Program Analysis *as* Model Checking of Abstract Interpretations

David Schmidt — Kansas State University * (USA)

Bernhard Steffen — Universität Dortmund** (D)

Abstract. This paper presents a collection of techniques, a methodology, in which abstract interpretation, flow analysis, and model checking are employed in the representation, abstraction, and analysis of programs. The methodology shows the areas of intersection of the different techniques as well as the opportunites that exist when one technique is used in support of another. The methodology is presented as a three-step process: First, from a (small-step) operational semantics definition and a program, one constructs a *program model*, which is a state-transition system that encodes the program's executions. Second, abstraction upon the program model is performed, reducing the detail of information in the model's nodes and arcs. Finally, the program model is analyzed for properties of its states and paths.

1 Motivation

Recent research suggests that the connections between iterative data-flow analysis and model checking are intimate. The most striking application is the use of a model checker to calculate iterative bit-vector-based data-flow analyses [44, 47–49]; this application depends on the encoding of the bit-vector's bits as boolean propositions which are decided by model checking.

But the application of a model checker as the engine for flow analysis is wider than bit-vector problems on sequential programs. Recent work by Steffen and his colleagues has adapted the basic construction to an efficient treatment of parallel programs [29], and work by Dwyer and others [19–21] shows how problems formally solved with data-flow analysis techniques are more simply expressed and solved by model-checking techniques. And there are numerous examples of program validation done with flow analysis that in the present day would be termed model checking [5, 6, 33, 34, 39, 40].

* Department of Computing and Information Sciences, Manhattan, Kansas (USA), schmidt@cis.ksu.edu. Supported by NSF/DARPA CCR-9633388 and NASA NAG-2-1209.

** Lehrstuhl für Programmiersysteme, Universität Dortmund, GB IV, Baroper Str. 301, D-44221 Dortmund (Germany), steffen@cs.uni-dortmund.de

To understand the connections between flow analysis and model checking, a third component, abstraction, more precisely, abstract interpretation, must be used. Abstract interpretation provides the foundation upon which safe program representations rest, and understanding why model checking is a proper computational tool for flow analysis depends upon an understanding of the underlying abstraction techniques.

The purpose of this paper is to introduce a collection of techniques, perhaps a methodology, in which abstraction, flow analysis, and model checking are employed in the representation, abstraction, and analysis of programs. The methodology is meant to show the areas of intersection of the different techniques as well as the opportunites that exist when one technique is used in support of another.

The methodology is based on a three-step process: First, from a (small-step) operational semantics definition and a program, one constructs a *program model*, which is a state-transition system that encodes one (or many, or all) of the program's executions. Second, one might abstract upon the program model, reducing the detail of information in the model's nodes and arcs. Finally, one analyses the model for properties of its paths, e.g., live variables information, redundancy information, safety properties, etc. The methods used within the three stages include abstract interpretation, flow analysis, and model checking. In some places the tools are interchangeable; in others, one tool clearly plays a singular role.

The paper is structured as follows: reviews of iterative data flow analysis and model checking are undertaken first. Next, the abstract interpretation of operational semantics definitions is reviewed, and it is shown how to construct program models. Following this, abstraction on program models is presented. Correctness issues are reviewed, and finally, the extraction of program properties within the framework is examined. The paper concludes with a brief look at extensions to the basic technique.

2 Iterative Flow Analysis

Entities of Observation

Data-flow information can be considered an "entity of observation"; the set of such entities is typically structured as a complete lattice, where the ordering on the elements, $\sqsubseteq$, models the precision of information where 'smaller' means more precise, and the lattice operations $\sqcup$ and $\sqcap$ construct least upper bounds and greatest lower bounds of arbitrary subsets. In this paper we typically assume a lattice structure for sets of entities of observation.

Program Models

A traditional iterative data-flow analysis works from a program's control-flow graph, which is one instance of a *program model*:

A *program model* $\mathcal{P}$ is a 5-tuple $(\mathcal{S}, \mathcal{A}, \rightarrow, s, E)$, where

1. $\mathcal{S}$ is a set of *nodes* or *program states*.

Source program: `while even x do x:=x div2 od; y:=2`

Program model is: $(\mathcal{S}, \mathcal{A}, \rightarrow)$,
where $\mathcal{S} = \{p_1, p_2, p_3, p_4\}$
$\mathcal{A} = \{\mathtt{even\ x}, \neg\mathtt{even\ x}, \mathtt{x} := \mathtt{x\ div2}, \mathtt{y} := 2\}$
$\rightarrow$ is defined below:

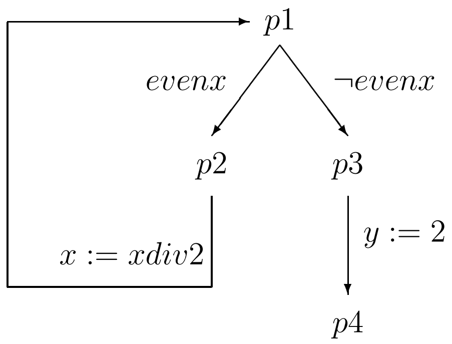


Fig. 1. An example program model for data-flow analysis.

2. $\mathcal{A}$ is a set of *actions*, modelling elementary statements
3. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is a set of *labelled transitions*, i.e., modeling the flow of control.
4. s and E are the start state and the set of end states respectively.

We will write $p \xrightarrow{a} q$ instead of $(p, a, q) \in \rightarrow$, say that p is the *source* of the arrow and q is the *target*, and call p an a -*predecessor* of q and q an a -*successor* of p. The set of all a -predecessors and a -successors will be abbreviated by $Pred_a$ and $Succ_a$, respectively. Similarly, for subsets $A \subseteq \mathcal{A}$, we call p an A -*predecessor* of q and q an A -*successor* of p, if A contains an action a for which these properties hold, and we abbreviate the set of all A -predecessors and A -successors by $Pred_A$ and $Succ_A$, respectively.

Remark: A program model represents an execution of a program or a family of executions of a program. A classic example of a program model is a control-flow graph, which encodes all possible executions of a program; Figure 1 presents a control-flow graph represented as a program model.

A standard issue is that a program model might have an infinite state set, e.g., consider the program model that results from executing a program that loops while counting upwards by ones. For static analysis, we work with program models that have finite state sets, and we will assume that adequate abstraction techniques like widening [14] can be employed, when necessary, to ensure finiteness, and we will not explore such techniques any further.

Iterative Data Flow Analyses

Given a program model, one can define upon it an iterative data-flow analysis by defining a lattice of entities of observation, and for each transition in the program model, a transfer function. One uses the lattice and functions to define a set of equations, one per program point (or state) in the model. Solution of the equations yields the desired data-flow information. We classify data-flow analyses as being either forwards or backwards[1]and as being either $\sqcup$- or $\sqcap$-based.

In simple terms (see [25, 28] for a formal definition), one defines a backwards-$\sqcup$ data-flow analysis of a program model by means of

- a set of entities of observation, D, partially ordered as a complete lattice.
- for each $A \in \mathcal{A}$, a *transfer function*, $f_A : D \to D$, that is monotonic on D: for $d, d' \in D, d \sqsubseteq d'$ implies $f_A(d) \sqsubseteq f_A(d')$.[2]
- for each program state, $p \in \mathcal{S}$, a *flow equation*,

$$val_p = \bigsqcup \{ f_A(val_q) \mid (p, A, q) \in \rightarrow \}$$

The adequate *solution* to the set of flow equations can be determined by a least fixed-point computation.

Similarly, one can define a backwards-$\sqcap$-flow analysis by replacing the previous occurrence of $\sqcup$ by $\sqcap$ and computing the greatest-fixed-point solution of the equations. Finally, one can define a *forwards* analysis by swapping the occurrences of val_p and val_q in the flow equation scheme.

One example of a backwards-$\sqcup$ data-flow analysis is live-variables analysis on a control-flow graph, which is formalized by

- $D = 2^{Var}$, where Var is the collection of the program's variables;
- $f_A(s) = \mathit{UsedIn}_A \cup (\mathit{notModifiedIn}_A \cap s)$, where UsedIn_A defines those variables referenced in action A, and $\mathit{notModifiedIn}_A$ defines those variables that are not modified (assigned to) in A.

The flow equations follow from the above information. Figure 2 shows the live variables analysis for the program model in Figure 1.

3 Model Checking

Model checking is employed to validate properties of finite-state program models. This scope is sufficient for the purposes of this paper, but interested readers are referred to [4] for techniques covering infinite state models that explicitly model interprocedural structure. The considered properties are logical predicates

[1] We do not consider bi-directional algorithms here, which can, in fact, usually be decomposed into uni-directional components [30].

[2] It is common to demand that each f_A be *distributive*: for $D' \subseteq D, f_A(\bigsqcup D') = \bigsqcup \{ f_A(d) \mid d \in D' \}$. Distributivity ensures that the fixed-point computation one performs on a set of flow equations calculates the same result as one obtains from a *meet-over-all-paths* analysis [27].

$$
\begin{array}{llll}
val_{p1} = f_{\{\texttt{evenx}\}}(val_{p2}) \cup f_{\{\neg\texttt{evenx}\}}(val_{p3}) & \textit{where} & f_{\{\texttt{evenx}\}}(s) = \{\texttt{x}\} \cup (\{\texttt{x},\texttt{y}\} \cap s) \\
val_{p2} = f_{\{\texttt{x:=xdiv2}\}}(val_{p1}) & & f_{\{\neg\texttt{evenx}\}}(s) = \{\texttt{x}\} \cup (\{\texttt{x},\texttt{y}\} \cap s) \\
val_{p3} = f_{\{\texttt{y:=2}\}}(val_{p4}) & & f_{\{\texttt{x:=xdiv2}\}}(s) = \{\texttt{x}\} \cup (\{\texttt{y}\} \cap s) \\
val_{p4} = \textit{initialization-information} & & f_{\{\texttt{y:=2}\}}(s) = \{\texttt{x}\} \cap s
\end{array}
$$

For *initialization-information*$= \{\texttt{y}\}$, the analysis computes:

$$val_{p1} = \{\texttt{x}\} \quad val_{p2} = \{\texttt{x}\} \quad val_{p3} = \{\} \quad val_{p4} = \{\texttt{y}\}$$

Fig. 2. Flow equations and solution for live-variables analysis

that typically discuss dependencies between occurrences of specific actions within paths in the model, e.g., "all paths starting from the current program point eventually include an a action" or "there exists a path including an a action infinitely often."

