demonstrate in simplest possible terms that *an iterative d.f.a. is a model check of a modal logic formula on a program's a.i.* Here is one famous example: The bit-vector flow equation that defines the set of very-busy expressions at a program point, p,

$$VBE(p) = \\ Used(p) \cup (notModified(p) \cap (\bigcap_{p' \in succ\ p} VBE(p')))$$

is mechanically reformatted into a proposition in the modal mu-calculus [37] that states the very-business of an expression, e, at a program point:

$$isVBE(e) = \nu Z.\ isUsed(e) \vee (\neg isModified(e) \wedge \Box Z)$$

When the latter is model checked on a program's control-flow graph (which is the simplest possible safe *a.i.* of the program), one obtains exactly the information calculated by the iterative *d.f.a.* The other classic *d.f.a.*s decode similarly, and two of them (live variables analysis and reaching definitions) are found to be *unsound.* This surprising development is explained and repaired in a simple way.

In the process of developing the above results, we observe that *(i)* model checking can be applied to *any a.i.*, not just a control-flow graph; and *(ii)* an iterative *d.f.a.* can be defined by any modal-mu calculus formula, not just a bit-vector-based one. These facts become obvious because we use *trace-based abstract interpretation* [55, 53], an operational-semantics-based version of abstract intepretation that represents a program's *a.i.* as a computation tree of traces [44]. (As noted above, a program's simplest *a.i.*-based computation tree is its control flow graph; of course, there are many more interesting abstract computation trees.) Using trace-based *a.i.*s, we can explain simply the crucial and often-misunderstood concept of *collecting semantics*; we note that there exist multiple versions of collecting semantics for a given *a.i.*, and we employ modal mu-calculus formulas to extract particular instances of collecting semantics. From here, it is easy to see how iterative *d.f.a.*s compute representations of collecting semantics defined by modal mu-calculus formulas.

The developments in this paper should make clear that the classic *d.f.a.*s merely scratch the surface as to the forms of iterative static analysis one can perform—the combination of the abstract interpretation design space and the modal mu-calculus defines the design

[0] 234 Nichols Hall, Manhattan, KS 66506 USA. schmidt@cis.ksu.edu. Supported by NSF CCR-9633388.

space for iterative static analyses. Also apparent is the growing intersection of terminology and techniques from *d.f.a.*, *a.i.*, and model checking.

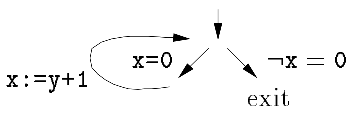
2 Related Research

The bodies of research literature on data flow analysis [3, 4, 30, 46], abstract interpretation [1, 16, 19, 21, 29, 32], and model checking [7, 10, 12, 26, 42] are huge. Appearing less frequently are papers that demonstrate the interaction of these areas: The groundbreaking papers of Cousot and Cousot [15, 16, 17, 18] provided an explanation of how one computes a *d.f.a.* with a least-fixed point calculation on a flowchart interpreted over sets of abstract values [16].

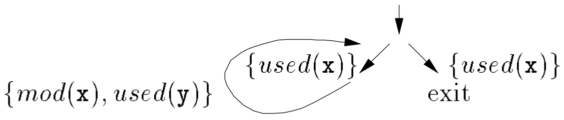
Donzeau-Gouge [25] and Nielson [47, 48, 49, 50] build upon Cousot and Cousot's work by using denotational-semantics-style definitions as the calculational engine for flow analysis. Nielson [47, 48] states proofs why such definitions calculate correct flow analyses, but such proofs are less frequent in the subsequent literature than one would hope.

Because of the state-space explosion problem in model checking, several researchers have explored abstract interpretations of large finite-state transition systems: Clark, Grumberg, and Long utilize abstractions based on modulo-n arithmetic to model check hardware problems [11], and Bensalem, et al. [6] state safety conditions under which such abstractions can be employed. Dams applies these results in a more general theory of abstract interpretation of reactive systems [22, 23, 24], and Bruns employs an *a.i.*-like abstraction on the labels of a labelled transition system [8]. Levi gives a formalization of this and more general abstractions [41]. Cleaveland, Iyer, and Yankelevich abstract upon a "democratic" transition system (cf. Larsen [40]). The latter is also an example of a *reduction* of a state transition system, where states (or transitions) are collapsed without loss of the system's properties [38].

The crucial connection between model checking and data-flow analysis was made by Steffen [60, 61], who encoded flow analyses in recursive Hennessy-Milner logic [7, 39, 63] (which is usually called "modal mu-calculus" as well) and model checked propositions on *DFA-models* of programs. For example, the statement, `while x=0 do x:= y+1`, is depicted as a graph where program phrases label the arcs:



Next, the phrases on the arcs are "abstracted" into *gen-* and *kill*-style sets, producing a DFA-model:



Finally, this model is checked with a formula in Hennessy-Milner logic. A proposition that states that variable **x** is live at a node in the DFA-model goes as follows [61]:

$$isLive(x) = \mu Z.\ \langle\{used(\mathtt{x})\}\rangle tt \vee \langle -\{mod(\mathtt{x})\}\rangle Z$$

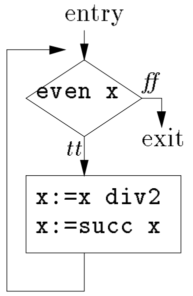
(Read $\langle\alpha\rangle\phi$ as "there exists a transition labelled by a member of α to a next state such that ϕ holds." As we see later, μZ creates recursive propositions; read tt as "true" and $-S$ as the complement of set S.) The model checker validates or refutes the proposition at all the nodes of the DFA-model; in doing so, it calculates live-variables analysis.

The current paper should be viewed as an exposition, a simplification, and a clarification of Steffen's original proposal: We use trace-based abstract interpretations, rather than DFA-models, and we use Kozen's mu-calculus, rather than recursive Hennessy-Milner logic. This gives a simple starting point from which intuitive formulations of the classic *d.f.a.*s arise simply. Extensions, such as labelling the transitions in a program's computation tree (as seen in the DFA-models) and then labelling the mu-calculus modalities, can be added to the framework, extending the range of application.

