Limiting State Explosion with Filter-Based Refinement

Matthew B. Dwyer and David A. Schmidt
Kansas State University*

Abstract

We introduce *filters*, an abstract-interpretation variant, to incrementally refine a naively generated state space and help validate path properties of the space via model checking. Filters can be represented equivalently as *(i)* state-transition-based abstract interpretations, *(ii)* "property automata," or *(iii)* path formulas in a CTL*-variant. We give examples of filters and show their application in the FLAVERS static analysis system.

It is no accident that a compiler uses a control flow graph (*cfg*) as its "abstract interpretation of choice" for a sequential program: A program's *cfg* possesses a manageable state space, and a variety of code improvements are enacted by conducting additional flow analyses on top of the *cfg*. Of course, such analyses are abstract interpretations (*ai* s) themselves, but what is noteworthy here is that the *cfg* is used as the base structure upon which another analysis is placed: one *ai* (the flow analysis) "filters" the other (the *cfg*).

Unfortunately, this methodology proves less successful for concurrent systems, because the construction of the *cfg* for a concurrent system, $P_1 \mid\mid P_2 \mid\mid \cdots \mid\mid P_n$, from the *cfg* s for the component processes, P_i, $i \in 1..n$, proves impractical because of state explosion due to interleavings of execution steps of the P_is.

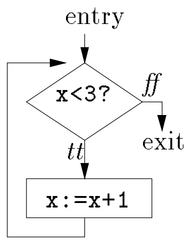
In this paper, we address the state-explosion problem for concurrent systems by applying the "filtering" idea, mentioned above, from sequential programming: We begin our analysis of a concurrent system, $P_1 \mid\mid P_2 \mid\mid \cdots \mid\mid P_n$, by defining the system's *naive cfg* , which consists of the union of the *cfg* s for the component P_i's, plus extra transitions, $\sigma_i \rightarrow \sigma_j$, for all states, σ_i in P_i's *cfg* and σ_j in P_j's *cfg* , for all $i, j \in 1..n, i \neq j$. Although it is easy to build and it is safe, the naive *cfg* contains many infeasible computation paths, and to validate rudimentary properties upon it one must apply additional *ai* s to the naive *cfg* to "filter out" infeasible computation paths.

An *ai* that is applied on top of the naive *cfg* is called a *filter*. A filter can be represented equivalently as *(i)* a state-transition-based *ai*, or *(ii)* a "property automaton" whose transitions are labelled by statements from the P_i's, or *(iii)* a path formula coded in a variant of CTL* [2]. We present all three forms in this paper.

If one wishes to validate a path formula, ϕ, for all actual executions of the concurrent system, one does so as follows:

1. Generate the naive *cfg*

*Computing and Information Sciences Department, 234 Nichols Hall, Manhattan, KS 66506 USA. schmidt@cis.ksu.edu. Supported by NSF CCR-9633388 and CCR-9703094.



Concrete interpretation:
$Store = Nat$

$n \xrightarrow{\texttt{x<3}} n$, if $n < 3$

$n \xrightarrow{\neg(\texttt{x<3})} n$, if $n \geq 3$

$n \xrightarrow{\texttt{x:=x+1}} n+1$

Concrete execution trace:

$\downarrow$
2
$\downarrow$ `x<3`
2
$\downarrow$ `x:=x+1`
3
$\downarrow$ $\neg$(`x<3`)
3 (exit)

Figure 1: Flowchart and concrete interpretation

2. Define one or more filters (path formulas), F_1, F_2, ..., F_m.

3. Model check the state formula $\forall((F_1 \wedge F_2 \wedge \cdots \wedge F_m) \supset \phi)$ on the naive *cfg*.

Standard choices for filters include those that encode the values and legal transitions of program variables, as illustrated in Section 2, and those filters, F_i, that encode the *cfg* for process P_i, as seen in Section 4. In the later case, the path formula, $F_i \supset \phi$, validates ϕ for those paths feasible with respect to P_i. Thus, filters provide a mechanism for selectively enforcing the structure of portions of a system's state space during validation.