The Syntax

As a logic for specifying static analysis algorithms, we will consider a variant of the temporal logic CTL [7] that includes a *parameterized* version of the "Henceforth" operator, the key for specifying *qualified safety* properties. This logic is particularly suited for expressing properties of states within a given (program) model.

The logic's syntax is parameterized on a denumerable set $\mathcal{B}$ of atomic propositions[3] on states and a complete lattice of actions $\mathcal{A}$. Let β range over $\mathcal{B}$ and $A \subseteq \mathcal{A}$:

$$\Phi ::= \beta \;\mid\; \Phi \wedge \Phi \;\mid\; \neg\Phi \;\mid\; [\,A\,]\,\Phi \;\mid\; \overline{[\,A\,]}\,\Phi \;\mid\; \mathbf{AG}_A\,\Phi \;\mid\; \overline{\mathbf{AG}}_A\,\Phi \;\mid$$

We write $p \models \Phi$ to denote that proposition Φ holds true at state p, and we say that p *satisfies* Φ.

The Semantics

Satisfaction is defined with respect to a given program model $\mathcal{P}$ containing p according to the following intuition: $p \models \beta$ is assumed to be decidable, $p \models \Phi_1 \wedge \Phi_2$ if p satisfies both Φ_1 and Φ_2. $p \models \neg\Phi$ if p does not satisfy Φ, and $p \models [\,A\,]\,\Phi$ if every one of p's A-successor states satisfies Φ. Note that this implies p satisfies $[\,A\,]\,\mathit{ff}$ exactly when p has no A -successors. Analogously, p satisfies $\overline{[\,A\,]}\,\Phi$ if every A -predecessor satisfies Φ. Thus in analogy, a program state p satisfies $\overline{[\,A\,]}\,\mathit{ff}$ exactly when p has no A -predecessors. Finally, $p \models \mathbf{AG}_A\,\Phi$ if Φ holds in every state reachable from p via A-transitions, and it satisfies $\overline{\mathbf{AG}}_A\,\Phi$, if exactly the same property holds for the inverted flow of control in the program model.

[3] In this paper we simply consider three atomic propositions, tt, `start`, and `end`, which characterize the set of all states, the start state and the set of end states, respectively. In general, any set of decidable characterizations of sets of states could be taken.

Remark: The difference between the standard Henceforth operator $\mathbf{AG}\ \Phi$ and the parameterized version $\mathbf{AG}_A\ \Phi$ is with respect to reachability: For $p \models \mathbf{AG}\ \Phi$ to hold, *all* states reachable from *all* transitions from p must satisfy Φ, whereas for $p \models \mathbf{AG}_A\ \Phi$, only those states reached from p by traversing transitions labelled by an action from A must satisfy Φ.
The formal semantic definition of the logic derived from the modal μ-calculus can be found in [47].

In the following we will write $[\,.\,]$ or $\overline{[\,.\,]}$ instead of $[\,\mathcal{A}\,]$ or $\overline{[\,\mathcal{A}\,]}$. Moreover, as usual, we can define the following duals to the operators of our language and the implication operator $\Rightarrow$ by:

$$\begin{array}{rlrl} ff &= \neg tt & \mathbf{EF}_A\Phi &= \neg\mathbf{AG}_A\neg\Phi \\ \Phi_1 \vee \Phi_2 &= \neg(\neg\Phi_1 \wedge \neg\Phi_2) & \overline{\mathbf{EF}}_A\Phi &= \neg\overline{\mathbf{AG}}_A\neg\Phi \\ \langle\, A\,\rangle\ \Phi &= \neg[\,A\,]\ (\neg\Phi) & \Phi \Rightarrow \Psi &= \neg\Phi \vee \Psi \\ \overline{\langle\, A\,\rangle}\ \Phi &= \neg\overline{[\,A\,]}\ (\neg\Phi) & & \end{array}$$

The Application

A crucial connection between iterative data-flow analysis and model checking was demonstrated by Steffen [47, 48], who noted the similarities between computing the results of a set of data-flow equations and computing the set of states that satisfied a modal mu-calculus specification. For example, the equation scheme that computes live variables,

$$Live_p = \bigcup\{\, UsedIn_a \cup (notModifiedIn_a \cap s)\ \mid source(a) = p\}$$

can be transliterated into the following formula of our logic that expressess whether a variable, x, is live at a program point:

$$isLive_x = \mathbf{EF}_{\{a\ \mid\ a \neq\ mod_x\}}\langle\, use_x\,\rangle\ tt$$

This formulation is based on abstractions of the actions that annotate the arcs of a program model: Assignment statements, $v := e$, are abstracted to actions of the form, mod_v and use_u, for those variables, u, used in e; tests, e, are abstracted to actions use_u, for those variables, u, used in e.

When one model checks the assertions, $p \models isLive_x$, for all variables x, one in effect computes the bit-vector solution for the flow equation $Live_p$. A standard iterative model checker would do this, e.g., for the variable x of Figure 1 by initially 'marking' all the states of the program model satisfying $\langle\, use_x\,\rangle\ tt$ with 'true', and, subsequently, stepwisely spreading this information to all states having a $\{a \mid a\ \neq mod_x\}$-successor being marked 'true' until a fixed point is reached, i.e., no further state can be marked 'true' according to the described rules. For the considered example program the fixed point is reached already by the initialization procedure.

Technically, formulae of the logic considered here would be first translated into modal equational systems [10], a compact representation of the modal μ-calculus, which have the appropriate granularity to directly steer the fixpoint computation process. $isLive_x$ would be translated into the following min-equation[4]

$$MIN: \qquad isLive_x = \langle\, use_x \,\rangle\, tt \vee \langle\, \{a \mid a \neq mod_x\} \,\rangle\, isLive_x$$

The fixpoint computation needs one bit for each such equation. In particular, as expected, one bit (per program variable) is sufficient for checking liveness of variables.

Why Safety Properties

In this paper, we will explore the connection between flow analysis and model checking while focussing on the preservation of safety properties, i.e. properties which are guaranteed to hold for *all* reachable states. In other words these properties are characterized by holding *everywhere* along *all* program executions, and therefore correspond to the properties which can be specified using the parameterized Henceforth operator $\mathbf{AG}_x$.

$isLive_x$ is not a safety property, as it quantifies *existentially* over the execution paths [5] as well as over the states of a given execution.[6] However, this causes no real harm, as one is not interested in the fact whether a variable is live, but in the complement, as the detection of dead variables is a key for dead-code elimination, and this property is indeed a safety property. And indeed, as program transformations must be correct for all possible program executions, it is clear that they must be based upon information which is horizontally universally valid. In contrast, there seem to be relevant problems, like e.g. *down safety* [30], i.e., the property that a certain expression will be executed in every continuation of the program, which seem to ask for existential vertical quantification. Surprisingly, also these properties typically can be better formulated as a (parameterized) safety property than as a liveness property. This is due to the fact that the program executions we are interested in are typically *terminating* program executions. E.g., for down safety, we do not require the expression to be executed on paths 'starving' in a loop,[7] but only on those which reach the end point of the program.[8] This property can readily be expressed with a single parameterized Henceforth operator, if we assume that the end point of the program is characterized by the atomic proposition *end*:

$$\mathbf{AG}_{\{a \mid a \neq use_x\}}(\neg end \wedge \langle\, mod_x \,\rangle\, ff)$$

[4] The distinction between min-equations and max-equations allows us to specify whether a minimal or a maximal fixpoint constituts the desired solution.

[5] This is apparent from the 'E' of the 'EF' operator, which means this 'horizontal' existential quantification.

[6] This is apparent from the 'F' of the 'EF' operator, which means this 'vertical' existential.

[7] Note that in control flow graphs, there is nothing to force a computation to leave a loop.

[8] In some sense this can be regarded as a 'partial correctness' view to the problem.

$$Store = Identifier \rightarrow Val$$
$$Val = Nat$$

$$c \ ::= \ v := e \mid \texttt{skip} \mid c_1; c_2 \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{fi} \mid \texttt{while}\ e\ \texttt{do}\ c\ \texttt{od}$$

$$v := e, \sigma \xrightarrow{v:=e} [x \mapsto v]\sigma, \quad \text{where } [e]\sigma \Rightarrow v$$

$$skip, \sigma \xrightarrow{skip} \sigma \qquad \frac{c_1, \sigma \xrightarrow{c} \sigma'}{c_1; c_2, \sigma \xrightarrow{c} c_2, \sigma'} \qquad \frac{c_1, \sigma \xrightarrow{c} c_1', \sigma'}{c_1; c_2, \sigma \xrightarrow{c} c_1'; c_2, \sigma'}$$

The operational semantics of `while` is not given by a separate rule but deduced according to the following identity:

$$\texttt{while}\ e\ \texttt{do}\ c\ \texttt{od} \equiv \texttt{if}\ e\ \texttt{then}\ c; \texttt{while}\ e\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}$$

$$\frac{[e]\sigma \Rightarrow true}{\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{fi}, \sigma \xrightarrow{e} c_1, \sigma} \qquad \frac{[e]\sigma \Rightarrow false}{\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{fi}, \sigma \xrightarrow{\neg e} c_2, \sigma}$$

Note: $[e]\sigma \Rightarrow v$ is defined by a relation, not given here.