3 Trace-Based Abstract Interpretation

For brevity, we focus upon flowchart programs; more complex programs are discussed in the paper's conclusion. Figure 1 shows a flowchart program, its concrete semantics (*concrete interpretation—c.i.* for short), expressed as a set of state transition rules, and a sample *concrete computation tree* (*concrete tree*) of the program with the input of 4. The concrete computation tree is an execution trace, where program states appear as nodes. In the example, a program state is an element of $Val \times ProgramPoint$ and is written as $n \vdash p$. (For simplicity, we let the statements in the example program be exactly the program points.) Of course, n is the value of variable **x**.

The fundamental concept of trace-based abstract interpretation is that an abstract interpretation (*a.i.*) is an execution of the program by transition rules that use property "tokens" instead of run-time values. The result is an *abstract computation tree* (*abstract tree*), where multiple possible execution traces are presented by nondeterministic branching in the abstract tree.



Concrete semantics transitions:
$Val = Nat$
$2n \vdash \texttt{even x} \rightarrow 2n \vdash \texttt{x := x div2}$
$2n+1 \vdash \texttt{even x} \rightarrow 2n+1 \vdash exit$
$2n \vdash \texttt{x := x div2} \rightarrow n \vdash \texttt{x := succ x}$
$2n+1 \vdash \texttt{x := x div2} \rightarrow n \vdash \texttt{x := succ x}$
$n \vdash \texttt{x := succ x} \rightarrow n+1 \vdash \texttt{even x}$

Concrete computation tree:
$4 \vdash \texttt{even x}$
$4 \vdash \texttt{x := x div2}$
$2 \vdash \texttt{x := succ x}$
$3 \vdash \texttt{even x}$
$3 \vdash \text{exit}$

Figure 1: Flowchart and concrete interpretation

For the example in Figure 1, one defines an even-odd analysis by by replacing the domain definition $Val = Nat$ by $AbsVal = \{e, o\}$ (e for "even", o for "odd")[1] and rewriting the state transition rules accordingly, by applying the obvious homomorphism criterion [11, 16, 53, 58]. (To be more specific, one can define a function, $\beta : Val \rightarrow AbsVal$, which maps concrete values to their most precise abstractions. Here, $\beta(2n) = e$ and $\beta(2n+1) = o$. The homomorphism property is: if $(v \vdash p) \rightarrow (v' \vdash p')$ is a concrete transition, then there must exist an abstract transition, $(\beta(v) \vdash p) \rightarrow (a' \vdash p')$, such that $\beta(v') \sqsubseteq a'$. *Note:* $\beta(v') = a'$ is preferred, but the former is safe, nonetheless.[2])

Figure 2 shows the abstract semantics and an abstract tree for the program in Figure 1. Since the even-odd properties lose precision, nondeterminism appears as branching in the abstract computation tree. Also, the tree is infinite, but in this particular case, the tree is *regular tree* [14], because a node repeats in every infinite path. *Widening* [16] or *memoization* [53] can force an abstract computation tree to be regular; we ignore this topic here because a trace-based *a.i.* with a finite-cardinality *AbsVal* set must generate regular trees. Note that a regular tree is of course a finite-state transition system.

The abstract computation trees generated by our traced-based abstract interpretation are "maximally polyvariant" [5] or "maximally relational" [33] analyses in the sense that distinct states are never joined. An implemention of such an *a.i.* is often more monovariant—e.g., states with the same program points are combined—nonetheless, the trace-based *a.i.* gives the correct starting point for refinements, implementations, and correctness proofs.

To be useful, a program's abstract computation tree must safely simulate the concrete computation tree that it represents. Of the many ways of stating this, we use the criterion from concurrency theory [44]. Let t be a computation tree, let $root(t)$ be its root node, and let t_i, for $i \in 0..n$, be t's immediate subtrees. We write $t \longrightarrow t_i$ to denote that there exists a state transition, $root(t) \rightarrow root(t_i)$. Read "$t \longrightarrow t_i$" as saying, "$t$ makes a transition and becomes t_i." A program's concrete computation tree, t_C, is *simulated* by an abstract tree, t_A, iff the binary relation, $t_C\ safe_{Tree}\ t_A$, holds true:

$$\begin{array}{l} t\ safe_{Tree}\ t' \text{ iff } root(t)\ safe_{State}\ root(t'), \\ \quad \text{and, for every transition, } t \longrightarrow t_i, \\ \quad \text{there exists a transition, } t' \longrightarrow t'_j, \\ \quad \text{such that } t_i\ safe_{Tree}\ t'_j \end{array}$$

For the example in the Figures, we define $(n \vdash p)\ safe_{State}\ (a \vdash p)$ iff $n\ safe_{Val}\ a$, and we define $2n\ safe_{Val}\ e$ and $2n+1\ safe_{Val}\ o$.[3] The intent of $safe_{Tree}$ is that every computation path in t_C is mirrored by one in t_A—the abstract tree contains transitions that *may* happen.

A technical issue is that the definition of $safe_{Tree} \subseteq ConcreteTree \times AbstractTree$ is recursive, and the largest such relation satisfying the recursion is desired [2, 20, 45, 54].

Of course, the proof of safe simulation can be performed directly upon the concrete and abstract semantics rules rather than upon specific pairs of computation trees: Indeed, the homormorphism property stated

[1] It is traditional to partially order the elements of *AbsVal*, typically because a join operation is needed to force termination of the analysis. In the example, we can use a discrete partial ordering upon *AbsVal*, because it has finite cardinality—termination is assured. As a general rule, finer partial orderings on *AbsVal* lend themselves to more precise analyses.