The filter-based approach to validation has been implemented as part of the FLAVERS static analysis system [10] and has been used to validate a variety of safety properties for concurrent Ada83 programs [1, 10, 14]

In the remainder of this summary paper, we present concrete and abstract interpretations, control flow graphs, and filters, and we explain how to use these components in model-checking-based validation. Finally, we describe how filters are implemented in the FLAVERS system.

1 Concrete and Abstract Interpretations

For simplicity, we study flowchart programs. We define a concrete interpretation as a set of state transition rules of the form $v_1 \overset{c}{\to} v_2$, where v_1 and v_2 are stores and c is a program statement. Figure 1 gives a concrete interpretation of a one-variable program accompanied by an example execution trace of the program with an initial store of 2. We find it convenient and concise to draw an execution trace as a "DFA model" [17, 18]: Store values label the nodes and the executed statements label the transitions.

An abstract interpretation (*ai*) is derived from the concrete interpretation by replacing the *Store* set in the figure by *AbsStore*, a set of properties that approximate the elements of *Store*. Figure 2 gives an even-odd *ai* for the program in Figure 1 along with a sample abstract execution trace. Nondeterminism in the abstract execution trace appears due to loss of precision. The trace in the Figure is infinite, but the trace is a *regular tree* [6], because the store, statement pair, $e \xrightarrow{\texttt{x<3}}$, repeats in the infinite path—a backwards arc can be inserted. If *AbsStore* is finite, regularity is ensured, else *widening* [7, 15] can be employed.

The abstract transition rules in the Figure are created from the concrete transition rules via an obvious homomorphism criterion [4, 7, 15, 16]. Regardless of the criterion

Abstract interpretation:

$AbsStore = \{e, o\}$

$a \xrightarrow{\texttt{x<3}} a$
for $a \in AbsStore$

$a \xrightarrow{\neg(\texttt{x<3})} a$
for $a \in AbsStore$

$e \xrightarrow{\texttt{x:=x+1}} o$

$o \xrightarrow{\texttt{x:=x+1}} e$

Abstract execution trace:

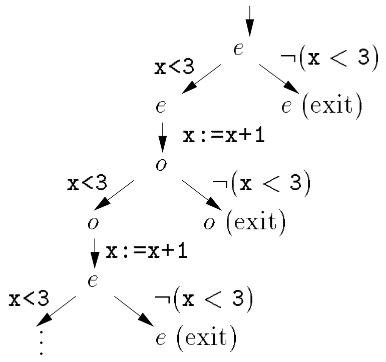
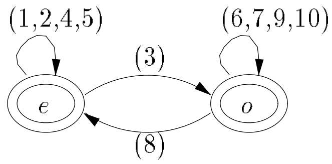


Figure 2: Abstract interpretation of flowchart



Automaton transitions:

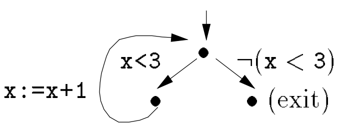
(1) $\delta(e, \texttt{E}) = e$, if $tt \in [\![\texttt{E}]\!]e$
(2) $\delta(e, \neg\texttt{E}) = e$, if $f\!f \in [\![\texttt{E}]\!]e$
(3) $\delta(e, \texttt{x} := \texttt{E}) = o$, if $o \in [\![\texttt{E}]\!]e$
(4) $\delta(e, \texttt{x} := \texttt{E}) = e$, if $e \in [\![\texttt{E}]\!]e$
(5) $\delta(e,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

Figure 3: Abstract intepretation of x as an automaton

applied, we assume there exists a concretization map, $\gamma_{AbsStore} : AbsStore \rightarrow \mathcal{P}(Store)$, which defines the set of concrete stores represented by an abstract one. (In the example, $\gamma_{AbsStore}(e) = \{2m \mid m \in Store\}$ and $\gamma_{AbsStore}(o) = \{2m+1 \mid m \in Store\}$.)

If the *AbsStore* set is finite, then an *ai* can be represented by a finite-state automaton, where the states are the elements of *AbsStore* and the transitions are the statement forms. Figure 3 gives the representation of even-odd analysis upon a single variable, x. ($[\![\texttt{E}]\!]a$ represents the denotation of E when x$=a$.) The automaton in the Figure, plus a program's flowchart, lets one generate abstract execution traces.