Fig. 3. Small-step semantics for imperative language

Technically, we will present a *simulation-based* criterion for guaranteeing safety, which bridges the gap between the operational semantics of a program model on different levels of abstraction.

4 Operational Semantics and Abstractions

Our definition of program model includes not only control-flow graphs but also execution traces, their abstract interpretations, behaviour trees, and other kinds of abstract transition systems. Indeed, a program's control-flow graph can be simply regarded as a program model arising from an abstract interpretation of the program's semantics, where the program's store is abstracted to *nil*.

Small-Step Structural Operational Semantics

We assume that a programming language comes with a small-step structural operational semantics. An example of such a semantics appears in Figure 3. It is convenient to label each transition with the primitive command/expression that generates the transition.

Figure 4 displays part of the operational semantics of the program in Figure 1 generated from the small-step semantics rules. We call the derivation in the figure a *concrete computation*. Note that the concrete computation is also a program model.

Let $p1$ denote `while even x do x := x div2 od; y := 2`
$p2$ denote `x := x div2; while even x do x := x div2 od; y := 2`
$p3$ denote `y := 2`

Let $\sigma_{m,n}$ denote the store $[\mathtt{x} \mapsto m][\mathtt{y} \mapsto n]$

$$\begin{tikzcd}
p1, \sigma_{6,1} \arrow[d, "evenx"] \\
p2, \sigma_{6,1} \arrow[d, "x := xdiv2"] \\
p1, \sigma_{3,1} \arrow[d, "\neg evenx"] \\
p3, \sigma_{3,1} \arrow[d, "y := 2"] \\
\sigma_{3,2}
\end{tikzcd}$$

Fig. 4. Concrete computation

Standard Models

Given a programming language with a small step-semantics, the standard program model looks as follows:

- program states are the configurations appearing between transition steps (e.g., program point, store pairs, (p, σ))
- actions are the elementary statements and expressions of the language.
- transitions are defined by the small steps and are labeled with the corresponding primitive command/expression.
- the start state is the initial configuration, and the end states are those states having no successor.

As we will see, these standard models fully comprise the computations associated with a small-step semantics. In fact, a concrete computation and the corresponding standard program model are mutual simulations of each other in the formal sense established in Section 5. Thus they are semantically fully interchangeable for the purposes considered here, and are therefore a convenient abstract program representation.

Abstract Interpretation

For static analysis purposes, we wish to generate a finite program model that comprises all relevant concrete operational semantics executions. To do this, we employ the abstract interpretation methodology of Cousot and Cousot [14–16]: We replace the concrete domains, *Val* and *Store*, by abstract domains, *AbsVal* and *AbsStore*, and we compute a program's operational semantics with the new domains to arrive at the corresponding program model. For ease of exposition, we require that the abstract domains be complete lattices. With the appropriate notion of abstraction of operations, the abstract domains will generate abstract program models that simulate the corresponding concrete computation models (cf. Section 5).

Galois Connections

To prove the simulation property, one must relate the abstract to concrete data domains by means of Galois connections[9] [14]. If we think of an abstract data domain, *AbsVal*, as the "entities of observation," then we must establish which subset of *Val* is denoted by an "entity" $a \in AbsVal$. We do so with a monotone mapping, a *concretization function*, $\gamma : AbsVal \to 2^{Val}$. Of course, there should be an inverse correspondence, a monotone $\alpha : 2^{Val} \to AbsVal$, the *abstraction function*; in particular, $\alpha(\{c\})$ identifies the element in *AbsVal* that "represents" $c \in Val$. We desire that (α, γ) form a Galois connection, because this implies $\alpha(\{c\}) = \sqcap \{a \,|\, c \in \gamma(a)\}$. It is well known that one adjoint of a Galois connection uniquely determines the other.

A useful intuition is that γ defines a binary "simulation" relation: for $c \in Val$, $a \in AbsVal$, c *safe* a iff $c \in \gamma(a)$. That is, c is simulated or safely approximated by a. Further, it is possible to begin with the binary relation, *safe*, and define a Galois connection from it: If *safe* is is both *U-closed*, i.e., c *safe* a_1 and $a_1 \sqsubseteq a_2$ imply c *safe* a_2, and *G-closed*, i.e., c' *safe* $\sqcap A$, where $A = \{a' \mid c' \textit{ safe } a'\}$, then one obtains a Galois connection [43].

Because of these equivalences, we define a Galois connection by any one of a γ, α, or UG-closed relation in the sequel.

Control Flow Graphs

Here is the abstraction for control-flow graphs:

$$\begin{array}{ll} AbsVal = \{nil\} & \gamma : AbsValue \to 2^{Val} \\ AbsStore = Identifier \to AbsVal = \{nil\} & \quad \gamma(nil) = Val \\ & \gamma : AbsStore \to 2^{Store} \\ & \quad \gamma(nil) = Store \end{array}$$

[9] Recall that a Galois connection is a pair of monotone functions, $(f\colon P \to Q, g\colon Q \to P)$, 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 $g(q)$).

Let $p1$ denote `while even x do x := x div2 od; y := 2`
$p2$ denote `x := x div2; while even x do x := x div2 od; y := 2`
$p3$ denote `y := 2`

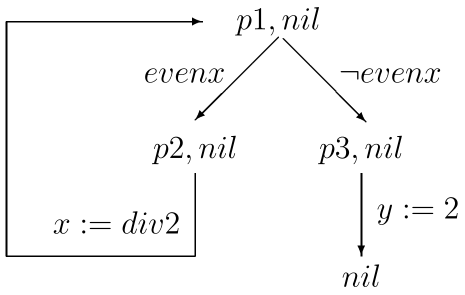


Fig. 5. Control-flow tree generated as an abstract computation

Figure 5 shows the standard program model generated from the control-flow abstraction of the program in Figure 4; it is of course an isomorphic representation of the usual control-flow graph. (Notice that the state, $(p1, nil)$, repeats in the model; hence, the arc from $(p2, nil)$ to $(p1, nil)$ can be written as a backwards arc.)

The next abstraction analyses a program's execution with respect to even-odd properties:

$$\mathit{AbsVal} = \{\bot, e, o, \top\} \text{ (usual partial ordering)}$$
$$\mathit{AbsStore} = \mathit{Identifier} \rightarrow \mathit{AbsVal} \text{ (usual partial ordering)}$$

$$\begin{aligned} &\gamma : \mathit{AbsVal} \rightarrow 2^{Val} \\ &\quad \gamma(\bot) = \{\} \\ &\quad \gamma(e) = \{n \in Value \mid n \; modulo \; 2 = 0\} \\ &\quad \gamma(o) = \{n \in Value \mid n \; modulo \; 2 = 1\} \\ &\quad \gamma(\top) = \mathit{Val} \end{aligned}$$

$$\begin{aligned} &\gamma : \mathit{AbsStore} \rightarrow 2^{Store} \\ &\quad \gamma(s) = \{s' \mid \text{ for all } i, s'(i) \in \gamma(s(i))\} \end{aligned}$$

Abstract Operations

Once the concrete data domains are correctly abstracted by abstract data domains, one must abstract the operations that use the data domains and also the small-step semantics rules that use the operations. Assuming that an operation, $f_C : \mathit{Val} \rightarrow \mathit{Val}$ is a function, we say that f_C is *safely approximated* by function $f_A : \mathit{AbsVal} \rightarrow \mathit{AbsVal}$ iff

$$\text{for all } a \in AbsValue, \{f_C(m) \mid m \in \gamma(a)\} \subseteq \gamma(f_A(a))$$

Returning to binary simulation relations, the above definition of safe approximation is equivalent to this formulation: $c \; \mathit{safe} \; a$ implies $f_C(c) \; \mathit{safe} \; f_A(a)$.

In the case of even-odd analysis, the abstraction of the successor operation, $succ(n) = n + 1$, is naturally approximated by this function:

$$\begin{array}{ll} succ(o) \Rightarrow e & \quad succ(\top) \Rightarrow \top \\ succ(e) \Rightarrow o & \quad succ(\bot) \Rightarrow \bot \end{array}$$

Sometimes, imprecision can arise due to abstraction; consider the abstract version of division by 2, which produces an answer of $\top$ when an even- or odd-valued number is divided by 2:

$$div2(\bot) \Rightarrow \bot \qquad div2(a) \Rightarrow \top, \text{ for all } a \in \{e, o, \top\}$$

Abstract Operational Semantics

One uses the proved-safe abstract operations to define a small-step semantics for generating abstract program models. The intuition is that one instantiates the rule schemes to use the abstract operations rather than the concrete ones. Then, one uses the instantiated rules to generate a program model. But a standard problem is deciding the tests of conditional commands. (For example, how does the even-odd analysis decide the test of the conditional command in this example: $\mathtt{x} := \mathtt{x}\ \mathtt{div2}; \mathtt{if\ even\ x\ then\ c_1\ else\ c_2\ fi}$?) A solution is to rewrite the rules for the conditional as follows:

$$\frac{[e]\sigma \Rightarrow v,\ \ true \sqsubseteq v}{\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \mathtt{fi}, \sigma \xrightarrow{e} c_1, \sigma} \qquad \frac{[e]\sigma \Rightarrow v,\ \ false \sqsubseteq v}{\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \mathtt{fi}, \sigma \xrightarrow{\neg e} c_2, \sigma}$$

This format assumes that the concrete *Boolean* domain is abstracted to a lattice, $\{\bot, \mathit{true}, \mathit{false}, \top\}$. If one wishes to retain the format of the original small-step rule schemes (and not perform abstraction on *Boolean*), then one must utilize relational definitions of abstract operations.[10] But this topic will not be developed further in this paper.

