Verification by Abstraction*

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (650) 859-5272 Fax: +1 (650) 859-2844

Abstract. Verification seeks to prove or refute putative properties of a given program. Deductive verification is carried out by constructing a proof that the program satisfies its specification, whereas model checking uses state exploration to find computations where the property fails. Model checking is largely automatic but is effective only for programs defined over small state spaces. Abstraction serves as a bridge between the more general deductive methods for program verification and the restricted but effective state exporation methods used in model checking. In verification by abstraction, deduction is used to construct a finite-state approximation of a program that preserves the property of interest. The resulting abstraction can be explored for offending computations through the use of model checking. We motivate the use of abstraction in verification and survey some of the recent advances.

1 Introduction

The goal of verification is to prove or refute the claim that a given program has a specified property. Deductive methods for imperative programs use program annotations to generate verification conditions that can be discharged using an automated theorem prover [Hoa69]. Model checking [CGP99] is an automatic verification technique based on explicit or symbolic state-space exploration that is applicable to finite-state programs and a limited class of infinite-state programs. The deductive approach requires annotations and manual guidance, whereas model checking does not scale well to large or infinite-state systems. Abstraction can serve as a bridge between these two methods yielding the benefit of generality, scale, and automation. A property-preserving abstraction of a program P and a property B is another program $\hat{P}$ and property $\hat{B}$ such that the verification $P \models B$ follows from $\hat{P} \models \hat{B}$. If the claim $\hat{P} \models \hat{B}$ can be verified by model

* Funded by NSF Grants CCR-0082560, DARPA/AFRL Contract F33615-00-C-3043, and NASA Contract NAS1-00079. The expert comments of Howard Barringer, Leonardo de Moura, John Rushby, Hassen Saïdi, Maria Sorea, and Tomás Uribe were helpful in the preparation of this article.

checking, and $\hat{P}$ and $\hat{B}$ can be automatically constructed from P and B, then we have a powerful verification method that does not depend on manually supplied program annotations.

Deduction can be used to automate the construction of $\hat{P}$ and $\hat{B}$ from P. The proof obligations required to construct property-preserving abstractions can usually be discharged by means of efficient decision procedures. The theorem proving involved in constructing property-preserving abstractions is also *failure tolerant*: the failure to discharge a valid proof obligation might yield a coarser abstraction but does not lead to unsoundness. The power of model checking is also enhanced through the use of abstraction to prune the size of the state space to manageable levels. Since the abstract model loses information, model checking can generate counterexamples that do not correspond to feasible behaviors at the concrete level. The concrete counterparts of abstract counterexamples can be examined using efficient decision procedures and used to refine the abstraction relation. Known properties of the concrete system can be used to sharpen the precision of the abstract model. Amir Pnueli in his invited talk[1] at CAV 2002 identified property-preserving abstraction as one of the cornerstones of a successful verification methodology. The abstraction paradigm brings about a useful synthesis of deduction and model checking where deduction is used *locally* to approximate individual formulas and transition rules, and exploration is used *globally* to calculate, for example, the reachable states on the abstract model. We briefly explain the basic ideas underlying verification by abstraction and summarize some of the recent advances.

2 Abstract Interpretation

The foundations of verification by abstraction go back to the abstract interpretation framework of Cousot and Cousot [CC77, CH78], but practical and general techniques for their use in verification are of more recent vintage. Abstract interpretation operates between a concrete partial order C and an abstract one A related by means of a *Galois connection* which is a pair (α, γ) of maps: α from C to A, and γ from A to C, such that for any $a \in A$ and $c \in C$, $\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. Intuitively, $\gamma(a)$ is the greatest concretization of a, and $\alpha(c)$ is the least abstraction for c, on the respective partial orders. Note that $c \leq_C \gamma(\alpha(c))$ and $\alpha(\gamma(a)) \leq_A a$ for any $c \in C$ and $a \in A$. The maps α and γ are order-preserving.