[2] If *AbsVal* is nondiscretely partially ordered, a monotonicity property must be enforced: if $(a_1 \vdash p) \rightarrow (a_2 \vdash p')$ is an abstract transition, and $a_1 \sqsubseteq a'_1$, then there must exist a transition $(a'_1 \vdash p) \rightarrow (a'_2 \vdash p')$, such that $a_2 \sqsubseteq a'_2$.

[3] If one begins with the function, $\beta : Val \rightarrow AbsVal$, then the proper definition for $safe_{Val}$ is $c\ safe_{Val}\ a$ iff $\beta(c) \sqsubseteq a$.

Abstract semantics transitions:
$AbsVal = \{e, o\}$

$e \vdash \texttt{even x} \rightarrow e \vdash \texttt{x := x div2}$
$o \vdash \texttt{even x} \rightarrow o \vdash exit$
$v \vdash \texttt{x := x div2} \rightarrow e \vdash \texttt{x := succ x}$
$v \vdash \texttt{x := x div2} \rightarrow o \vdash \texttt{x := succ x}$,
for all $v \in AbsVal$
$e \vdash \texttt{x := succ x} \rightarrow o \vdash \texttt{even x}$
$o \vdash \texttt{x := succ x} \rightarrow e \vdash \texttt{even x}$

Abstract computation tree:

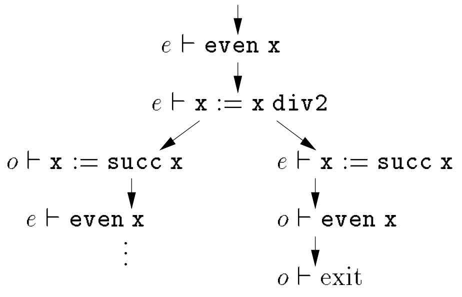


Figure 2: Abstract interpretation of flowchart

earlier implies the safety result [53] for pairs of corresponding trees generated by the concrete and abstract semantics.

The above relates to the traditional, Galois connection framework [18, 19, 43, 47, 54][4] in the following way: If $AbsVal$ is a complete lattice and $safe_{Val}$ is both *U-closed* ($c\ safe_{Val}\ a_1$ and $a_1 \sqsubseteq a_2$ imply $c\ safe_{Val}\ a_2$) and *G-closed* ($c'\ safe_{Val}\ \sqcap A$, where $A = \{a' \mid c'\ safe_{Val}\ a'\}$), then one obtains the Galois connection, $(\alpha: \mathcal{P}(Val) \rightarrow AbsVal,\ \gamma: AbsVal \rightarrow \mathcal{P}(Val))$ by defining $\gamma(a) = \{c \mid c\ safe_{Val}\ a\}$ and $\alpha(S) = \sqcup_{c \in S}\{\sqcap\{a \mid c\ safe_{Val}\ a\}\}$.[5]

There is a dual to safe simulation, called *live simulation*, where a program's abstract tree is simulated by its concrete tree, that is, the abstract tree contains only those transitions that *must* happen in the concrete tree. (Such transitions are sometimes called *conservative transitions* [13].) Live simulations are useful when liveness properties must be proved from an *a.i.* Since classical data-flow analysis concerns itself with safe simulations, we study only safe simulations in this paper.

4 Collecting Semantics

An *a.i.* is not a *d.f.a.*, but the *a.i.*'s *collecting semantics* turns out to be the information calculated by a *d.f.a.* A collecting semantics is information extracted from the nodes and paths of a computation tree. The classic, "first-order" [48] collecting semantics extracts the states from a computation tree: For tree, t, its first-order collecting semantics has form $coll_t : ProgramPoint \rightarrow \mathcal{P}(Val)$ and is defined

$$coll_t(p) = \{v \mid v \vdash p \text{ is a state in } t\}$$

In Figure 1, $coll_{t_C}(\texttt{even x}) = \{3, 4\}$, and in Figure 2, $coll_{t_A}(\texttt{even x}) = \{e, o\}$—one can extract a collecting semantics from concrete as well as abstract trees.

Constant-propagation and type-inference analyses calculate answers that are first-order collecting semantics.

A more interesting collecting semantics is "second order" or path based: It extracts paths from the computation tree. The set of paths that go into a program point, p, is defined

$$fcoll_t(p) = \{r \mid r \text{ is a path in } t \text{ from } root(t) \text{ to some } v \vdash p\}$$

and the set of paths that emanate from p is

$$bcoll_t(p) = \{r \mid r \text{ is a maximal path in } t \text{ such that } root(r) = v \vdash p\}$$

We will see that several classic *d.f.a.*s compute representations of path-based collecting semantics. The two collecting semantics are named *fcoll* and *bcoll* because they underlie the information one obtains from forwards and backwards iterative flow analyses, respectively.

The above forms of collecting semantics are "primitive" in the sense that no judgement about the extracted information is made. In practice, the information one desires from a data flow analysis is a judgement whether some property holds true of the input values to a program point or of the paths flowing into/out of a program point.

To include such judgements, Cousot and Cousot [19] suggest that a computation tree's collecting semantics

[4] Recall that a Galois connection is a pair of monotone functions, $(f: P \rightarrow Q, g: Q \rightarrow P)$, for complete lattices P and Q, such that $f \circ g \sqsubseteq id_Q$ and $id_P \sqsubseteq g \circ f$. The intuition is that $f(p)$ identifies p's most precise representative within Q (and similarly for $g(q)$).

[5] If one begins with $\beta : Val \rightarrow AbsVal$, then $\beta(c) = \sqcap\{a \mid c\ safe_{Val}\ a\}$ and $\alpha(S) = \sqcup\{\beta(c) \mid c \in S\}$.

can be a set of properties expressed in a logic, $\mathcal{L}$. Given tree, t, and proposition, $\phi \in \mathcal{L}$, we write $t \models \phi$ if ϕ holds true of t. Next, we define the collecting semantics of the entire tree, t, to be $coll_t = \{\phi \mid t \models \phi\}$. As before, collecting semantics exist for both concrete and abstract computation trees, and we assume for simplicity that the same $\mathcal{L}$ can be used with both concrete and abstract trees. (But it need not be—see [8, 41].)