When we use abstract semantics rule schemes like the two above, the program models contain nondeterministic branching (as would be the case in the example program, $\mathtt{x} := \mathtt{x}\ \mathtt{div2}; \mathtt{if\ even\ x\ then\ c_1\ else\ c_2\ fi}$), because the outcome of the test expression of a conditional command might not be uniquely *true* or *false*.

Figure 6 shows two abstract program models generated from the even-odd abstraction for the program in Figure 4. The first program model shows the result of analyzing a program where `x` starts as odd-valued; the second model results when `x` starts as even-valued. The examples show that more precise information about program paths can be elicited than what one obtains from a control-flow graph. Such models might be used if temporal properties of the program's paths must be validated, or if the inputs to a program are restricted to a particular range (e.g., input `x` is restricted to be odd-valued), or if better quality code can be

[10] For example, a relational definition of $div2$ would be: $div2(v) \Rightarrow e$ and also $div2(v) \Rightarrow o$, for all $v \in \{o, e, \top\}$. Also, $div2(\bot) = \bot$. Similarly, a relational definition of the predicate, $even$, would be $even(o) \Rightarrow false$, $even(e) \Rightarrow true$, $even(\top) \Rightarrow false$, $even(\top) \Rightarrow true$, $even(\bot) \Rightarrow \bot$.

Let $p1$ denote `while even x do x := x div2 od; y := 2`
$p2$ denote `x := x div2; while even x do x := x div2 od; y := 2`
$p3$ denote `y := 2`

Let $\sigma_{a,b}$ denote the store $[\mathtt{x} \mapsto a][\mathtt{y} \mapsto b]$

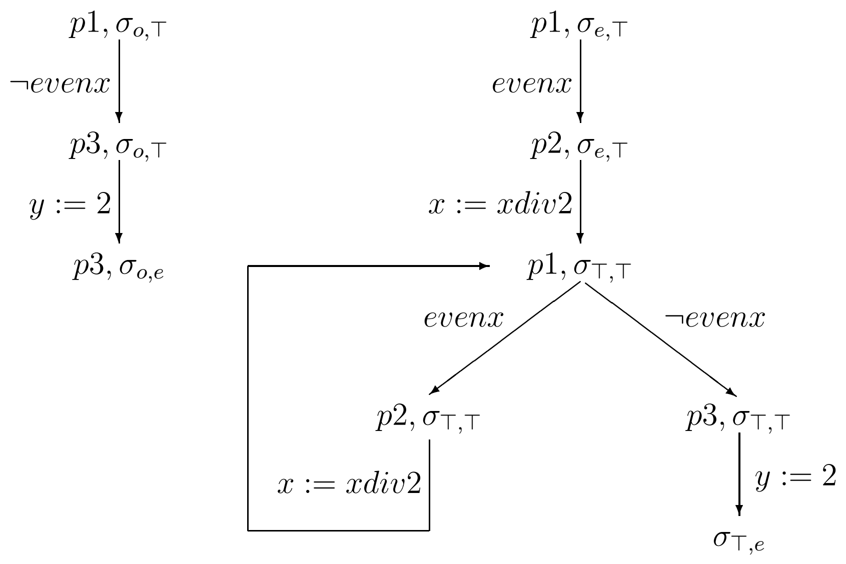


Fig. 6. Two abstract computations for even-odd analysis

generated when, say, the inputs to a command are even-valued. (If we analyze the example program where both `x` and `y` are initialized to $\top$, the resulting program model is isomorphic to the control-flow graph.)

Of course, we have the obligation of proving that the abstract program model is a safe approximation or simulation of the concrete model. Thus far, the safety properties enforced by Galois connections are "local" in that they relate concrete values to abstract values and concrete program states to abstract states. But there is still the burden of relating the *paths* in concrete program models (executions) to the paths in an abstract program model, that is, we must define precisely which set of concrete program models are approximated by an abstract program model. This must be established by a "global" safety property, to be defined in Section 5.

5 Simulations

One step of the "globalization" of the abstraction of a program model is the abstraction of the actions that label the arcs of the model. In the following, we will first introduce abstraction on actions by means of the prominent Use-Mod abstraction, and afterwards we will consider the correctness of such abstractions relative to the proofs of correctness of the abstractions on data domains and operations.

Abstraction of Actions

The development in the previous sections of abstraction of data domains and operations is standard. But we can abstract on source program syntax as well, more precisely, we can abstract on the actions that label the arrows of a program model

One example of abstraction of actions is the replacement of a command or expression by its Use-Mod information. For example, the Use-Mod abstraction of the command, `x:=x + y`, would be $\{mod_x, use_x, use_y\}$, and the abstraction of the expression, `even x`, would be $\{use_x\}$.

Here is a formalization of Use-Mod abstraction; because Use-values are fundamentally covariant and Mod-values are fundamentally contravariant, we must take care in formulating the lattice, *UseMod*:

$$\begin{aligned}
&Mod = \{\mathit{isExpr}\} \ \cup \ \{mod_x \mid x \in \mathit{Identifier}\}\\
&\mathit{Use} = \{use_x \mid x \in \mathit{Identifier}\}\\
&\mathit{UseMod} = \{\bot, \top\} \ \cup \textstyle\sum_{i \in Mod} 2^{\mathit{Use}}
\end{aligned}$$

The *Use* set is ordered by superset inclusion, and *UseMod* is ordered as a disjoint union: For $(i_1, S_1), (i_2, S_2) \in \mathit{UseMod}$, $(i_1, S_1) \sqsubseteq (i_2, S_2)$ iff $i_1 = i_2$ and $S_2 \subseteq S_1$, and also $\bot \sqsubseteq v$ and $v \sqsubseteq \top$, for all $v \in \mathit{UseMod}$. (In the examples, we will continue to write (mod_x, S) as $\{mod_x\} \cup S$ and (isExpr, S) as S.)

Next, we can define precisely how to map a single expression or assignment to its Use-Mod approximation:

$$\begin{aligned}
&\alpha_0 : \mathit{Command} \cup \mathit{Expression} \to \mathit{UseMod}\\
&\quad \alpha_0(x := e) = (mod_x, \{use_v \mid v \text{ appears in } e\})\\
&\quad \alpha_0(e) = (\mathit{isExpr}, \{use_v \mid v \text{ appears in } e\})
\end{aligned}$$

We define the lower adjoint, α, of a Galois connection in the expected way: $\alpha(S) = \bigsqcup\{\alpha_0(s) \mid s \in S\}$. The mate to α must be this γ:

$$\begin{aligned}
&\gamma : \mathit{UseMod} \to 2^{\mathit{Command} \cup \mathit{Expression}}\\
&\quad \gamma(\mathit{isExpr}, S) = \{e \mid v \in S \text{ implies } v \in e\}\\
&\quad \gamma(mod_x, S) = \{x := e \mid v \in S \text{ implies } v \in e\}\\
&\quad \gamma(\bot) = \{\}, \qquad \gamma(\top) = \mathit{Command} \cup \mathit{Expression}
\end{aligned}$$

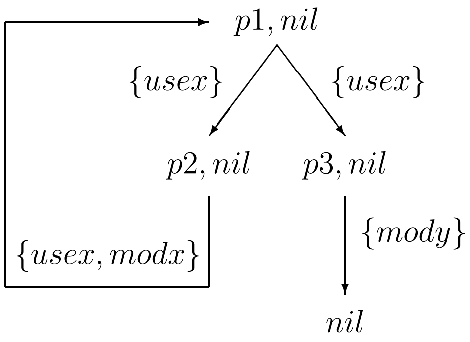


Fig. 7. Program model with actions abstracted to mod-use information

Model Construction and Abstraction of Actions

We intend to apply the $UseMod$ domain to a live-variables analysis, and we do so in a surprising fashion: we first generate a program model where we do *not* abstract the set of actions, and we then replace actions, A, on the model's arcs by $\alpha(A)$. As an example, Figure 7 shows the replacement of the actions in the control-flow graph in Figure 5 by their abstractions: As we will justify in the next section, the revised program model is a safe simulation of all concrete executions of the program, hence it is possible to check safety properties on the model. In particular, one can validate that variable `x` is definitely dead at a program point, p, by performing the model check, $p \models \neg isLive_x$. This is the way that we formulate models for implementing bit-vector analyses.

Why did we take this approach of generating a program model first before abstracting its actions to Use-Mod information? After all, the standard abstraction methodology suggests that we apply the Use-Mod abstraction to the small-step semantics rule schemes first and generate a new set of small-step rule schemes that generate transitions based on Use-Mod "syntax." But the Use-Mod abstraction discards so much syntactic structure that the resulting abstract rule schemes generate safe program models of worthless precision.

In such a situation, the way around the problem is to perform the above trick of abstracting the actions on an earlier existing, known-safe, program model. The result is a safe program model *for the set of concrete computations modelled safely by the earlier existing model*. In the above example, the Use-Mod abstraction gains precision by attaching itself to the control-flow-graph program model, which has a more precise branching structure than that obtained by using the small-step rule schemes based on Use-Mod syntax to generate a program model.