Perhaps the simplest example of an *ai* is the control-flow-graph interpretation, where $AbsStore = \{\bullet\}$. Then, all abstract state transitions take the form, $\bullet \xrightarrow{\texttt{c}} \bullet$—a program's abstract execution trace is just its control-flow graph (*cfg*). The *cfg* for the earlier example is of course



If the abstract intepretation has been defined properly, then one obtains the safety

```
if even x
  then α else β;
if even 2x
  then β else α;
```

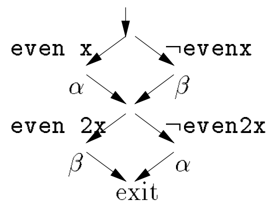
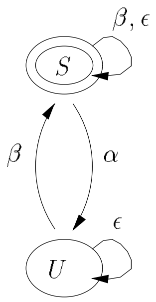


Figure 4: Example program

Say that $\epsilon \neq \alpha$ and $\epsilon \neq \beta$:

$AbsStore = \{S, U\}$
$S \xrightarrow{\alpha} U$
$a \xrightarrow{\beta} S$
for $a \in AbsStore$
$a \xrightarrow{\epsilon} a$
for $a \in AbsStore$

(a)



(b)

$$\begin{aligned} is.S \;\; &\textit{iff}\; \mathbf{X}_\alpha is.U \\ &\vee \mathbf{X}_\beta is.S \\ &\vee \mathbf{X}_\epsilon is.S \\ &\vee isExit \\ is.U \;\; &\textit{iff}\; \mathbf{X}_\beta is.S \vee \mathbf{X}_\epsilon is.U \end{aligned}$$

(c)

Figure 5: Representations of stability property

property that every abstract execution trace safely simulates a set of corresponding concrete execution traces [4, 8, 9, 15]. Regardless of how this property is formally defined, we assume that the following concretization property is maintained: for $P \in Program$, $d \in Store$, and their concrete execution trace, $t_C(P, d)$, we have that $t_C(P, d) \in \gamma_{AbsTrace}(t_A(P, a))$, where $a \in AbsStore$, $d \in \gamma_{AbsStore}(a)$, and $t_A(P, a)$ is the abstract execution trace.

2 Filters

One hopes to validate a program property, ϕ, with a simple-minded *ai*. If the initial attempt at validation fails, one hopes to improve the precision of the analysis without discarding work already accomplished. This is done by "overlaying" one *ai* upon another; we call the overlaid *ai* a *filter*.

Here is a small example with filters. For the program in Figure 4, we hope to validate that every α-statement is followed by a β-statement before another α is executed. To formalize this, say that the program starts in a "stable" state, S, but moves to an "unstable" state, U, when an α-statement is executed. The program returns to state S once a β is executed, provided that no α intervenes.

The desired "stability property" can be coded in any one of three equivalent ways:

- as an abstract interpretation (see Figure 5(a)); call the *ai* "stability analysis."

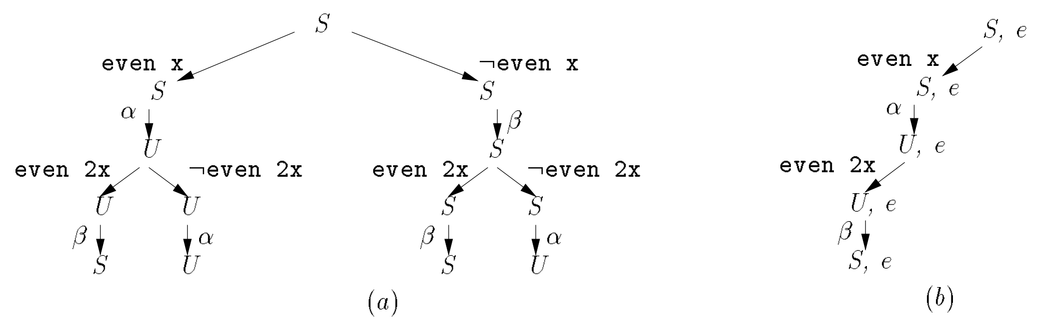


Figure 6: Abstract execution trace and its filtered version

- as an automaton (see Figure 5(b)), which we call a *property automaton*;
- as a path formula in a CTL* variant (see Figure 5(c)). (We formalize the syntax for path formulas in the next section.)