If $\langle C, \leq_C \rangle$ and $\langle A, \leq_A \rangle$ are complete lattices, then they admit least and greatest fixpoints of monotone operators. If F_C is a monotone operator on C, then μF_C is the least fixpoint of F in C and can be defined as $\bigcap\{X | F_C(X) \leq_C X\}$. It is possible to derive an abstract operator $\widehat{F_C} = \alpha \circ F_C \circ \gamma$ from a concrete operator F_C. It is easy to see that $\mu F_C \leq_C \gamma(\mu \widehat{F_C})$. Furthermore, if $F(a) \leq_A F'(a)$ for

[1] The slides from this talk are available at `http://www.wisdom.weizmann.ac.il/~amir/invited-talks.html`.

$\hat{+}$	0	+1	−1	$\top$
0	0	+1	−1	$\top$
+1	+1	+1	$\top$	$\top$
−1	−1	$\top$	−1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\hat{-}$	0	+1	−1	$\top$
0	0	−1	+1	$\top$
+1	+1	$\top$	+1	$\top$
−1	−1	−1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Fig. 1. Abstract versions of + and −

all $a \in A$, $\mu F \leq_A \mu F'$. In particular, if $\widehat{F_C}(a) \leq F_A(a)$ for all $a \in A$, then $\mu F_C \leq \gamma(\mu F_A)$. We write $\mu X : F[X]$ as a shorthand for $\mu(\lambda X : F[X])$. Linear-time and branching-time temporal operators from logics such as CTL, LTL, and CTL* can be expressed in terms of least and greatest fixpoints of monotone predicate transformers on the lattice of predicates or sets of states.

The approximation of concrete fixpoints by abstract fixpoints over a finite space can be used to derive concrete properties. A program over a state space Σ is given by an initialization predicate I and a binary next-state relation N over Σ. The concrete lattice C corresponding to Σ is the boolean algebra of predicates over Σ. Given such a program, the set of reachable states is characterized by $\mu X : I \vee \mathit{post}(N)(X)$, where $\mathit{post}(N)(X) = \{s' \in \Sigma | (\exists (s : \Sigma) : N(s, s')\}$. This is the strongest invariant of the program. Iteration is one way to compute the fixpoint as the limit of $I \vee \mathit{post}(N)(I) \vee \mathit{post}(N)(\mathit{post}(N)(I)) \vee \ldots$, but this computation might not terminate when the state space Σ is infinite.

As a running example, we consider a program π over a state space Σ consisting of an input integer variable x and an output integer variable y. The initialization predicate I_π is given by $y = 0$ and transition relation N_π between the current (unprimed) state and the next (primed) state is given by the formula

$$(x \geq 0 \wedge y' = y + x) \vee (x \leq 0 \wedge y' = y - x).$$

The task is to verify the invariant $y \geq 0$. It can be verified that the postcondition computation $\mu X : I_\pi \vee \mathit{post}(N_\pi)(X)$ on this transition system does not converge. We can use an abstract lattice to compute an approximation $\widehat{\mathit{post}(N_\pi)}$ to $\mathit{post}(N_\pi)$. For this purpose, we use a *sign abstraction* for the integer domain given by an abstract domain $D = \{0, +1, -1, \top\}$, where $\gamma(0)$ is the set $\{0\}$, $\gamma(+1)$ is the set of non-negative integers $[0, \infty)$, $\gamma(-1)$ is the set of non-positive integers $(-\infty, 0]$, and $\gamma(\top)$ is the set of integers $(-\infty, \infty)$. The operations + and − on integers can be lifted to the corresponding operations $\hat{+}$ and $\hat{-}$ on D as shown in Figure 1. The table of entries in Figure 1 can be precomputed using theorem proving.

The domain D is a finite lattice as is $D \times D$ where the first projection is represented by $\hat{x}$ and the second projection by $\hat{y}$, and $\gamma(\langle \hat{x}, \hat{y} \rangle)$ is just $\langle \gamma(\hat{x}), \gamma(\hat{y}) \rangle$. It is easy to see that the initialization predicate I_π is approximated by $\alpha(I_\pi) = \langle \top, 0 \rangle$. With $\sqcap$ and $\sqcup$ as the *meet* and *join* operations on the lattice D, the postcondition

operator $post(N_\pi)$ can be approximated by

$$\widehat{post(N_\pi)} = \langle\langle \hat{x}, \hat{y}\rangle \mapsto \langle \top, (\hat{y}\hat{+}(+1 \sqcap \hat{x})) \sqcup (\hat{y}\hat{-}(-1 \sqcap \hat{x}))\rangle\rangle.$$

The fixpoint $\mu\hat{X} : \alpha(I_\pi) \sqcup \widehat{post(N_\pi)}(\hat{X})$ can be calculated to yield $\langle \top, +1\rangle$. The concretization $\gamma(\langle \top, +1\rangle)$ of the abstract fixpoint yields the concrete invariant $y \geq 0$.