For an *a.i.* to be of use, we require a *weak consistency* property of the safety relation, $safe_{Tree}$, and $\mathcal{L}$:

$$t_C \; safe_{Tree} \; t_A \Rightarrow (\text{ for all } \phi \in \mathcal{L}, t_A \models \phi \Rightarrow t_C \models \phi)$$

That is, any property possessed by an abstract tree, t_A, must also hold for a corresponding concrete tree, t_C. By tightening the two implications in the above formula into logical equivalences, we obtain weak completeness and strong completeness, respectively. The former is studied in [19]; the latter is employed to justify correctness of reductions of state spaces in concurrency theory [13, 22, 38].

5 Defining Collecting Semantics with the Modal Mu-Calculus

The logic we use in this paper to define a collecting semantics is the modal mu-calculus [37] extended with reverse modalities [59, 60]. (The latter, promoted by Steffen, lets us express properties about paths that flow into a state.) Figure 3 gives the syntax and semantics we use. A judgement takes the form $s \models_t \phi$, where t is a tree, s is a state in t, and ϕ is a proposition about state s. (Note the slight difference in notation from $t \models \phi$ in the previous section.[6]) Primitive properties, q, are "first-order" properties about state, e.g., "variable $\mathtt{x}$'s value at state s is positive" (written $s \models_t (\mathtt{x} > 0)$). We assume that t contains states that are detailed enough that $s \models_t q$ can be decided. The modalities are the usual ones and are used to define "second-order" properties, e.g., "there exists a path starting from s such that variable $\mathtt{x}$ has a positive value in two transitions" ($s \models_t \Diamond\Diamond(\mathtt{x} > 0)$), and "all immediate predecessors to s have $\mathtt{x}$ with a positive value" ($s \models_t \overline{\Box} (\mathtt{x} > 0)$). Finally, the least fixed-point operator, μ, defines properties that hold true at some point finitely far into the future (or past), e.g., "at some state now or in the past, x was positive" ($s \models_t \mu Z.(\mathtt{x} > 0) \vee \overline{\Diamond} Z$), and the greatest fixed-point operator, ν, defines properties that hold true indefinitely, even infinitely, e.g., "from now on, x is always positive" ($s \models_t \nu Z.(\mathtt{x} > 0) \wedge \Box Z$).

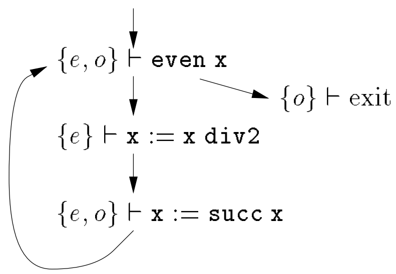
The semantics of μ and ν in Figure 3 are simpler than usual because we work with trees that have a finite number of distinct states. For an infinite-state tree with finite branching, we must employ the usual definitions: Let $[\![\phi]\!] \in Env \to \mathcal{P}(StateInTree\_t)$ and $\rho \in Env = Identifier \to \mathcal{P}(StateInTree\_t)$:

$$[\![q]\!]\rho = \{s \mid q \text{ holds at } s\}$$
$$\ldots$$
$$[\![\Box\phi]\!]\rho = \{s \mid \text{ for all } s \to s', s' \in [\![\phi]\!]\rho\}$$
$$\ldots$$
$$[\![\mu Z.\phi]\!]\rho = \bigcup_{i \geq 0} S_i, \text{ where } \begin{cases} S_0 = \emptyset \\ S_{i+1} = [\![\phi]\!]([Z \mapsto S_i]\rho) \end{cases}$$
$$[\![\nu Z.\phi]\!]\rho = \bigcap_{i \geq 0} S_i, \text{ where } \begin{cases} S_0 = State \\ S_{i+1} = [\![\phi]\!]([Z \mapsto S_i]\rho) \end{cases}$$
$$[\![Z]\!]\rho = \rho(Z)$$

When the computation tree is finite-state, the above simplifies to the definitions in the Figure.

The usual application of the modal mu-calculus is to *model checking*, that is, the mechanical verification of a judgement, $s \models_t \phi$. Here, we use model checking to compute a collecting semantics for an abstract tree.

Here are two small examples based on Figure 2. Say that improved code can be generated for a statement when its input is an even number. Therefore, we desire a collecting semantics that tells us which statements receive only even-valued inputs. The property to be model checked is first order and trivial, namely, $isEven(\mathtt{x})$. If a model check upon the tree, t_A, in Figure 2 decides that $(a \vdash p_0) \models_{t_A} isEven(\mathtt{x})$ holds true for every occurrence of a appearing with program point p_0 in t_A, then p_0's code can be improved. (Note: the story gets even simpler if we reformat the tree in Figure 2 with its program points merged, that is, we draw



before we perform the model check. This graph defines the "meet over all paths" solution [34, 48].) The model check computes that the statement `x:= x div2` has the desired property.

[6] The modification of the judgements, $t \models \phi$, into $s \models_t \phi$, is due to the awkwardness in defining precisely $t \models \overline{\Box}\, \phi$, which requires knowledge of all trees, t', that contain t as a child subtree so that one can verify $t' \models \phi$. This implies that the first argument, t, of $t \models \phi$ must be restricted to range over exactly all subtrees of some initial computation tree, t_0. But it is traditional to use the states, s, within t_0 instead of the subtrees, thus giving us the notation, $s \models_t \phi$. Indeed, propositions ϕ are commonly called "state formulas," anyway!

$$s \in \mathit{StateInTree\_t} \qquad \phi \in \mathit{Proposition} \qquad q \in \mathit{PrimitiveProposition} \qquad Z \in \mathit{Identifier}$$