Having noted the above, we note there do exist examples in the literature where abstraction on syntax generates small-step rule schemes that *can* be used to generate useful program models. The best known example is the "Kleene-star abstraction" technique of Codish, Falaschi, and Marriott [12, 11], where the size of an unfolded Prolog program is controlled by joining together goal clauses that use the same predicate symbol. A Prolog program is therefore abstracted into a syntax of regular expressions. Schmidt [42] uses a similar regular expression language to abstract the syntax of pi-calculus configurations. Approximations of program syntax by sets of context-free grammar rules have been done for functional programs by Giannotti and Latella [23] and for pi-calculus programs by Venet [52, 53].

Abstract Actions as Abstract Operations

In the above narrative, we did not place formal restrictions on the form of Galois connection that relates concrete actions to abstract actions. But in practice, the concrete actions represent commands—transfer functions—that update program state. When we write a transition, $c_1 \xrightarrow{A_c} c_2$, with a concrete semantics, and when we write the abstracted transition, $a_1 \xrightarrow{A_c} a_2$, in the abstract semantics, we assume that the "local simulation" described in the previous section is preserved by the abstraction of actions—c_1 *safe* a_1 implies c_2 *safe* a_2. This property is so desirable, we spend some time to study it.

Consider a concrete domain, D_c and the abstract domain, D_a, connected by a Galois connection (α, γ). (Standard instantiations of these domains are 2^{Store} and *AbsStore* of the previous section, but as the abstraction process may well be iterated, other instantiations are possible as well.)

The Galois connection induces a binary simulation relation, $\mathit{safe} \subseteq D_c \times D_a$ on the two domains (that is, for all $v \in D_c$, $u \in D_a$, v *safe* u iff $v \sqsubseteq \gamma(u)$, where $\gamma : D_a \to D_c$ is the upper adjoint of the Galois connection).

Next, assume that every action possesses a transfer function: For each action, $A_c \in \mathcal{A}_c$, let $[\![A_c]\!]_c : D_c \to D_c$ be its transfer function, and similarly, for each abstracted action, $A_a \in \mathcal{A}_a$, let $[\![A_a]\!]_a : D_a \to D_a$ be its transfer function. (Thus, for this story to be sensible, abstract actions must have transfer functions as well.)

The desired preservation property for action abstraction reads as follows:

$$\begin{array}{l}\text{for all } A_c \in \mathcal{A}_c, \text{ and for all } A_a \in \mathcal{A}_a, \text{ such that } A_a \text{ abstracts } A_c,\\ \quad\text{for all } v \in D_c,\ u \in D_a,\ v\ \mathit{safe}\ u \text{ implies } [\![A_c]\!]_c(v)\ \mathit{safe}\ [\![A_a]\!]_a(u)\end{array}$$

(By "A_a abstracts A_c," we mean of course that $A_c \in \gamma_A(A_a)$ or equivalently, $\alpha_A(A_c) \sqsubseteq A_a$, using the Galois connection, (α_A, γ_C), between the two action sets.)

If one decodes the consequent of the implication into the Galois connection between D_c and D_a, one obtains this inclusion at the concrete level D_c:

$$\bigsqcup\{[\![A_c]\!]_c(v) \mid v \sqsubseteq \gamma(u)\} \sqsubseteq \gamma([\![A_a]\!]_a(u))$$

which is equivalent to the following inclusion on the functional level, which we will use in the next section:

$$\alpha \circ [\![A_c]\!]_c \sqsubseteq [\![A_a]\!]_a \circ \alpha$$

From here on, we demand the following of the abstraction of concrete actions to abstract actions: *if* A_a *abstracts* A_c*, then* $\alpha \circ [\![A_c]\!]_c \sqsubseteq [\![A_a]\!]_a \circ \alpha$. That is, the abstraction of actions preserves simulation on program states.

Remark Depending on the abstraction of program state, the preservation property just defined can be achieved trivially. Consider again the live-variables example in Figure 7, where both the "concrete" and "abstract" store sets are merely $\{nil\}$. (In the Figure, we do not care about the value of store, because it is the information on the arcs of the program model that matter.) The semantic transfer functions for both "concrete" and "abstract actions" are trivial.

We can make the live-variables example more interesting by letting concrete and abstract program stores tell us which expressions *will-be-used*$_e$ for all expressions e. Assignments of the form $x := t$ operate on these entities backwards in the following fashion:

$$[\![x := t]\!]_c(S) = \{e \,|\, e \text{ is a subexpression of } t \text{ or } (\, e \in S \text{ and } [e]\sigma = [e]\sigma[[t]\sigma/x]\,)\,\}$$

whereas the operation of the corresponding Use-Mod abstraction, which we assume here to be straightforwardly extended to expressions, is defined by

$$[\![\alpha_0(x := t)]\!]_a(S) = \{e \mid use_e \in \alpha_0(x := t) \text{ or } (\, e \in S \text{ and } mod_e \notin \alpha_0(x := t)\,)\,\}$$

As long as we do not change the graph structure of the underlying program model, it is easy to verify that the Use-Mod abstraction computes a safe approximation of the concrete *will-be-used*$_e$ information in the following sense: Whenever the abstract analysis tells us that an expression will be used in future then this holds also of the concrete analysis. (There are obviously situations where the reverse implication fails.)

The following section addresses the problem of changing graph structure, e.g. the collapsing of nodes, which is essential for fighting the state explosion problem.

Simulation Relations

The previous sections formalized in what sense a concrete value/operation/action is safely simulated or approximated by an abstract value/operation/action; these simulations are "local," and it is time to extend simulation to program models, giving us a "global simulation" property that holds between program models, so that we can say precisely when a program model built with abstract operations and transitions safely describes a concrete program execution, which is represented by a concrete program model. Let, therefore

- $\mathcal{P}_c = (\mathcal{S}_c, \mathcal{A}_c, \rightarrow_c, s_c, E_c)$ and $\mathcal{P}_a = (\mathcal{S}_a, \mathcal{A}_a, \rightarrow_a, s_a, E_a)$ be two program models,
- D_c and D_a be complete lattices constituting the concrete respectively abstract domains
- $[\![\,.\,]\!]_c : \mathcal{A}_c \rightarrow (D_c \rightarrow D_c)$ and $[\![\,.\,]\!]_a : \mathcal{A}_a \rightarrow (D_a \rightarrow D_a)$ meaning functions, associating monotone/distributive transfer functions with their argument actions,

– (α, γ) be a pair of adjoints for D_c and D_a,

Under these circumstances, we wish to formalize that $\mathcal{P}_c$ is safely approximated by $\mathcal{P}_a$. The intuition is that every execution path in $\mathcal{P}_c$ is imitated by one in $\mathcal{P}_a$. Since the arrows in the paths are labelled with actions, we must also verify that the corresponding labels on the corresponding arrows are compatible according to the abstraction function.

In this setting, the notion of safe approximation is essentially the property of simulation from concurrency theory. We formalize these intuitions precisely by means of a *simulation relation*, $\mathcal{R}_\alpha$:

$\mathcal{P}_a$ α -simulates $\mathcal{P}_c$, iff there exists a binary relation $\mathcal{R}_\alpha \in \mathcal{S}_c \times \mathcal{S}_a$ satisfying that

– $(s_c, s_a) \in \mathcal{R}_\alpha$
– for each pair $(s, t) \in \mathcal{R}_\alpha$, and all $A_c \in \mathcal{A}_c$ that:
  $s \xrightarrow{A_c}_c s'$ guarantees a transition $t \xrightarrow{A_a}_a t'$ with
  - $(s', t') \in \mathcal{R}_\alpha$ and
  - $\alpha \circ [\![A_c]\!]_c \sqsubseteq [\![A_a]\!]_a \circ \alpha$

It should be noted that the union of all simulation relations, $\preceq_\alpha$, is again a simulation relation. Indeed, the largest possible simulation satisfying the above relation is exactly this union and is also the greatest-fixed point of the functional defined by the recursive definition [1, 16, 36, 35]. Thus a program model $\mathcal{P}_a$ *globally simulates* a program model $\mathcal{P}_c$, iff $(s_c, s_a) \in \preceq_\alpha$, which we will in future write like $s_c \preceq_\alpha s_a$.

As promised in the previous section, we use the preservation of program state property, $\alpha \circ [\![A_c]\!]_c \sqsubseteq [\![A_a]\!]_a \circ \alpha$, to assert that concrete actions, A_c, are safely abstracted by abstract actions, A_a.

Our use of simulation to relate program models is a continuation of the machinery used in the earlier sections: Recall that a simulation relation, *safe* , can be used in safety proofs if it is G-closed, and *safe* defines a Galois connection if it is UG-closed. Although we do not prove it here, an α-simulation, $\preceq_\alpha$, is U-closed, when one orders the set of abstract program models so that $s_{a1} \sqsubseteq s_{a2}$ when every path in s_{a1} exists in s_{a2}. If we refine the ordering to take into account the partial orderings on the information on the nodes and arcs, we can prove that $\preceq_\alpha$ is UG-closed. So, Galois connection notions carry forward to the top level of our levels of abstraction.

Guaranteeing Global Safety

As mentioned already, the primary notion of correctness (safety) is based on a transformational view: the information or property computed for a certain program point must be valid whenever a program execution reaches this point! Data flow analyses or model checking typically guarantee this criterion for a fixed level of abstraction. $\alpha-$simulations provide a framework to guarantee this correctness criterion for backwards safety properties in the context of abstraction in the following sense:[11]

$$s_c \preceq_\alpha s_a \text{ implies } s_a \models_a \phi \Rightarrow s_c \models_c \phi \text{ for all backwards safety properties } \Phi.$$

Thus, under the considered circumstances, "global simulation" guarantees "local simulation".