The abstraction can also map to an infinite domain, in which case acceleration techniques like widening and narrowing [CC77, CC92] have to be used to make the fixpoint computations converge. Techniques based on abstract interpretation and fixpoint calculations have been widely used in invariant generation [BBM97, BLS96, GS96, DLNS98, JH98, TRSS01].

3 Property-Preserving Abstraction

In the example above, the sign abstraction allowed us to approximate a fixpoint computation over a concrete lattice by one over an abstract finite lattice. In order to use model checking to explore the abstraction, we need to generate an abstract transition system that preserves the desired property of the concrete transition system. The formal use of abstraction in model checking was considered by Kurshan [Kur93]. Clarke, Grumberg, and Long [CGL92] gave a data abstraction method that preserved ∀CTL* (and hence, LTL) properties by demonstrating a simulation relation between the concrete and abstract systems. Dams, Gerth, and Grumberg [DGG94, Dam96] showed that a bisimulation relation between the abstract and concrete systems preserves CTL* properties. The relationship between abstract interpretation and the familiar simulation and bisimulation relations used to obtain property preservation, has been studied by Loiseaux, Graf, Sifakis, Bensalem, and Bouajjani [LGS+95]. Kesten and Pnueli [KP98] present a data abstraction method for fair transition systems with respect to linear-time temporal logic (LTL) properties. Approximate model checking algorithms based on abstract interpretation for mu-calculus and CTL have been given by Pardo and Hachtel [PH97].

The main challenge in using abstraction in the verification of temporal properties is that of constructing the abstract program $\hat{P}$ and property$\hat{B}$, from the concrete program P and property B. When P is itself a finite-state program given by $\langle I, N\rangle$, and the abstraction between the concrete state space Σ and the abstract state space $\hat{\Sigma}$ is given by a relation ρ, then the abstract initialization $\hat{I}$ can be constructed so that $\hat{I}(\hat{s}) = (\exists s : \rho(s, \hat{s}) \wedge I(s))$ and the transition relation $\hat{N}$ can be computed from the concrete one as $\rho^{-1} \circ N \circ \rho$. Both $\hat{I}$ and $\hat{N}$ can be represented as reduced ordered binary decision diagrams for the purposes of symbolic model checking. Such an abstraction can be shown to preserve ∀CTL* properties.

$\hat{+}$	0	+1	−1
0	$\{0\}$	$\{+1\}$	$\{-1\}$
+1	$\{+1\}$	$\{+1\}$	$\{0,-1,+1\}$
−1	$\{-1\}$	$\{0,-1,+1\}$	$\{-1\}$

$\hat{-}$	0	+1	−1
0	$\{0\}$	$\{-1\}$	$\{+1\}$
+1	$\{+1\}$	$\{0,-1,+1\}$	$\{+1\}$
−1	$\{-1\}$	$\{-1\}$	$\{0,-1,+1\}$

Fig. 2. Data Abstraction of + and −

3.1 Syntactic Data Abstraction

Data abstraction is based on assigning individual abstract values to subsets of a data domain so that a concrete variable over an infinite domain, like x in the program π above, is replaced by an abstract variable, $\hat{x}$, over a finite domain. We have already used data abstraction in the analysis of the example program P above. The Bandera program analysis tool [CDH+00], the Cadence SMV model checker [McM98], and the SCR verification tool [HKL+98] employ precomputed data abstractions to syntactically transform a transition system to a corresponding abstract one. For the case of the program π considered above, we can take the abstract data domain D to be $\{0,+1,-1\}$ where $\gamma(\{0\}) = \{0\}$, $\gamma(\{+1\}) = (0,\infty)$, $\gamma(\{-1\}) = (-\infty,0)$, and $\gamma(\hat{X} \cup \hat{Y}) = \gamma(\hat{X}) \cup \gamma(\hat{Y})$. The abstract operations $\hat{+}$ and $\hat{-}$ can then be precomputed using theorem proving as shown in Figure 2.