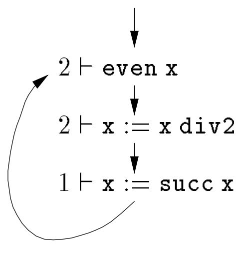
$$\phi \ ::= \ q \ \mid \ \neg q \ \mid \ \phi_1 \wedge \phi_2 \ \mid \ \phi_1 \vee \phi_2 \ \mid \ \Box\phi \ \mid \ \overline{\Box}\,\phi \ \mid \ \Diamond\phi \ \mid \ \overline{\Diamond}\,\phi \ \mid \ \mu Z.\phi \ \mid \ \nu Z.\phi \ \mid \ Z$$

$s \models_t q$ is given
$s \not\models_t q$ iff not $s \models_t q$
$s \models_t \phi_1 \wedge \phi_2$ iff $s \models_t \phi_1$ and $s \models_t \phi_2$
$s \models_t \phi_1 \vee \phi_2$ iff $s \models_t \phi_1$ or $s \models_t \phi_2$
$s \models_t \Box\phi$ iff for all s' such that $s \rightarrow s', s' \models_t \phi$
$s \models_t \overline{\Box}\,\phi$ iff for all s' such that $s' \rightarrow s, s' \models_t \phi$
$s \models_t \Diamond\phi$ iff there exists s' such that $s \rightarrow s'$ and $s' \models_t \phi$
$s \models_t \overline{\Diamond}\,\phi$ iff there exists s' such that $s' \rightarrow s$ and $s' \models_t \phi$
$s \models_t \mu Z.\phi$ iff there exists $i \geq 0$ such that $s \models_t \phi_i$, where $\begin{cases} \phi_0 = \mathit{false} \\ \phi_{i+1} = [\phi_i/Z]\phi \end{cases}$
$s \models_t \nu Z.\phi$ iff for all $i \geq 0, s \models_t \phi_i$, where $\begin{cases} \phi_0 = \mathit{true} \\ \phi_{i+1} = [\phi_i/Z]\phi \end{cases}$

Figure 3: Mu-calculus syntax and semantics

For a second example, say that we wish to deallocate variable cells from the storage vector when they are no longer referenced in the future. The property that variable `x` is no longer referenced in future states is encoded $\nu Z.\ \neg \mathit{Used}(\mathtt{x}) \wedge \Box Z$. This is a second-order property and requires a nontrivial model check. For the tree in Figure 2, the analysis discovers that the property holds only at the state, $o \vdash$ exit.

Recall that we require weak consistency of the modal mu-calculus: A property that holds true for a state of the abstract tree must also hold true for the corresponding state of the corresponding concrete tree. Unfortunately, the $\Diamond$ and $\overline{\Diamond}$ modalities are problematic for safe simulations: Consider again the abstract tree in Figure 2; it is a safe simulation of every concrete interpretation that executes with an even-numbered input. In particular, it is a safe simulation of the program's concrete tree with input 2:

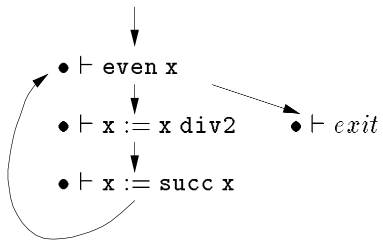


because every transition in the above concrete tree is mirrored by a transition in the abstract tree. Now, consider a model check of the property that the program "may terminate": $\mu Z.\ \mathit{isExit} \vee \Diamond Z$. This property holds true of the root state of the tree in Figure 2 because there does indeed exist a path that leads to an exit statement. But the same property fails to hold for the corresponding root state of the concrete tree just seen above. The upshot is that the diamond modalities violate the weak consistency property. Fortunately, weak consistency—indeed, weak completeness—holds for the *box mu-calculus*, that is, the modal mu-calculus with only box-modalities [6]. If we negate the above proposition: $\neg(\mu Z.\ \mathit{isExit} \vee \Diamond Z) = \nu Z.\ \neg \mathit{isExit} \wedge \Box Z$, we obtain the "must loop" property, which is appropriate to model check on the abstract tree. Of course, this property cannot be validated on (the root of) the tree, hence we cannot conclude that all corresponding concrete trees must loop. (Indeed, almost all of them do not.)

For the record, a dual result holds: The diamond mu-calculus is weakly complete for live simulations [6].

6 Why a Data-Flow Analysis is a Model Check

An iterative data-flow analysis (*d.f.a.*) operates on a program's control-flow graph. But a control-flow graph is a program's simplest possible safe simulation: In the case of Figure 1, define *AbsVal* to be just $\{\bullet\}$, and translate each concrete state-transition rule, $v \vdash p \rightarrow v' \vdash p'$, into the abstract state-transition rule, $\bullet \vdash p \rightarrow \bullet \vdash p'$. With this abstract semantics, the program's abstract computation tree is exactly its control-flow graph, e.g.:



Of course, more interesting *a.i.* s can be used to perform a *d.f.a.*—even-odd analysis is but one simple example—but the classical *d.f.a.* s use control flow graphs.

A usual iterative *d.f.a.* is defined by a set of flow equations, one equation per program point (that is, one equation per state in the control-flow graph). A bit-vector-based *d.f.a.* uses flow equations with set-theoretic operations; a standard example is *very busy expressions* analysis, which calculates for each program point the set of expressions that must be referenced sometime in the future. The flow equation for program point, p, reads as follows [35]:

$$\begin{aligned} &\mathit{VBE}(p) = \\ &\quad \mathit{Used}(p) \cup (\mathit{notModified}(p) \cap (\textstyle\bigcap_{p' \in succ\ p} \mathit{VBE}(p'))) \end{aligned}$$

That is, the set of very-busy expressions at the entry to program point p consists of the expressions used (referenced) within statement p unioned with those expressions that are not modified within p by reassignment and are very busy at all of p's successor program points.

