Syntactic Program Transformations for Automatic Abstraction

Kedar S. Namjoshi and Robert P. Kurshan

Bell Laboratories, Lucent Technologies
{kedar,k}@research.bell-labs.com
http://cm.bell-labs.com/cm/cs/who/{kedar,k}

Abstract. We present an algorithm that constructs a finite state "abstract" program from a given, possibly infinite state, "concrete" program by means of a *syntactic* program transformation. Starting with an initial set of predicates from a specification, the algorithm iteratively computes the predicates required for the abstraction relative to that specification. These predicates are represented by boolean variables in the abstract program. We show that the method is *sound*, in that the abstract program is always guaranteed to simulate the original. We also show that the method is *complete*, in that, if the concrete program has a finite abstraction with respect to simulation (bisimulation) equivalence, the algorithm can produce a finite simulation-equivalent (bisimulation-equivalent) abstract program. Syntactic abstraction has two key advantages: it can be applied to infinite state programs or programs with large data paths, and it permits the effective application of other reduction methods for model checking. We show that our method generalizes several known algorithms for analyzing syntactically restricted, data-insensitive programs.

1 Introduction

Model Checking [CE81,QS82] is a fully automatic method for checking that a finite state program satisfies a propositional temporal specification. It has proved to be quite useful for the analysis of concurrent hardware and software systems; there are several academic and commercial model checking tools. The main obstacle to the wider application of model checking is the exponential growth in the size of the state space with increasing program size: current tools are typically limited to programs with a few hundred boolean state variables. There are two main approaches to ameliorating this state-explosion problem: *compositional verification*, where one manually constructs a proof outline that exploits the compositional structure of the program, while model-checking sufficiently small components, and *abstraction*, where one constructs a smaller *abstract* program in a manner which ensures that the specification holds for the original program if it holds for the abstract program.

Abstraction is often carried out manually and justified only informally. For large programs, such manual abstraction is error-prone and often infeasible. Our focus in this paper, therefore, is on automating the abstraction process. We present an algorithm that constructs a *finite state* "abstract" program from a given, possibly infinite state, "concrete" program by means of a *syntactic* program transformation. The abstract program represents predicates of the concrete program by boolean variables. Starting with the atomic predicates from a specification or statement of a property that is to be verified, the algorithm iteratively computes the predicates required for the abstraction relative to that specification, as well as the necessary updates to the corresponding boolean variables. This is achieved by a syntactic analysis that does not construct the explicit transition graph either of the original or of the abstract program, each of which may be too large to compute.

We show that the algorithm is *sound*, in that the abstract program is always guaranteed to be a conservative approximation of (i.e., simulates) the original, with respect to the set of specification predicates, AP. Under certain conditions, the algorithm produces an *exact* (i.e., bisimular) abstraction of the original program. The soundness result implies that a temporal property in $ACTL^*$ over AP holds for the original program if it holds for the abstraction. For an exact abstraction, this is true for properties written in the full $\mu-$calculus. We also show that the algorithm is *complete* in the sense that, if the state transition graph of the original program has a finite simulation (bisimulation) quotient, then the algorithm can produce a finite simulation-equivalent (bisimulation-equivalent) abstract program.

Syntactic abstraction has several advantages:

- The algorithm produces an implicit (syntactic) description of the abstract program. Hence, other methods for model checking, such as symbolic (BDD-based) model checking and partial-order reduction, can be applied to the abstract program.
- It supports the abstraction of *data-insensitive* programs with large or infinite data paths. Our method generalizes several earlier algorithms for analyzing data-insensitive programs and may also be used in other cases, such as symmetric programs, where large reductions can be achieved through bisimulation minimization.
- It can often be substantially more efficient than the symbolic minimization algorithms of [BFH90,LY92,HHK95] as, unlike these methods, our algorithm does not construct the explicit transition graph of the minimized system, which could be quite large.
- For programs with bounded non-determinacy, our algorithm is able to construct abstractions without the manual application of theorem-provers, as proposed for other predicate abstraction methods (cf. [GS97,BLO98]).

The paper is structured as follows. In Section 2, we provide some background on simulation, bisimulation, temporal logic and model checking. We describe the basic algorithm in Section 3 and prove the soundness and completeness claims. In Section 4, we present some useful extensions to the basic algorithm. In Section 5, we describe several applications of the algorithm. Section 6 concludes the paper with a discussion of related work.