The program $\hat{\pi}$ can then be syntactically computed [Sha02] so that $I_{\hat{\pi}} = (\hat{y} = 0)$ and $N_{\hat{\pi}} = (\hat{x} \in \{0,+1\} \wedge \hat{y}' \in \hat{y}\hat{+}\hat{x}) \vee (\hat{x} \in \{0,-1\} \wedge \hat{y}' \in \hat{y}\hat{-}\hat{x})$. Model checking can be used to compute the set of reachable abstract states as $\mu\hat{X} : I \vee post(\hat{N})(\hat{X})$ to yield the invariant $\hat{y} \in \{0,+1\}$. The concrete counterpart of the abstract invariant is the desired concrete invariant $y \geq 0$.

3.2 Predicate Abstraction

Predicate or *boolean abstraction* is another way of reducing an infinite-state system to a finite-state one by introducing boolean variables that correspond to assertions over the (possibly infinite) state of a program. Predicate abstraction was introduced by Graf and Saïdi [SG97] as a way of computing invariants by abstract interpretation. In this form of abstraction, boolean variables $b_1, \ldots, b_n$ are used to represent concrete predicates $p_1, \ldots, p_n$. The concrete lattice consists of predicates over the concrete state space Σ and the abstract lattice consists of monomials, i.e., conjunctions over literals, b_i or $\neg b_i$, over the abstract boolean variables. The concrete reachability assertion $\mu X : I \vee post(N)(X)$ can be approximated by $\gamma(\mu X : \hat{I} \vee post(\hat{N})(X))$. Here $\gamma(\hat{p})$ for an abstract assertion $\hat{p}$

is the result of substituting the concrete assertion p_i corresponding to the abstract boolean variable b_i. The abstract initialization predicate $\hat{I}$ is computed as $\alpha(I)$, where $\alpha(p) = \bigwedge\{l_i| \vdash p \supset \gamma(l_i)\}$ where each l_i is either b_i or $\neg b_i$. Deduction is used in the construction of the $\alpha(p)$. The strongest invariant is then computed iteratively as $\mu X : \alpha(I) \vee \alpha(post(N)(\gamma(X)))$. In each step of the iteration, the partial invariant X is concretized as $\gamma(X)$, the concrete postcondition $post(N)(\gamma(X))$ is computed, and then abstracted using the definition of α given above. Since the state space of the abstract program is finite, the abstract reachability computation converges.

For the program π, the abstraction predicates are given by $\gamma(b) = (y \geq 0)$. The abstraction of the initialization predicate, $\alpha(I_\pi)$, is computed to be b. The iteration step $post(N_\pi)(y \geq 0)$ yields $y \geq 0$ and $\alpha(y \geq 0)$ is clearly b. This therefore yields the abstract invariant b as the fixpoint, and $\gamma(b)$ is the desired invariant $y \geq 0$. Graf and Saïdi further noticed that many typical abstraction predicates can be obtained from the initialization, guards, and assignments in the program, and only a few have to be introduced manually. Abstract reachability computation has also been studied by Das, Dill, and Park [DDP99] for the full boolean lattice instead of the monomial lattice.

Graf and Saïdi [SG97] gave a method for calculating the abstract transition relation in terms of monomials. We have already seen how I can be abstracted as $\alpha(I)$. The next state relation N can be abstracted as $\bigvee\{\hat{p} \wedge \hat{q}'|\hat{p} \in \mathcal{M}, \hat{q} = \alpha(post(N)(\gamma(\hat{p})))\}$, where $\mathcal{M}$ is the set of monomials over the abstract boolean variables b_i, $\hat{q}' = \bigwedge_i l_i'$ for $\hat{q} = \bigwedge_i l_i$. The weakest liberal precondition of a transition relation $\widetilde{pre}(N)(p)$ is defined as $\{s : \Sigma|(\forall s' : N(s, s') \wedge p(s'))\}$. Since $post(N)(q) \supset p \iff q \supset \widetilde{pre}(N)(p)$ (a Galois connection), we can compute $\alpha(post(N)(q))$ using $\widetilde{pre}(N)(p)$ which is more easily computed syntactically from the program. Note, however, that the abstract reachability computation $\mu X : \alpha(I) \vee \alpha(post(N)(\gamma(X)))$ can be more precise than $\mu X : \alpha(I) \vee post(\hat{N})(X)$ at the cost of requiring more calls to the theorem prover since the abstraction is computed at each iteration. A variant of the Graf-Saïdi method is used in the SLAM project at Microsoft [BMMR01] for analyzing C programs.