The set of simultaneous flow equations for the states of the control-flow graph are solved with a greatest fixed-point calculation (that is, the initial approximation to $\mathit{VBE}(p)$ is defined as the set of all expressions, and subsequent iterations cut the set down to size). But the obvious relationship between the set-theoretic operations in the above equation and the propositional connectives gives us this modal-mu calculus formula:

$$\mathit{isVBE}(e) = \nu Z.\ \mathit{isUsed}(e) \vee (\neg \mathit{isModified}(e) \wedge \Box Z)$$

That is, an expression, e, is very busy at a state if it is used at the state or it is not modified at the state and for all successor states it is very busy. A model check of $\mathit{isVBE}(e)$ for all the states of the control-flow graph yields the same information about e as does the iterative *d.f.a.* (This is proved by induction on the greatest-fixed point definition of the flow equations and the definition of the mu-calculus proposition.)

7 Why Some Flow Analyses are Unsound

Figure 4 shows four canonical iterative *d.f.a.* s and their encodings as modal mu-calculus formulas [3, 35, 29].

The encodings are straightforward: Simple union and intersection translate into disjunction and conjunction, respectively; "big" unions and intersections on the predecessor and successor states translate into diamond and box modalities, respectively; a forwards analysis uses overlined modalities (because it calculates information about histories) and a backwards analysis uses unlined modalities (because it calculates information about futures); and the least and greatest fixed-point solutions of the equations are stated explicitly by the μ- and ν-operators.

Since the definitions calculate information as it appears upon entry to a program point, an encoding of a forwards analysis uses its modality operator as its outermost operator, whereas a backwards analysis embeds the modality operator within its formula.

The four examples in the Figure are perhaps the most famous examples of flow analyses and are meant to portray the four combinations one achieves by varying overlined and unlined modalities (that is, forwards and backward flow) with least and greatest fixed points. (For example, available-expressions analysis uses overlined modality and greatest-fixed point.) Yet another variation is with diamond and box modalities (union- and intersection-based flow). (E.g., available-expressions analysis uses box modality.) The resulting eight combinations give the flow analysis "cube" of Cousot and Cousot [16], but in practice, diamond modalities appear only with least-fixed point operators and box modalities appear only with greatest-fixed point operators, because the initial approximations for least-fixed point equations are "false" (empty sets) and the initial approximations for greatest-fixed point equations are "true" (universe sets) (*cf.* Figure 3). But additional variations of the primitive propositions and propositional operators in the equations are obvious—examples appeared in the previous sections.

In Figure 4 lies a problem: Recall that only the box-mu-calculus is consistent with safe simulations (of which a control-flow graph is one), and the analyses for live variables and reaching definitions use diamond modalities, which are consistent with live simulations (of which the control-flow graph is *not*). This implies that the two analyses are *unsound*. The problem is not deep but it is nonetheless significant: Figure 5 displays a flowchart program that computes upon variables **x** and **y**. A concrete computation tree for the program with inputs of $(4, 4)$ also appears, and finally there is an abstract tree that results from an even-odd *a.i.* that

Available Expressions:

$$AE(p) = \bigcap_{p' \in pred\ p}((AE(p') \cap notModified(p')) \cup Gen(p'))$$

$$isAvail(e) = \nu Z.\ \overline{\Box}\ ((Z \wedge \neg isModified(e)) \vee isGen(e))$$

Live Variables:

$$LV(p) = Used(p) \cup (notModified(p) \cap (\bigcup_{p' \in succ\ p} LV(p')))$$

$$isLive(x) = \mu Z.\ isUsed(x) \vee (\neg isModified(x) \wedge (\Diamond Z))$$

Very Busy Expressions:

$$VBE(p) = Used(p) \cup (notModified(p) \cap (\bigcap_{p' \in succ\ p'} VBE(p'))$$

$$isVBE(e) = \nu Z.\ isUsed(e) \vee (\neg isModified(e) \wedge \Box Z)$$

Reaching Definitions:

$$RD(p) = \bigcup_{p' \in pred\ p}((RD(p') \cap notModified(p')) \cup Defined(p'))$$

$$isReaching(d) = \mu Z.\ \overline{\Diamond}\ ((Z \wedge \neg isModified(d)) \vee isDefined(d))$$

Figure 4: Four classic flow analyses

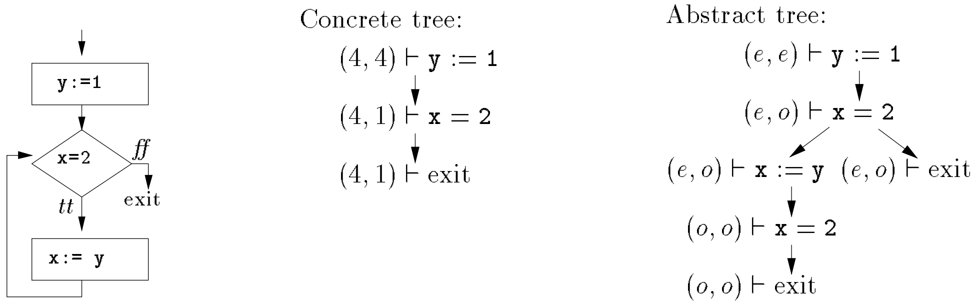


Figure 5: Live Variables Analysis is Unsound

is a safe simulation of the *c.i.* It is easy to verify of the abstract tree, at the test statement `x=2`, that `y` is live. But this property is not true at the test statement of the corresponding concrete tree. Therefore, any verification or code improvement based on the positive liveness of `y` at the test in the abstract tree is incorrect.

Of course, data-flow practitioners are well aware of the above problem, and disaster does not arise in practice, because live variables analysis is used "dually"—it is used to detect *dead variables.* That is,

$$\begin{aligned} isDead(x) &= \neg(isLive(x)) \\ &= \nu Z.\ \neg isUsed(x) \wedge (isModified(x) \vee \Box Z) \end{aligned}$$

which *is* proper to model check upon a safe simulation. We are fortunate that we can use a live variables analysis to detect dead variables; this works only because $s \models_t \neg\phi$ iff $s \not\models_t \phi$ (this is a classical logic), hence $s \models_t isDead(x)$ iff $s \not\models_t \neg(isLive(x))$ iff $s \not\models isLive(x)$. But we might not be so fortunate in general.