For forward analyses we have to consider backwards simulations[12] in order to obtain a similar result. Luckily, many abstractions on program models can be used for both forwards and backwards analyses. In these cases, all safety properties of the abstract model also hold of the concrete model.

Guaranteeing the Simulation Property

Two preservation properties are of interest, one concerning the operational semantics and one concerning program models.

Abstraction of the Operational Semantics. Once one has defined simulations/Galois connections for the concrete and abstract domains and also for the concrete and abstract actions, then one can instantiate the small-step semantics rule schemes with the concrete domains/actions and one can instantiate the rule schemes with the abstract domains/actions. As a corollary, one wants the result that the concrete semantics rules are "simulated" by the abstract semantics rules. And in fact, in [43], this result is indeed proved true for state-transition operational semantics. But for the richer format of Structural Operational Semantics, one requires a precise formalization of what it means to *instantiate* the rule schemes, a task going well beyond the scope of this paper. The reader is referred to [3, 16, 45] for the relevant machinery.

These result only address (forward simulation) and therefore only guarantee the preservation of backwards analysis for safety.

Abstraction of the Program Models. Besides abstraction on the domain and action level, which are admissible as long as corresponding Galois connections exist, the typical transformation steps are *collapsing*, i.e., identification of nodes on the abstract level while collecting the transition potential, and *unfolding*, i.e., copying of subgraphs with several entries. Whereas collapsing is clearly an abstraction, unfolding looks like a concretization at first sight, and, indeed, it is only an abstraction if it goes hand in hand with an abstraction on the domain level, where the concrete domain also distinguishes the different versions of duplicated nodes. We have:

[11] For simplicity we assume here that the formulas are identical on both levels of abstraction. In general this need not be true, e.g., it is often convenient to also abstract atomic propositions of the underlying logic.

[12] These are simply simulations of a program model with reversed transitions arcs—the "op" category, if you please.

- Locally correct abstraction on the domain and action side – characterized by the equation $\alpha \circ [\![A_c]\!]_c \sqsubseteq [\![A_a]\!]_a \circ \alpha$ – together with any kind of collapsing guarantees the existence of a forwards *and* backwards simulation. Thus it supports the verification of *all* safety properties.
- Unfolding preserves forwards simulation only. In fact, it lies in the kernel of simulation, i.e. argument and result model simulate each other both ways. Thus unfolding can freely be used whenever only backwards analysis for safety is concerned.

These are only a few sufficient conditions for guaranteeing the preservation of simulation, and there is a huge potential for elaboration here and for enlarging the class of possible simulation-preserving constructions and transformations. Thus simulation provides a powerful background for establishing safety preserving abstractions.

6 Collecting Semantics

One can construct an abstract interpretation in order to extract from it a *collecting semantics*. A collecting semantics is information extracted from the nodes and paths of a computation graph. The classic, "first-order" [38] collecting semantics extracts the states from a computation graph: For a program model (or in general, a graph), g, its first-order collecting semantics has form $coll_g$: $ProgramPoint \rightarrow 2^{Val}$ and is defined

$$coll_g(p) = \{v | (p, v) \text{ is a node in } g\}$$

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

Another form of collecting semantics is "second order" or path based; it extracts paths from the model. The set of paths that go into a program point, p, is defined

$$fcoll_g(p) = \{r \mid r \text{ is a path in } g \text{ from } root(g) \text{ to some } (p, v)\}$$

and the set of paths that emanate from p is

$$bcoll_g(p) = \{r \mid r \text{ is a maximal path in } g \text{ such that } root(r) = (p, v)\}$$

The two collecting semantics are named *fcoll* and *bcoll* because they underlie the information one obtains from forwards and backwards iterative flow analyses, respectively. For example, a live-variables analysis calculates (properties of) $bcoll_g$, and an available expressions analysis calculates (properties of) $fcoll_g$.

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. (For example, a live-variables analysis makes judgements about which variables are live for program point p in paths in $bcoll_g(p)$.)

To include such judgements, Cousot and Cousot [17] suggest that a model's collecting semantics can be a set of properties expressed in some logic, $\mathcal{L}$. Given a program model, g, and a proposition, $\phi \in \mathcal{L}$, we write $\phi \in [\![g]\!]$ if ϕ holds true of g.

For judgements to be useful, we require a *weak consistency* property of $\preceq_\alpha$ and $\mathcal{L}$:

$$g_C \preceq_\alpha g_A \;\Rightarrow\; (\text{ for all } \phi \in \mathcal{L},\, \phi \in [\![g_a]\!] \Rightarrow \phi \in [\![g_C]\!]\,)$$

That is, any property possessed by an abstract model, g_A, must also hold for a corresponding concrete model, g_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 [17]; the latter is employed to justify correctness of reductions of state spaces in concurrency theory [9, 18, 32].

With this viewpoint, the extraction of collecting semantics—flow information—is in fact the extraction of properties from a program model. This suggests that there is little difference between the collection of data-flow information and the validation of logical safety and liveness properties from a program model.

Since validation of properties of finite-state models can be done with a model checker, model checking becomes the natural tool for computing collecting semantics,[13] where collecting semantics is encoded as logical properties, as suggested above. In the next section, we employ a model checker to calculate collecting semantics encoded as properties in CTL.

7 An Analysis Methodology

The previous sections have introduced the necessary techniques for construction, abstraction, and analysis of program models. When the techniques are organized into a methodology for program interpretation and static analysis, the following three stage approach to abstraction appears, a framework that we have used in practice:

[13] In fact, iterative model checkers are directly computing a collecting semantics in this sense.

Three Stages of Abstraction

---

source program + abstraction of input data

$\Big\downarrow$ generate abstract computation graph

finite program model:
arcs labelled by program actions,
nodes labelled by abstract states

$\Big\downarrow$ replace labels by abstractions

finite program model:
arcs labelled by abstract actions,
nodes labelled by abstract states

$\Big\downarrow$ extract collecting semantics by
checking the model with formulas in CTL

static analysis results

The first stage in the picture might be termed "abstraction of state," because it is typical that a program's input data and store are abstracted to "entities of observation" when generating the finite program model. The abstraction might be a trivial one, where both input data and program store are abstracted to *nil*, or it might be nontrivial, where input data are abstracted to *even*-or-*odd* and the store is a vector of variables with *even-odd* values.

The second stage in the above picture might be named "abstraction of action," because the actions (primitive statements) in the program are abstracted to a property of interest. The abstraction might be trivial, where actions are left as they are, or it might be nontrivial, where expressions and commands are abstracted to Use-Mod information, like that used for bit-vector analyses. The result of the second stage is a finite program model whose paths show the order in which properties of actions might occur.

With this perspective, we note that the purpose of the first two stages in the picture is a complete removal of a program's "syntactic overhead," as contained in the program's source syntax and its operational semantics rules. The result, the abstracted program model, contains the data- and control-essence of the program, which can be queried by a model checker to elicit the program's semantic properties.

If the third stage in the picture must be given a name, it would be called the abstraction of the program model itself, because one queries the program model about the information at its states (nodes) and its actions (the labels on the arcs of its paths), and one collects this information into a static analysis, which is rightly an abstraction of the program model itself.

Querying and Collecting Data Flow Information

As suggested by the previous section and Section 3, the queries one makes of the program model can be given in a temporal logic such as CTL, or better in the version of CTL with the parameterized Henceforth operator considered here.[14] For example, one can assemble a complete summary of dead-variable information[15] by asking the following question, for each state (node) of a program model whose arcs are labelled with Use-Mod information

$$isDead_x = \mathbf{AG}_{\{a \mid a \neq use_x\}}(\neg end \wedge \langle\, mod_x \,\rangle\, ff)$$

for each variable, `x`, in the source program. The answers to all these queries are collected together; they constitute a dead-variables analysis.

CTL is a good language for stating queries about a program model, because it is a language for expressing patterns of information that one encounters when one traverses the paths of a model. Indeed, most static analyses are defined to collect information about the patterns of states and actions one encounters along a program's execution paths.

For example, dead-variables analysis is a static analysis that calculates, for each program point, the paths that connect a program point, p, to uses of variables. (If a variable, `x`, is not used at the end of any such path from p, it is "dead" at p.) When a bit-vector-based dead-variables analysis does a similar calculation, it attaches a bit vector, one bit per program variable, to each program point, p. The bit for variable `x` at p is set "off" if there exists a path from p to a use of `x`, which makes the variable live. Thus, the bits in a bit-vector encode a set of yes-no answers to queries about the state of variables along program paths, and a variable is dead at p if its bit maintains its initial value true.

A similar story can be told for other bit-vector-based, iterative data-flow analyses—the bit vectors encode the answers to queries about the paths leading out of (or the paths leading into) the program points in a program model.

In this way, bit-vector analyses are neatly described by our framework. But the framework can do more than handle just bit-vector-based data-flow analyses.

Beyond Bitvector Analyses

1. At the first abstraction stage, abstraction of state, one can use nontrivial abstractions—even-odd properties, sign properties, finite-ranges-of-values properties, constant-value properties—to generate program models more precise than just a control-flow graph. The "precision" in this case might mean that the program model is smaller than the control-flow graph, which may

[14] We will simply write CTL also for this parameterized version in the rest of this paper.