Other methods have been proposed for constructing boolean abstractions. In the InVest system [BLO98], Bensalem, Lakhnech, and Owre use the *elimination method* to construct the abstract transition graph so that a transition $\langle \hat{s}, \hat{s}' \rangle$ is eliminated from the abstract transition relation $\hat{N}$ if $\vdash \gamma(s) \supset \widetilde{pre}(N)(\neg\gamma(s'))$. Each abstract state s is a monomial of the form $\bigwedge_{i \in [0,n)} l_i$. Methods for constructing predicate abstractions based on the full boolean lattice have been explored by Colón and Uribe [CU98], Das, Dill, and Park [DDP99], Saïdi and Shankar [SS99], and Flanagan and Qadeer [FQ02].

The Saïdi–Shankar method has been implemented in an abstractor for PVS 2.4 that integrates both predicate and data abstraction [Sha02]. In this enumeration method, the abstract overapproximation $\alpha(p)$ of a concrete predicate p is computed as the conjunction of clauses c such that $\vdash p \supset \gamma(c)$, where each clause is a disjunction of literals l_i that are *relevant* to p. The criterion of relevance was

introduced in InVest [BLO98]. A relation $b_i \sim b_j$ holds between two abstract variables if $vars(\gamma(b_i)) \cap vars(\gamma(b_j)) \neq \emptyset$, where $vars(p)$ is the set of concrete program variables in p. The $\sim$ relation is also closed under transitivity so that it is an equivalence relation. Let $[b_i]$ represent the equivalence class of b_i under $\sim$, and $vars([b_i])$ be the set $\bigcup\{vars(b)|b \in [b_i]\}$. The boolean variable b_i is relevant to p if $vars(p) \cap vars([b_i]) \neq \emptyset$. The clauses c are generated in order of increasing length and tested with the proof goal $p \supset \gamma(c)$. If clause c passes the test, i.e., $p \supset \gamma(c)$ is provable, then any clause that is subsumed by c need not be tested. If a clause d is such that $c \wedge d$ is equivalent to a proper subclause of c then the test must have failed, and hence d also need not be tested. The transition relation N can also be overapproximated as $\alpha(N)$ over $2n$ boolean variables. Data abstraction where the target of the abstraction is an enumerated type can be smoothly integrated with this method by extending the boolean case analysis in the above enumeration to a multi-valued domain.

On the example program π, we can construct an abstraction where the single abstract variable b represents the concrete predicate $y \geq 0$. The predicate abstraction process yields b for $\alpha(I_\pi)$ and $\neg b \vee b'$ for $\alpha(N_\pi)$. Clearly, b is an invariant for the resulting abstract transition system, and hence the concrete invariant $y \geq 0$ follows. Note that unlike the method shown in Section 3.1, the abstraction here is not syntactic. Since the abstraction is not precomputed, it is possible to construct a more econonomical abstract transition system but with the cost of an exponential number of proof goals. The above enumeration method requires the testing of 3^n proof goals in the worst case, when n abstract boolean variables are introduced.

3.3 Counterexample-Guided Abstraction

Counterexample-guided abstraction is an active current research topic in abstraction. Since the abstraction is only a conservative approximation of the concrete system, the abstract version of a concrete property may fail to hold of the abstracted system. The verification then generates an abstract counterexample. The validity of this counterexample can be examined at the concrete level. If the counterexample is spurious, it is possible to refine the abstraction. Such an idea was investigated by Sipma, Uribe, and Manna [SUM96] in the context of explicit-state LTL model checking. The InVest system [BLO98] examines the failure of the concretized counterexamples to suggest new predicates. Rusu and Singerman [RS99] propose a tight integration of theorem proving and model checking where the deductive proof is used to suggest predicates and identify spurious counterexamples, and model checking on the abstraction generates useful invariants that are fed back to the theorem prover. The verification proceeds until either a proof is successfully completed or a valid counterexample is generated. Clarke, Grumberg, Jha, Lu, and Veith [CGJ+00] have also studied counterexample-guided refinement in the context of the abstraction of finite-state systems as a way of suggesting new abstraction predicates. Saïdi [Saï00] introduces new predicates to eliminate nondeterminism in the abstract model.