Each representation can be used to generate an abstract execution trace for the program, which appears in Figure 6(a). The trace shows that not all execution paths terminate in stable states—the attempt to validate the property has failed.

Next, say that we apply an even-odd analysis for variable `x` to the program, and rather than starting from scratch we apply the analysis to the stability-analysis trace we just generated. Assuming that `x` starts as an even number, the analysis appears in Figure 6(b) and it shows that those traces that start with an even number for `x` satisfy the stability property. We have used even-odd analysis as a *filter* upon stability analysis—irrelevant paths are "filtered out."

As seen in Figure 3, even-odd analysis can be coded as a property automaton, and we shall see shortly that every such automaton can be coded as a path formula. Let $x.is.e$ be the path formula encoding even-odd analysis when `x` is initialized to be an even number. Then, the filtering in Figure 6(b) is the path discovered by the model check of the proposition $x.is.e$ upon the paths of the trace in Figure 6(a).

But this can be taken one step further: If we assume that the program starts in a stable state, S, then we can formulate a path predicate, $is.S$ (which asserts that a path starts in state S and satisfies the stability criterion), and model check this assertion:

$$s_0 \models \forall(x.is.e \supset is.S)$$

where s_0 is the start state of the program's control-flow graph. This assertion checks whether all paths starting at node s_0 with an even value for `x` satisfy the stability property—the implication's antecedent "filters" its consequent. In this way, we adapt Steffen's encoding of flow analyses as model checks of *cfg* s to validate path predicates [17, 18, 19]. As we see in the next section, the translation of an *ai* into a path predicate encodes a precondition semantics of the *ai* .

Our use of filters as antecedents of implications is reminiscent of the encoding of path-fairness predicates into a model check of ϕ, *e.g.*, *is.fair.path* $\supset \phi$ [2, 3].

$$\begin{array}{ll} \phi \in \text{PathProposition} & Z \in \text{Iden} \\ q \in \text{PrimitiveStateProp} & \alpha \in \text{Label} \\ \pi \in \text{Path} & s \in \text{State} \end{array}$$

$$\phi \;::=\; q \mid \neg q \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \mathbf{X}_\alpha \phi \mid \nu Z.\phi \mid Z$$

$$\begin{array}{l}
\pi \models q \text{ iff } q \text{ holds true of } s \text{ and } \pi_1 = s \\
\pi \not\models q \text{ iff not } \pi \models q \\
\pi \models \phi_1 \wedge \phi_2 \text{ iff } \pi \models \phi_1 \text{ and } \pi \models \phi_2 \\
\pi \models \phi_1 \vee \phi_2 \text{ iff } \pi \models \phi_1 \text{ or } \pi \models \phi_2 \\
\pi \models \mathbf{X}_\alpha \phi \text{ iff } \pi_2 \models \phi \text{ and } \pi = s \xrightarrow{\alpha} \pi_2 \\
s \models \nu Z.\phi \text{ iff } \text{ for all } i \geq 0, s \models \phi_i, \text{ where } \begin{cases} \phi_0 = \mathit{true} \\ \phi_{i+1} = [\phi_i/Z]\phi \end{cases}
\end{array}$$

Figure 7: Syntax and semantics of path propositions

3 Filters are Path Formulas

It is easy to translate a property automaton into a path formula. Figure 7 gives the syntax and semantics of the language of path formulas we use. (We assume finite-state models.) For brevity, only the greatest fixed-point operator is included.