A similar story can be told for reaching-definitions analysis and other iterative *d.f.a.*s that implicitly encode diamond modalities. Obviously, errors might arise when complex iterative *d.f.a.*s are encoded with disregard to their modalities. For this reason, specifying a *d.f.a.* as a modal-mu calculus proposition provides a valuable safety check of the soundness of the *d.f.a.* Indeed, Steffen and his collegues at Passau have built tools that prototype *d.f.a.*s as formulas in recursive Hennessy-Milner logic [36, 62].

8 Extensions, Connections, and Conclusions

With simple machinery, we have exposed that the classic iterative flow analyses are model checks of a program's trace-based *a.i.* We also noted that some of the classic *d.f.a.*s are unsound, and we explained how a *d.f.a.* can be analyzed for soundness and repaired, if necessary.

Even if a *d.f.a.* is not automatically synthesized by generating an abstract computation tree and doing a model check, the methodology proves valuable for specification and validation of the *d.f.a.* algorithm that is in fact implemented.

There are a variety of extensions to the framework presented in this paper. First, one might decompose an execution state, $\sigma \vdash p$, into its components. If p is considered to be an "active" statement—a store-transfer function—rather than an "inactive" program point, then it makes good sense to label the transitions in a computation tree by the transfer functions, p, that transform stores, σ, into σ'; this results in transitions of the form, $\sigma \xrightarrow{p} \sigma'$, like those found in labelled transition systems [44] and suggests that the appropriate logic for model checking the resulting computation trees is recursive Hennessy-Milner logic [7]. Indeed, this is the framework Steffen used to present his original results [59, 60, 61].

In the present paper, we used unlabelled transition systems and Kozen's mu-calculus to provide a simplest possible starting point, to clarify the basic import of Steffen's results, to tighten the tie between model checking and the classic data-flow analyses, and to fill a small gap in the literature on the subject.

Although the present paper focussed solely on flowchart programs with simplistic transition semantics, one can apply trace-based abstract interpretation to derivations constructed with a big-step (*natural*) semantics and also with a Plotkin-style small-step structural operational semantics [52].

Trace-based abstract interpretation of big-step semantics generates derivation trees that derive abstract properties rather than run-time values; the novelty is the necessity for infinite derivations, which are disallowed under the usual inductive interpretation. Therefore, a coinductive interpretation of the big-step rules must be undertaken. The concepts of coinductive big-step semantics were laid down by Cousot and Cousot [20], applied by Schmidt [54] and refined in subsequent work [53]. In a related line of work, Gouranton and Le Métayer extract execution path information from big-step derivation trees and use it to derive standard *d.f.a.*s [28]; general frameworks for doing this are presented in Gouranton's thesis [27].

At this time, there is no precedent for analyzing directly big-step derivation trees by means of model checking, but Gouranton and Le Métayer's efforts [27, 28] imply that this should be a matter of routine formulation.

Trace-based *a.i.* of small-step semantics is developed in detail by Schmidt [55], where abstraction on the source language's syntax (in this case, a π-calculus variant) is needed to ensure construction of a finite-state abstract computation tree. Model checking the resulting trees proceeds exactly like the examples seen in the present paper, and indeed, the concurrency theory literature abounds with similar examples.

Of course, both big-step and small-step structural operational semantics can be used to express the semantics of higher-order languages, and an outstanding question is whether any of the varieties of control-flow (closure) analyses [31, 51, 56, 57] can be encoded elegantly as model checks upon appropriate abstract computation trees.

Finally, it is worthwhile to consider why flow analysis and model checking are so intimately related: The reason why a *d.f.a.* can be implemented by a model check is because the usual "engine" that implements model checking is just a fixed-point calculation algorithm [26]. Not surprisingly, the dual is achievable: The canonical model checking algorithm can be encoded as a set of flow equations [9]. Indeed, even the tableaus generated from a tableau-based model checker [12] can be encoded as abstract computation trees where the data part, a, of a state, $a \vdash p$, is abstracted to a mu-calculus proposition.

These connections suggest that the machinery and methods of flow analysis, abstract interpretation, and model checking are growing together. Researchers can profitably use techniques from one area to improve results in the others. Recent interest in model checking abstractions of finite- and infinite-state models is yet another indication of the growing overlap. These areas will thrive if they draw from one another to advance their respective causes.

9 Acknowledgements

Bernhard Steffen, Carolyn Talcott, and Mitchell Wand studied drafts of this and a related paper and made many useful suggestions. Also, Stephen Brookes, Edmund Clarke, Olivier Danvy, Peter Mosses, and Colin Stirling are thanked for hosting me during my sabbatical year journeys.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract interpretation of declarative languages*. Ellis Horwood, Chichester, 1987.

[2] P. Aczel. *Non-Well-Founded Sets*. Lecture Notes 14, Center for Study of Language and Information, Stanford, CA, 1988.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[4] A. Aho and J. Ullman. *Principles of Compiler Design*. Addison Wesley, 1977.

[5] A. Banerjee. A modular, polyvariant, type-based closure analysis. In *Proc. 2d International Conference on Functional Programming: ICFP'97*, 1997.

[6] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In G. vanBochman and D. Probst, editors, *Computer Aided Verification: CAV'92*, number 663 in Lecture Notes in Computer Science, pages 260–273. Springer-Verlag, 1992.

[7] J. Bradfield. *Verifying Temporal Properties of Systems*. Birkhauser, 1992.

[8] G. Bruns. A practical technique for process abstraction. In *4th International Conference on Concurrency Theory (CONCUR'93)*, Lecture Notes in Computer Science 715, pages 37–49. Springer-Verlag, 1993.

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

[10] E.M. Clarke, O. Grumberg, and D.E. Long. Verification tools for finite-state concurrent systems. In J.W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, number 803 in Lecture Notes in Computer Science, pages 124–175. Springer, 1993.

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

[12] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.