Das and Dill [DD01] use a partial abstraction technique that is refined through the elimination of spurious counterexamples. The intuition here is that since transition systems are typically sparse when seen as graphs, the construction of the precise abstract transition system can be wasteful. The Das–Dill method starts with a coarsely abstracted transition system with respect to a given set of abstraction predicates. Such an initial approximation can be obtained by placing some bound on the proof goals used in constructing the abstract transition relation. Model checking is used to construct a possibly spurious counterexample. A decision procedure is used to isolate the abstract transition that is concretely infeasible. The conflict set corresponding to the infeasible transition yields the minimal subset of literals that are sufficient for guaranteeing infeasibility. The negation of conjunction of the literals given by the conflict set is conjoined to the abstract transition relation to yield a more refined abstract transition relation where the spurious counterexample has been eliminated. The next round of the iteration repeats the model checking using the refined abstraction. The lazy abstraction method of Henzinger, Jhala, Majumdar, and Sutre [HJMS02] integrates the refinement and model checking procedures so that the verified parts of the state space are not re-explored in subsequent iterations.

3.4 Abstractions Preserving Liveness

Naïve abstractions are typically not useful for verifying liveness properties since the abstraction can introduce nonterminating loops that do not occur in the corresponding concrete system. Liveness properties are preserved in the sense that they do hold of the concrete system when provable of the abstract system, but the latter situation is largely hypothetical. For example, in the example program P above, we may wish to show that if the input x is infinitely often non-zero, then output y is unbounded. This property fails to hold of the abstract program even when we enrich the abstraction with the predicates $y \leq M$ for a bound M. We add a state variable r to the abstraction ranging over $\{dec, inc, same\}$ such that r is set to *dec*, *inc*, or *same* according to whether the rank $M \mathbin{\dot{-}} y$ has decreased, increased, or remained the same. Since the rank is decreased according to a well-founded order, we can introduce a fairness constraint asserting that r cannot take the value *dec* infinitely often along a path unless it also takes the value *inc* infinitely often. Abstraction techniques for progress properties based on the introduction of additional fairness constraints have been studied by Dams [Dam96], Merz [Mer97], Uribe [Uri98], Kesten and Pnueli [KP00], and by Pnueli, Xu, and Zuck [PXZ02]. In this way, abstraction yields a relatively complete verification method [KPV99] for the verification of LTL properties.

3.5 Abstracting Parameterized Systems

The verification of parametric systems consisting of n identical processes through abstraction, has been studied through the *network invariants* method

of Wolper and Lovinfosse [WL89] and Kurshan and McMillan [KM89]. Lesens and Saïdi [LS97] combine predicate abstraction with a counting abstraction to verify parameterized networks of processes. A similar 0-1-many abstraction has been studied by Pong and Dubois [PD95] and Pnueli, Xu, and Zuck [PXZ02]. The PAX system [BBLS00] captures parametric systems using the WS1S logic so that finite-state abstractions can be constructed using the MONA tool. The resulting abstractions are model checked using SPIN [Hol91] or SMV [McM93]. The PAX tool has been used to verify safety and liveness properties of examples including algorithms for mutual exclusion and group membership.

3.6 Abstracting Timed and Hybrid Systems

Timed and hybrid systems constitute natural candidates for abstraction. The region graph constructions used in model checking such systems is already a form of abstraction [ACD93]. Colón and Uribe [CU98,Uri98] carried out a verification of a two-process instance of Fischer's real time mutual exclusion protocol using predicate abstraction. Möller, Rueß, and Sorea [MRS02] present a predicate abstraction method for next-free temporal properties of timed systems that uses a restricted semantics for time-flow to ensure non-Zeno behavior in the abstraction. In this restricted semantics, each time increment must ensure that some clock either reaches or crosses an integer boundary. Namjoshi and Kurshan [NK00] give a method for systematically constructing abstraction predicates that is complete for systems with finite bisimulations.