The recipe for translating a property automaton into a path formula goes as follows:

- Each transition, $a \xrightarrow{c} a'$ defines the clause, $is.a$ *if* $\mathbf{X}_c is.a'$.
- Each final state, a', defines the clause, $is.a'$ *if* $isExit$

The individual clauses, $is.a$ *if* ϕ_i, $i \in 1..n$, for a proposition, $is.a$, are combined by disjunctions into $is.a$ *iff* $\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$, and recursions are interpreted by the ν-operator, e.g., $is.a = \nu Z.[Z/is.a]\phi$. The path formulas for the stability analysis defined in Figure 5 are encoded as follows:

$$\begin{array}{l} is.S \;\; \mathit{iff} \;\; \mathbf{X}_\alpha is.U \vee \mathbf{X}_\beta is.S \vee \mathbf{X}_\delta is.S \vee isExit \\ is.U \;\; \mathit{iff} \;\; \mathbf{X}_\beta is.S \vee \mathbf{X}_\delta is.U \end{array}$$

The formulas define a precondition semantics in the sense that $\pi \models is.S$ holds if the starting state of path π is in the stable state, S, and π satisfies the stability criterion. For example, $\pi_0 = I \xrightarrow{\beta} exit$ satisfies $\pi_0 \models is.S \vee is.U$, because the path satisfies the stability criterion regardless of the stability of its initial state, I.

Similarly, even-odd analysis for a single variable, `x`, is encoded by this path formula; for $b \in \{e, o\}$:

$$\begin{array}{l} x.is.e \;\; \mathit{iff} \;\; (b \in [\![E]\!]e \wedge \mathbf{X}_{\mathtt{x=E}} x.is.b) \vee (\mathtt{y} \neq \mathtt{x} \wedge \mathbf{X}_{\mathtt{y=E}} x.is.e) \\ \quad \vee\, (tt \in [\![E]\!]e \wedge \mathbf{X}_{\mathtt{E?}} x.is.e) \vee (\mathit{ff} \in [\![E]\!]e \wedge \mathbf{X}_{\neg\mathtt{E?}} x.is.e) \\ \quad \vee\, isExit \end{array}$$

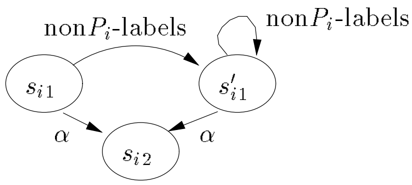
The proposition for $x.is.o$ is similar.

The claim, $\pi \models x.is.e$, holds when π is a well-formed execution path whose initial state has `x` as even-valued. It is easy to see that $\pi \models x.is.e \supset is.S$ holds true when path π, starting with `x` even-valued and in a stable state, is a well-formed execution path that satisfies the stability criterion.

4 Filters for Concurrent Systems

As noted in the Introduction, the primary obstacle to validating path properties of concurrent systems is the state explosion that arises in the generation of the system's *cfg* . We propose the *naive cfg* as a solution to the problem: For a concurrent system $P_1 \mid\mid P_2 \mid\mid \cdots \mid\mid P_n$, a program's naive *cfg* consists of the union of the *cfg* s of the P_i processes along with extra transitions, $\sigma_j \xrightarrow{\alpha} \sigma_k$, where *(i)* σ_j is a state from P_j's *cfg* ; *(ii)* σ_k is a state from P_k's *cfg* ; and *(iii)* α is a statement from P_k; for all $j, k \in 1..n$. (Note: if the concurrent system contains synchronization points between the P_i's, the complementary synchronization points for P_j and P_k are "pasted together" in the obvious way; this retricts to some degree the number of extra transitions to be added. For brevity, we do not develop synchronization; see [10] for details.)

Model checking of a state formula, $\forall\phi$, is likely to be defeated by the infeasible paths in the naive *cfg* —filters are essential. A natural and useful filter is one that encodes the legal execution paths from process P_i's perspective. Stated as a property automaton, the *process filter*, F_i, is P_i's *cfg* where each transition, $s_{i1} \overset{\alpha}{\to} s_{i2}$, is revised into this set of transitions:



The new transitions indicate that an execution path is consistent with P_i if it contains the execution of P_i's statements in their correct order interleaved with execution of arbitrarily many other statements in P_js, for $j \neq i$.

One might attempt to validate $\forall\phi$ about processes $P_1, \ldots, P_i$ by model checking the formula $\forall((F_1 \wedge F_2 \wedge \cdots \wedge F_i) \supset \phi)$. Selecting a minimal set of filters for validation is a difficult problem; in the worst case all process filters F_i, $i \in 1..n$, are required, at which point the analysis is no less costly than traditional state-space methods. In practice, however, we have found it is typically the case that only a few process filters are required for validating a given ϕ.