[15] This information is dual to the live-variable information.

be the case when the additional information suffices to evaluate conditionals, or that the program points are "split" (duplicated with different store values at different nodes), producing what the partial-evaluation community calls a *polyvariant analysis* [26]. The same idea also underlies the property-oriented expansion proposed in [49].
In addition, a program's input data can be restricted to reflect a precondition. For example, the first program model in Figure 6 shows the precision one gains when one knows from the outset that input variable `x` must be restricted to an odd number.
2. At the second abstraction stage, abstraction of actions, one can do abstraction of actions to other forms of information besides Use-Mod information. A standard example of action abstraction is seen in the behaviour trees generated by CCS-expressions [35], where actions are abstracted to "channels." Also, one might abstract all actions to *nil*; which would be appropriate for calculations of "first-order collecting semantics," as suggested in Section 6.
3. At the third stage, the extraction of collecting semantics, one might use one of a variety of logics to ask questions of the program model. Alternatives to CTL, such as CTL*, LTL, mu-calculus, and Büchi automata, can also be used this way, and the next section gives examples.
Finally, one can use the program logic to validate safety properties about paths in the program model. That is, the CTL formula we use to extract collecting semantics is also a logical proposition stating a "safety property" of a path. And of course, program validation of real safety properties can be performed in the very same framework that we have promoted for data-flow analysis. Of course, the dual is well known: data-flow analysis has been used for program validation [19, 33, 34, 39, 40], but the point has not been made as strongly as here, where we use the *very same logic* for both flow analysis and program validation.

Another benefit from the framework presented here is the clarification it gives to the stages of a static analysis—one can, at least conceptually, build a program model *first* and extract static information from it *second*.[16] Often, these two stages are intertwined in presentations of static analyses, making correctness proofs harder to write and extensions harder to implement.

8 Discussion

We now discuss several applications where our methodology led to improved solutions of static analysis problems.

State Explosion

Consider property validation for large programs or for systems of communicating processes: If one generates a program model for a system of processes, the model's

[16] This does of course not exclude an 'on-the-fly' implementation, where the model is only constructed on demand.

state space quickly becomes intractable. One solution is to construct a naive program graph model, where many—perhaps, too many—states are merged, and the price one pays is that the merged states generate many new paths that are impossible in practice. Such impossible paths thwart validation of elementary safety properties.

State-space explosion arises in model construction for single sequential programs, as well. To limit state space, a compiler builds a naive program model—the program's control-flow graph—which contains many more paths than what will actually be used during execution. The extra paths might well prevent validation of safety properties. Here is a contrived example: the control-flow graph for the program, `if even(x) then x:=succ x fi; if odd(x) then x:=0 else x:=1 fi`, contains two impossible paths, which prevent validation that the program must terminate with `x` equals 0.

Incremental Refinement

For these reasons, one wants a mechanism that incrementally refines a naive program model, eliminating impossible paths. We illustrate here a technique based on *filters* [20].

Say that we begin with a naive program model and try to validate that Φ holds true for all paths in the model. Perhaps the validation fails, because the naive model includes impossible paths that make Φ false. We want to refine the model—not abandon it—and repeat the attempt at validation. To do so, we define an additional abstract interpretation and apply it to the naive program model. We do so by *encoding the abstract interpretation as a proposition, Ψ, and we model check the formula, $\Psi \Rightarrow \Phi$, on every path of the naive model.* The abstract interpretation, coded as Ψ, filters out, on the fly, impossible paths.

This technique is reminiscent of a standard practice in model checking, that of attaching logical preconditions to strengthen the hypothesis of a formula to be proved (for example, limiting model search to just fair paths, e.g., $\mathit{isFairPath} \Rightarrow \Phi$ [7, 8]). But what is notable here is that an abstract interpretation, rather than a logical precondition, is attached. This technique has been used with success by Dwyer and Pasareanu [21] on a variety of problems in communicating systems.

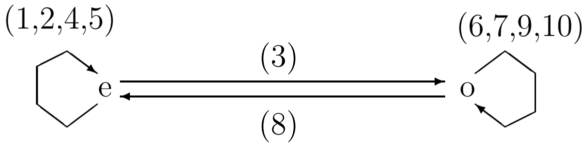
An abstract interpretation that uses a finite domain of abstract values can be encoded as a path proposition in this variant of LTL:

$$\Psi ::= \beta \mid X \mid \Psi \wedge \Psi \mid \neg\Psi \mid \langle A \rangle \Psi \mid \nu X.\, \Psi$$

Recall that LTL is logic of paths: for a path, p, say that $p \models \beta$ if state formula β holds true for p's start state; $p \models \langle A \rangle \Psi$ holds if $p = s_0 \xrightarrow{A} p_1$, and $p_1 \models \Psi$ holds (this is the "next state" modality); and $p \models \nu X.\, \Psi$ holds iff $p \models [\nu X.\, \Psi/X]\Psi$, that is, $\nu X.\, \Psi$ is a recursive proposition.

Characterizing Automata

To understand how to use LTL, we exploit the correspondence between finite-state automata and LTL, and present in Figure 8 an automaton that encodes even-odd abstract interpretation of a variable, `x`.



Automaton transitions:

(1)	$\delta(e, \mathtt{E}) = e,$ if $tt \in [\![\mathtt{E}]\!]e$	(6)	$\delta(o, \mathtt{E}) = o,$ if $tt \in [\![\mathtt{E}]\!]o$
(2)	$\delta(e, \neg\mathtt{E}) = e,$ if $f\!f \in [\![\mathtt{E}]\!]e$	(7)	$\delta(o, \neg\mathtt{E}) = o,$ if $f\!f \in [\![\mathtt{E}]\!]o$
(3)	$\delta(e, \mathtt{x} := \mathtt{E}) = o,$ if $o \in [\![\mathtt{E}]\!]e$	(8)	$\delta(o, \mathtt{x} := \mathtt{E}) = e,$ if $e \in [\![\mathtt{E}]\!]o$
(4)	$\delta(e, \mathtt{x} := \mathtt{E}) = e,$ if $e \in [\![\mathtt{E}]\!]e$	(9)	$\delta(o, \mathtt{x} := \mathtt{E}) = o,$ if $o \in [\![\mathtt{E}]\!]o$
(5)	$\delta(e, \mathtt{y} := \mathtt{E}) = e,$ if $\mathtt{y} \neq \mathtt{x}$	(10)	$\delta(o, \mathtt{y} := \mathtt{E}) = o,$ if $\mathtt{y} \neq \mathtt{x}$

Both states are possible end states.

Fig. 8. Even-odd abstract interpretation encoded as a path formula

It is easy to imagine this automaton executed on the paths of a control-flow graph—as long as the automaton can continue to move while it traverses a path, the path is well-formed with respect to the even-odd-ness value of `x`.[17] The automaton "filters out" those paths in a naive model that are impossible with respect to `x`'s even-odd-ness, and in the process generates a more accurate, restricted, reduced model. (Consider the example program at the beginning of this section—the impossible paths are filtered out.)

Next, imagine the automaton executed on the naive model *in parallel* with a model check of a safety property, Φ. This would validate Φ just on those paths well formed with respect to `x`'s even-odd-ness. No intermediate, reduced model is built; the work is done on the naive model.

The automaton in the Figure is just a pictorial representation of this LTL formula: (For brevity, we limit the set of actions to just `even x`, `¬even x`, `x:=x+1`, and `y:=2`.)

$$
\begin{array}{llll}
isEven_x \text{ iff} & \langle\, \mathtt{even\ x} \,\rangle isEven_x & \quad isOdd_x \text{ iff} & \langle\, \neg\mathtt{even\ x} \,\rangle isOdd_x \\
 & \langle\, \mathtt{x} := \mathtt{x} + 1 \,\rangle isOdd_x & & \langle\, \mathtt{x} := \mathtt{x} + 1 \,\rangle isEven_x \\
 & \langle\, \mathtt{y} := 2 \,\rangle isEven_x & & \langle\, \mathtt{y} := 2 \,\rangle isOdd_x \\
 & isFinalState & & isFinalState
\end{array}
$$

Thus, the parallel execution of even-odd analysis with the model checking of safety property Φ is just the model check of the formula

$$(isEven_x \Rightarrow \Phi) \wedge (isOdd_x \Rightarrow \Phi)$$

[17] In fact, on simply traverses the synchonous product between the program model, which can be itself regarded a automaton, and the characterizing automaton [15, 24].

Binding Time in Program Analysis

The method proposed above is based on the observation that program analysis can be refined by verifying implications on the logical level, which restricts the conditions under which a certain property must hold, or, equivalently, by verifying the original property for a product program model consisting of the original program model and a 'filter' automaton which restricts the behaviour of the program model to its desired fragment. This is possible, because the formulae considered as premises can be fully expressed in terms of automata. Based on this idea, it is now possible to choose

– where to put the complexity: in the formula or in the automaton,
– how to combine the product construction and the model checking.

Straightforward would be to first compute the product program model and subsequently apply the model checker, as suggested in the previous section. This would directly corresponds to the property-oriented expansion approach proposed in [49], which considers product construction with automata specified in different paradigms according to the unifying model idea presented in [50].

However, as also mentioned in the previous section, there are 'on-the-fly' model checkers allowing to construct the product program model on demand during the model checking procedure, which may help to avoid the state explosion problem. In fact, the same method can also be applied if one wants to verify a parallel program.

In this way, the line between abstract interpretation in model construction and property extraction via model checking has been blurred. The issue becomes one of "binding times"—when is the abstract interpretation "bound" into the program model? The answer often hinges on the desired complexity of the model versus the complexity of the formula to be model checked.

9 Conclusion

This paper has attempted to explain how abstract interpretation, flow analysis, and model checking can interact within a comprehensive static analysis methodology. In particular, we hoped to emphasize that the machinery and methods of flow analysis, abstract interpretation, and model checking have grown together, and that researchers can profitably use techniques from one area to improve results in the others.

In the future, it is planned to specifically support the study of method interaction, in order to improve the understanding of the interdependencies between and the 'optimal' problem-specific combinations of algorithms for e.g. model construction, abstraction and analysis as addressed here. A corresponding public platform is already available [46], and it is going to be used in program analysis [51].