Khanna and Tiwari [TK02] give a *qualitative* abstraction method for hybrid systems where the flows over a vector $(x_1, \ldots, x_n)$ are specified as $\dot{x_i} = f_i(x_1, \ldots, x_n)$, where $f_i(x_1, \ldots, x_n)$ is a polynomial. The initial set Π_0 of polynomials is fixed to contain the flow polynomials $f_i(x_1, \ldots, x_n)$ and the polynomials occurring in the initializations, guards and assignments. The set Π_0 is then enriched to obtain Π by adding the derivative $\dot{p}$ for each polynomial $p \in \Pi$, unless $\dot{p}$ is a constant or a constant multiple of some polynomial q already in Π. The construction of Π can be terminated at any point without affecting the soundness of the abstraction. Then a sign abstraction with respect to these derivatives is computed by introducing a variable sp ranging over $\{-, +, 0\}$ for each $p \in \Pi$. PVS augmented with the QEPCAD decision procedure [CH91] for the first-order theory of real closed fields is used to determine the concrete feasibility of abstract states. There is a transition on the abstract system from state $\hat{s}$ to $\hat{s}'$ only when the signs of the flows are consistent with the transition. For example, if sp goes from $+$ in $\hat{s}$ to 0 in $\hat{s}'$ for a polynomial p, then sq must be $-$ in $\hat{s}$ for $q = \dot{p}$. Discrete transitions are abstracted as shown earlier for transition systems. Alur, Dang and Ivančić [ADI02] present a predicate abstraction technique for linear hybrid automata.

4 Conclusions

In summary, abstraction is a powerful verification paradigm that combines deduction and model checking. The keys to the effectiveness of abstraction are that

1. Guessing useful abstraction maps is easier than identifying program annotations and invariant strengthenings.
2. It makes use of failure-tolerant theorem proving, largely in effectively decidable domains, to deliver possibly approximate results.
3. The refinement of abstractions can be guided by counterexamples.
4. Abstraction interacts effectively with other verification techniques such as invariant generation, progress verification, and refinement.

We have given a basic introduction to verification methods based on abstraction. The practicality of these abstraction techniques has been demonstrated on several large examples. Flanagan and Qadeer [FQ02] have applied predicate abstraction to a large (44KLOC) file system program and automatically derived over 90% of the loop invariants. Predicate abstraction has also been used in the SLAM project [BMMR01] at Microsoft to find bugs in device driver routines.

Abstraction is an effective approach to the verification of both finite and infinite-state systems since it can be applied to large and complex systems with minimal user guidance. The SAL (Symbolic Analysis Laboratory) framework [BGL$^+$00] developed at SRI provides a toolbus and an intermediate description language for tying together a number of verification tools through the use of property-preserving abstractions.

References

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[ADI02] Rajeev Alur, Thao Dang, and Franjo Ivančić. Reachability analysis of hybrid systems via predicate abstraction. In Tomlin and Greenstreet [TG02], pages 35–48.

[AH96] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[BBLS00] Kai Baukus, Saddek Bensalem, Yassine Lakhnech, and Karsten Stahl. Abstracting WS1S systems to verify parameterized networks. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, number 1785 in Lecture Notes in Computer Science, pages 188–203, Berlin, Germany, March 2000. Springer-Verlag.

[BBM97] Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, June 2000. NASA Langley Research Center. Proceedings available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`.

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [HV98], pages 319–331.

[BLS96] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [AH96], pages 323–335.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation, 2001*, pages 203–313. ACM Press, 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis. In *4th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, July 1992.

[CDH+00] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[CGP99] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables. In *5th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1978.

[CH91] G. E. Collins and H. Hong. Partial cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 12(3):299–328, 1991.

[CU98] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Hu and Vardi [HV98], pages 293–304.

[Dam96] Dennis René Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, July 1996.

[DD01] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Annual IEEE Symposium on Logic in Computer Science01*, pages 51–60. The Institute of Electrical and Electronics Engineers, 2001.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [HP99], pages 160–171.

[DGG94] Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving ∀CTL*, ∃CTL* and CTL*. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET '94)*, pages 561–581, 1994.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *ACM Symposium on Principles of Programming Languages02*, pages 191–202. Association for Computing Machinery, January 2002.

[Gru97] Orna Grumberg, editor. *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.