5 FLAVERS

The FLAVERS static analysis system uses an incremental approach to construct representations of program behavior: Analysis begins with an imprecise, conservative model (the naive *cfg*), and conservatively defined filters are applied to compare the filtered behaviors to the specification of desired behavior. This lets one begin with a low-cost analysis and incorporate only as much information as is necessary for validation.

FLAVERS is described in detail in [10]; its main components are:

- a system's *control-flow graph*

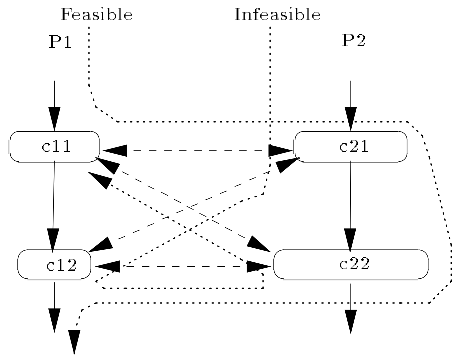


Figure 8: Naive control-flow graph

- one or more *property automata*, which define filters and properties to be validated
- a *state propagation analysis*, which implements the iterative flow analysis/model check defined by the automata.

Figure 8 shows how FLAVERS assembles two processes into a naive *cfg* . The dotted extra edges, called *may immediately precede* (*MIP*) edges, represent pairwise interleavings of program actions. The Figure shows that infeasible execution paths easily arise.

Given a *cfg* and a property automaton, ϕ, FLAVERS's state-propagation algorithm computes for each node, n, in the *cfg* the set of states in ϕ that can be reached by any path to n from the *cfg* 's start node. This is implemented as the usual monotone data flow framework [11], so the time complexity of this flow analysis is $O(|S|(|N|^2))$, where S are the states of the property automaton, and N the nodes of the *cfg*. State-propagation is an "any-path" problem and is therefore free to collapse paths that carry equivalent information, where two paths leading to a node are defined equivalent if they both result in the same state of property automaton ϕ. This equivalence-based reduction is similar to reductions in concurrency theory [5, 12].

When the validation of ϕ must be strengthened by filter F, FLAVERS implements the resulting property, $F \supset \phi$, by a state-propagation analysis on the *cfg* with the classically equivalent formula $\neg F \vee \phi$—the state propagation algorithm computes the states reached by the automata for F and ϕ, restricting validation of ϕ to just those paths that satisfy F.

The current implementation of FLAVERS automatically constructs process filters and filters for boolean and bounded-counter variables, but clearly any finite abstraction of system behavior can be implemented similarly.

FLAVERS has been applied to the analysis of concurrent Ada-83 applications. These include the traditional examples from the concurrency analysis literature [1, 10]: gas station, dining philosophers, readers-writers, simulation systems, communication protocols [14] and architectural system descriptions [13]. Analyses with FLAVERS have verified conclusively properties including mutual-exclusion, bounded-response, event-ordering, and interval behaviors.

6 Conclusion

We have attacked the state-explosion problem with an incremental approach to static analysis, where a naive abstract execution trace is filtered by subsequent abstract interpretations. The use of a filter, F, to help validate a path property, ϕ, is written $F \supset \phi$. This specification can be treated as a path formula and model checked, or it can be implemented in automata-theoretic form, as it is done in the FLAVERS system.

The incremental abstraction technique is general purpose and can be applied to any trace-based semantics of programming.

References

[1] T. Chamillard. *An Empirical Comparison of Static Concurrency Analysis Techniques.* PhD thesis, University of Massachusetts at Amherst, May 1996.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] M.B. Dwyer and L.A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. Technical Report 97-7, Kansas State University, Department of Computing and Information Sciences, 1997.

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

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

[13] G.N. Naumovich, G.A. Avrunin, L.A. Clarke, and L.J. Osterweil. Applying static analysis to software architectures. In *Proceedings of the Sixth Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 1997. to appear.

[14] G.N. Naumovich, L.A. Clarke, and L.J. Osterweil. Verification of communication protocols using data flow analysis. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1996.

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

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

[17] 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.

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

[19] 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.