[13] R. Cleaveland, P. Iyer, and D. Yankelevich. Optimality in abstractions of model checking. In *SAS'95: Proc. 2d. Static Analysis Symposium*, Lecture Notes in Computer Science 983, pages 51–63. Springer, 1995.

[14] G. Cousineau and M. Nivat. On rational expressions representing infinite rational trees. In *8th Conf. Math. Foundations of Computer Science: MFCS'79*, Lecture Notes in Computer Science 74, pages 567–580. Springer, 1979.

[15] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*. PhD thesis, University of Grenoble, 1978.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[17] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. *SIGPLAN Notices*, 12(8):1–12, 1977.

[18] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM Symp. on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[19] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[20] P. Cousot and R. Cousot. Inductive definitions, semantics, and abstract interpretation. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, pages 83–94. ACM Press, 1992.

[21] P. Cousot and R. Cousot. Higher-order abstract interpretation. In *Proc. IEEE Int'l. Conf. Programming Languages*. IEEE Press, 1994.

[22] D. Dams. *Abstract interpretation and partition refinement for model checking*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.

[23] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM TOPLAS*, 19:253–291, 1997.

[24] D. Dams, O. Grumberg, and R. Gerth. Abstract intepretation of reactive systems. In E.-R. Olderog, editor, *Proc. IFIP Working Conference on Programming Concepts, Methods, and Calculi*. North-Holland, 1994.

[25] V. Donzeau-Gouge. Denotational definition of properties of program's computations. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[26] E.A. Emerson and C.L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *First Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE, 1986.

[27] V. Gouranton. *Dérivation d'analyseurs dynamiques et statiques à patir de spécifications opérationnelles*. PhD thesis, University of Rennes, 1997.

[28] V. Gouranton and D. LeMétayer. Derivation of static analysers of functional programs from path properties of a natural semantics. Technical Report Research Report 2607, INRIA, 1995.

[29] C. Hankin, A. Mycroft, F. Nielson, and H. Riis-Nielson. *Principles of Program Analysis*. In Preparation, 1999.

[30] M. Hecht. *Flow Analysis of Computer Programs*. Elsevier, 1977.

[31] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Proc. 22d. ACM Symp. Principles of Programming Languages*, pages 393–407, 1995.

[32] N. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4*, pages 527–636. Oxford Univ. Press, 1995.

[33] N.D. Jones and S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Proc. 6th. ACM Symp. Principles of Programming Languages*, pages 244–256, 1979.

[34] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23:158–171, 1976.

[35] K. Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, 1981.

[36] M. Klein, D. Koschuetzki, J. Knoop, and B. Steffen. DFA&OPT-MetaFrame: a tool kit for program analysis and optimization. In *Proc. TACAS'96*, pages 422–426. Lecture Notes in Computer Science 1055, Springer, Berlin, 1996.

[37] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[38] Y.S. Kwong. On reduction of asynchronous systems. *Theoretical Computer Science*, 5:25–50, 1977.

[39] K. Larsen. Proof systems for hennessy-milner logic with recursion. In M. Duachet and M. Nivat, editors, *CAAP88*, number 299 in Lecture Notes in Computer Science. Springer-Verlag, 1988.

[40] K. Larsen. Modal specifications. In J. Sifakis, editor, *CAV'89*, number 407 in Lecture Notes in Computer Science, pages 232–246. Springer-Verlag, 1989.

[41] F. Levi. Abstract model checking of value-passing processes. In Annalisa Bossi, editor, *International Workshop on Verification, Model Checking and Abstra ct Interpretation, Port Jefferson, Long Island, N.Y.*, `http://www.dsi.unive.it/~bossi/VMCAI.html`, 1997.

[42] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[43] A. Melton, G. Strecker, and D. Schmidt. Galois connections and computer science applications. In *Category Theory and Computer Programming*, pages 299–312. Lecture Notes in Computer Science 240, Springer-Verlag, 1985.

[44] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[45] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 17:209–220, 1992.

[46] S. Muchnick and N.D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[47] F. Nielson. Semantic foundations of data flow analysis. Technical Report Report DAIMI PB-131, Aarhus University, Denmark, 1981.

[48] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[49] F. Nielson. Program transformations in a denotational setting. *ACM Trans. Prog. Languages and Systems*, 7:359–379, 1985.

[50] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117–242, 1989.

[51] F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proc. ACM POPL'97*, 1997.

[52] H. R. Nielson and F. Nielson. *Semantics with Applications, a formal introduction*. Wiley Professional Computing. John Wiley and Sons, 1992.

[53] D.A. Schmidt. Trace-based abstract interpretation of operational semantics. *J. Lisp and Symbolic Computation*. In press. Available from `www.cis.ksu.edu/~schmidt/papers/aiosh.ps.Z`

[54] D.A. Schmidt. Natural-semantics-based abstract interpretation. In A. Mycroft, editor, *Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 1–18. Springer-Verlag, 1995.

[55] D.A. Schmidt. Abstract interpretation of small-step semantics. In M. Dam and F. Orava, editors, *Proc. 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[56] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, Copenhagen University, 1991.

[57] O. Shivers. *Control Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[58] J. Sifakis. Property preserving homomorphisms of transition systems. In *Logics of Programs*, Lecture Notes in Computer Science 164. Springer, 1983.

[59] B. Steffen. Data flow analysis as model checking. In A. Meyer, editor, *Theoretical Aspects of Computer Software: TACS'91*, volume 526 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[60] B. Steffen. Generating data-flow analysis algorithms for modal specifications. *Science of Computer Programming*, 21:115–139, 1993.

[61] B. Steffen. Property-oriented expansion. In R. Cousot and D. Schmidt, editors, *Static Analysis Symposium: SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, 1996.

[62] B. Steffen, A. Classen, M. Klein, J. Knoop, and T. Margaria. The fixpoint analysis machine. In I. Lee and S. Smolka, editors, *Proc. CONCUR'95*, volume 962 of *Lecture Notes in Computer Science*, pages 72–87. Springer-Verlag, 1995.

[63] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 477–563. Oxford University Press, 1992.