[GS96] Susanne Graf and Hassen Saïdi. Verifying invariants using theorem proving. In Alur and Henzinger [AH96], pages 196–207.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *ACM Symposium on Principles of Programming Languages02*, pages 58–70. Association for Computing Machinery, January 2002.

[HKL+98] Constance Heitmeyer, James Kirby, Jr., Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, 1969.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[HP99] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[JH98] Ralph Jeffords and Constance Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 56–69, Lake Buena Vista, FL, November 1998. Association for Computing Machinery.

[KM89] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Eighth ACM Symposium on Principles of Distributed Computing*, pages 239–248, Edmonton, Alberta, Canada, August 1989.

[KP98] Yonit Kesten and Amir Pnueli. Modularization and abstraction: The keys to practical formal verification. In *Mathematical Foundations of Computer Science*, pages 54–71, 1998.

[KP00] Yonit Kesten and Amir Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.

[KPV99] Yonit Kesten, Amir Pnueli, and Moshe Y. Vardi. Verification by augmented abstraction: The automata-theoretic view. In J. Flum and M. R. Artalejo, editors, *CSL: Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 141–156. Springer-Verlag, 1999.

[Kur93] R.P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1993.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.

[LS97] David Lesens and Hassen Saïdi. Automatic verification of parameterized networks of processes by abstraction. In Faron Moller, editor, *2nd International Workshop on Verification of Infinite State Systems: Infinity '97*, volume 9 of *Electronic Notes in Theoretical Computer Science*, Bologna, Italy, July 1997. Elsevier.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[McM98] Ken McMillan. Minimalist proof assistants: Interactions of technology and methodology in formal system level verification. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1998. Springer-Verlag. Invited presentation—no paper in proceedings, but slides available at `http://www-cad.eecs.berkeley.edu/~kenmcmil/`.

[Mer97] Stephan Merz. Rules for abstraction. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science—ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 32–45, Kathmandu, Nepal, December 1997. Springer-Verlag.

[MRS02] M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate abstraction for dense real-time systems. *Electronic Notes in Theoretical Computer Science*, 65(6), 2002. Full version available as Technical Report BRICS-RS-01-44, Department of Computer Science, University of Aarhus, Denmark.

[NK00] K. Namjoshi and R. Kurshan. Syntactic program transformations for automatic abstraction. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 435–449, Chicago, IL, July 2000. Springer-Verlag.

[PD95] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, August 1995.

[PH97] Abelardo Pardo and Gary D. Hachtel. Automatic abstraction techniques for propositional mu-calculus model checking. In Grumberg [Gru97], pages 12–23.

[PXZ02] Amir Pnueli, Jessie Xu, and Lenore Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *Computer-Aided Verification, CAV '02*, Lecture Notes in Computer Science. Springer-Verlag, July 2002.

[RS99] Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 178–192, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[Saï00] Hassen Saïdi. Model checking guided abstraction and analysis. In Jens Palsberg, editor, *Seventh International Static Analysis Symposium (SAS'00)*, volume 1824 of *Lecture Notes in Computer Science*, pages 377–396, Santa Barbara CA, June 2000. Springer-Verlag.

[SG97] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Grumberg [Gru97], pages 72–83.

[Sha02] Natarajan Shankar. Automated verification using deduction, exploration, and abstraction. In *Essays on Programming Methodology*. Springer-Verlag, 2002. To appear.

[SS99] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [HP99], pages 443–454.

[SUM96] H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. In Alur and Henzinger [AH96], pages 208–219.

[TG02] C.J. Tomlin and M.R. Greenstreet, editors. *Hybrid Systems: Computation and Control, 5th International Workshop, HSCC 2002*, volume 2289 of *Lecture Notes in Computer Science*, Stanford, CA, March 2002. Springer-Verlag.

[TK02] Ashish Tiwari and Gaurav Khanna. Series of abstractions for hybrid automata. In Tomlin and Greenstreet [TG02], pages 465–478.

[TRSS01] Ashish Tiwari, Harald Rueß, Hassen Saïdi, and N. Shankar. A technique for invariant generation. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127, Genova, Italy, April 2001. Springer-Verlag.

[Uri98] Tomás E. Uribe. *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Stanford University, 1998. Available as Stanford University Computer Science Department Technical Report No. STAN-CS-TR-99-1618.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 68–80, Grenoble, France, 1989. Springer-Verlag.