Acknowledgements

We are grateful to Tiziana Margaria for proof reading, to Markus Müller-Olm for his constructive criticism, and to Andreas Holzmann for his support in the type setting.

References

1. P. Aczel. *Non-Well-Founded Sets*, Lecture Notes 14, Center for Study of Language and Information, Stanford, CA, 1988.
2. S. Bensalem and A. Bouajjani and C. Loiseaux and J. Sifakis. *Property preserving simulations.* Computer Aided Verification: CAV'92. Lecture Notes in Computer Science 663, Springer, 1992, 260–273.
3. D. Berry. *Generating Program Animators from Programming Language Semantics*, Ph.D. Thesis, LFCS Report ECS-LFCS-91-148, University of Edinburgh, 1991.
4. O. Burkart and B. Steffen. *Model Checking for Context-Free Processes.* Proceedings of the International Conference on Concurrency Theory, Concur95, LNCS 630, 1992
5. S. C. Cheung and J. Kramer. *An Integrated Method For Effective Behaviour Analysis of Distributed Systems.* Proceedings of the 16th International Conference on Software Engineering, Sorrento, CA, USA, 1994, pp. 309–320.
6. S. C. Cheung and J. Kramer. *Tractable Flow Analysis for Distributed Systems.* IEEE Transactions on Software Engineering 20-9 (1994).
7. E. Clarke and E. Emerson and A. Sistla. *Automatic verification of finite-state concurrent systems using temporal logic specifications.* ACM Transactions on Programming Languages and Systems 8 (1986) 244–263.
8. E.M. Clarke and O. Grumberg and D.E. Long. *Verification tools for finite-state concurrent systems.* In A Decade of Concurrency: Reflections and Perspectives, J.W. deBakker and W.-P. deRoever and G. Rozenberg", editors, Springer LNCS 803, 1993, pp. 124-175.
9. R. Cleaveland and P. Iyer and D. Yankelevich. *Optimality in abstractions of model checking.* Proc. SAS'95: Proc. 2d. Static Analysis Symposium, Lecture Notes in Computer Science 983, Springer, 1995, 1995.
10. R. Cleaveland, M. Klein and B. Steffen. *Faster Model Checking for the Modal μ-Calculus.* Proceedings of the International Workshop on Computer Aided Verification, CAV'92, LNCS 663, 1992
11. M. Codish and S. Debray and R. Giacobazzi. *Compositional analysis of modular logic programs.* Proc. 20th ACM Symp. on Principles of Programming Languages, 1993, pp. 451-464.
12. M. Codish and M. Falaschi and K. Marriott, *Suspension analysis for concurrent logic programs.* Proc. 8th Int'l. Conf. on Logic Programming, MIT Press, 1991, pp. 331-345.
13. G. Cousineau and M. Nivat. *On rational expressions representing infinite rational trees.* Proc. 8th Conf. Math. Foundations of Computer Science: MFCS'79, Lecture Notes in Computer Science 74, Springer, 1979, pp. 567–580.
14. P. Cousot, R. Cousot. *Abstract interpretation: A unified Lattice Model for static Analysis of Programs by Construction or Approximation of Fixpoints.* In Proceedings 4th ACM Symp. on Principles of Programming Languages, POPL'77, Los Angeles, California, January, 1977

15. P. Cousot and R. Cousot. *Systematic design of program analysis frameworks.* Proc. 6th ACM Symp. on Principles of Programming Languages, POPL'79, 1979, pages 269-282.
16. P. Cousot and R. Cousot. *Inductive Definitions, Semantics, and Abstract Interpretation.* Proc. 19th ACM Symp. on Principles of Programming Languages, POPL'92, 1992, pp. 83–94.
17. P. Cousot and R. Cousot. *Abstract interpretation frameworks.* Journal of Logic and Computation 2 (1992) 511-547.
18. D. Dams. *Abstract interpretation and partition refinement for model checking.* Ph.D. thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.
19. M. Dwyer and L. Clark. *Data Flow Analysis for Verifying Properties of Concurrent Programs.* Proc. 2d ACM SIGSOFT Symposium on Foundations of Software Engineering, 1994, pp.62-75.
20. M. Dwyer and D. Schmidt, *Limiting State Explosion with Filter-Based Refinement.* Proc. International Workshop on Verification, Model Checking and Abstract Interpretation, Port Jefferson, Long Island, N.Y., `http://www.cis.ksu.edu/~schmidt/papers/filter.ps.Z`, 1997.
21. M. Dwyer and C. Pasareanu. *Filter-based Model Checking of Partial Systems.* Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Orlando, FL, USA, 1998.
22. E. Emerson, J. Lei, *Efficient model checking in fragments of the propositional mu-calculus.* In Proceedings LICS'86, 267-278, 1986
23. F. Giannotti and D. Latella, *Gate splitting in LOTOS specifications using abstract interpretation.* In Proc. TAPSOFT'93, M.-C. Gaudel and J.-P. Jouannaud, eds. LNCS 668, Springer, 1993, pp. 437-452.
24. Godefroid, P. and Wolper, P. *Using Partial orders for the efficient verification of deadlock freedom and safety properties.* Proc. of the Third Workshop on Computer Aided Verification, Springer-Verlag, LNCS 575, 1991, pp. 417–428.
25. M. Hecht, *Flow Analysis of Computer Programs.* Elsevier, 1977
26. N.D. Jones and C. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
27. J. Kam and J. Ullman. *Global data flow analysis and iterative algorithms.* J.ournal of the ACM 23 (1976) 158–171.
28. G. A. Kildall. *A unified approach to global program optimization.* In *Conf. Rec. 1st ACM Symposium on Principles of Programming Languages* (*POPL'73*), pages 194 – 206. ACM, New York, 1973.
29. J. Knoop, B. Steffen and J. Vollmer *Parallelism for Free: Bitvector Analysis $\Rightarrow$ No State explosion!* Proceedings of the International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'95, LNCS 1019, 1995
30. J. Knoop, O. Rüthing and B. Steffen. *Lazy Code Motion.* Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94), Olando, Florida, SIPLAN Notices 30, 6 (1994), 233 - 245.
31. D. Kozen, *Results on the propositional mu-calculus.* Theoretical Computer Science, 27 (1983) 333–354.
32. Y.S. Kwong, *On reduction of asynchronous systems.* Theoretical Computer Science 5 (1977) 25–50.
33. S.P. Masticola and B.G. Ryder. *Static Infinite Wait Anomaly Detection in Polynomial Time.* Proceedings of ACM International Conference on Parallel Processing, 1990.

34. S.P. Masticola and B.G. Ryder. *A Model of Ada Programs for Static Deadlock Detection in Polynomial Time.* Proceedings ACM Workshop on Parallel and Distributed Debugging, 1991.
35. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
36. R. Milner and M. Tofte. *Co-induction in relational semantics.* Theoretical Computer Science, 17 (1992) 209-220.
37. A. Mycroft and N.D. Jones. *A relational framework for abstract interpretation.* In *Programs as Data Objects*, Lecture Notes in Computer Science 217, Springer, 1985, pp. 156–171.
38. F. Nielson, *A Denotational Framework for Data Flow Analysis.* Acta Informatica 18 (1982) 265-287.
39. K.M. Olender and L.J. Osterweil. *Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation.* IEEE Transactions on Software Engineering 16-3 (1990) 268–280.
40. K.M. Olender and L.J. Osterweil. *Interprocedural Static Analysis of Sequencing Constraints.* ACM Transactions on Software Engineering and Methodology 1-1 (1992) 21–52.
41. Gordon D. Plotkin. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, University of Aarhus, Denmark, 1981.
42. D.A. Schmidt, *Abstract interpretation of small-step semantics.* Proc. 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, M. Dam and F. Orava, eds. Springer, 1996.
43. D.A. Schmidt, *Trace-based abstract interpretation of operational semantics. J. Lisp and Symbolic Computation*, 10 (1998) 237-271.
44. D.A. Schmidt, *Data-flow analysis is model checking of abstract interpretations.* Proc. 25th ACM Symp. on Principles of Prog. Languages, POPL98, 1998.
45. F. daSilva. *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics.* Ph.D. thesis, LFCS report ECS-LFCS-92-241, Edinburgh University, Scotland, 1992.
46. B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration platform: concepts and design*, [51] 1(1), pp. 9-30.
47. B. Steffen. *Data Flow Analysis as Model Checking.* Proceedings of the International Concerence on Theoretical Aspects of Computer Software, TACS'91, LNCS 526, 1991
48. B. Steffen. *Generating Data Flow Analysis Algorithms from Modal Specifications*, International Journal on Science of Computer Programming, N. 21, 1993, pp. 115-139.
49. B. Steffen, *Property-oriented expansion.* Proc. Static Analysis Symposium: SAS'96, Lecture Notes in Computer Science 1145. Springer, 1996, pp. 22–41.
50. B. Steffen, *Unifying Models.* Proc. of the Annual Symposium on Theoretical Aspects of Computer Science, STACS'97, Lecture Notes in Computer Science 1200. Springer, 1997, pp. 1–20.
51. *Special Section on Programming Language Tools*, Int. Journal on *Software Tools for Technology Transfer*, Vol. 3, Springer Verlag, October 1998
52. A. Venet, *Abstract interpretation of the pi-calculus.* Proc. LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, M. Dam and F. Orava, eds., LNCS 1192, Springer, 1996.
53. A. Venet, *Automatic Determination of Communication Topologies in Mobile Systems.* Proc. SAS'98, G. Levi, ed. Springer LNCS, 1998.