2 Background

We provide some background on abstraction and model checking, and present a simple program syntax on which the analyses of the algorithm are defined.

2.1 Labeled Transition Systems

The state transition graph of a program is represented by a *Transition System* (TS, for short) [Kel76] which is a tuple (S, Δ, I, AP, L), where
• S is the set of *states*,
• $\Delta \subseteq S \times S$, is the (left-total) transition relation. We write $s \longrightarrow t$ instead of $(s,t) \in \Delta$ for clarity.
• $I \subseteq S$ is the set of initial states,
• AP is the set of atomic propositions, and
• $L : S \to 2^{AP}$ is the state labeling function, which maps each state to the set of atomic propositions that hold at that state.

A *computation* σ of the TS is an infinite sequence of states such that $\sigma_0 \in I$, and for each i, $\sigma_i \longrightarrow \sigma_{i+1}$. A TS is often constrained by a fairness condition, expressed as a boolean combination of basic fairness conditions "infinitely often p", where p is a set of state pairs [1]. A *fair computation* of the TS is a computation that satisfies the fairness condition, where the basic fairness formula above is satisfied iff transitions from p appear infinitely often on the sequence.

2.2 The μ-calculus

The μ-calculus [Koz83] is a branching-time temporal logic, where formulas are built from atomic propositions, boolean connectives, the least-fixpoint operator, and the modality $\langle\rangle\phi$ ("there is a successor satisfying ϕ"). For a state s in a TS M and a μ-calculus formula ϕ over the atomic propositions of M, we write $M, s \models \phi$ to mean that "ϕ is true at state s in model M". The sub-logic $A\mu$ consists of those μ-calculus formulae where every $\langle\rangle$ operator is under an odd number of negations. It is possible [EL86] to encode a number of temporal logics, including *CTL*, *CTL** [CES86,EH86] and *LTL* [Pnu77] into the μ-calculus.

2.3 Simulation and Bisimulation

The correctness of any abstraction method is determined by the nature of the relationship between the concrete and the abstract program. A typical relationship is that the abstract program is able to match every computation of the concrete program. This is formalized in the definitions below.

Definition 0 (Simulation Relation) [Mil71] *A relation $R \subseteq S \times S$ is a simulation relation on a TS $M = (S, \Delta, I, AP, L)$ iff for any $(s,t) \in R$, $L(s) = L(t)$ and for any u such that $s \longrightarrow u$, there exists v such that $t \longrightarrow v$ and $(u,v) \in R$.*

[1] The usual unconditional, weak and strong fairness conditions can be written as boolean combinations of basic fairness conditions.

Definition 1 (Bisimulation Relation) [Par81] *A relation R is a bisimulation on TS M iff R is symmetric and a simulation relation on M.*

State t *simulates* a state s iff (s,t) is in the greatest simulation, which exists as simulations are closed under arbitrary union. States s and t are *simulation-equivalent* (written as $s \sim t$) iff they simulate each other. States s and t are *bisimular* in M (written as $s \approx t$) iff (s,t) is contained in the greatest bisimulation relation on M. The connection between model checking and these notions of program equivalence is as follows.

Theorem 0 *For a TS M and states s,t in M,*

1. (cf.[GL94]) *For any $A\mu$ formula f on AP, if t simulates s then $M,t \models f$ implies $M,s \models f$.*
2. (cf. [BCG88]) *For any μ-calculus formula f on AP, if $s \approx t$ then $M,s \models f$ iff $M,t \models f$.* $\square$

As $\sim$ ($\approx$) is an equivalence relation, it induces a *quotient* TS whose states are the equivalence classes of M w.r.t. $\sim$ ($\approx$), the initial set of states is the equivalence classes of the initial states of M, and there is a transition (C,a,D) iff $(\exists s,t : s \in C \wedge t \in D : s \xrightarrow{a} t)$.

2.4 Program Syntax

Our algorithms transform one program text to another. For our purposes, instead of specifying a particular program syntax, it suffices to consider a program as being defined on a set of variables, and being specified once an initial condition, a transition relation and a fairness constraint are defined. These are specified syntactically as *predicates*: a predicate is a quantifier-free formula of a first-order logic. The relation symbols form the *atomic* predicates. Some relation and function symbols may have fixed interpretations, for instance $=, \leq, +$.

Definition 2 (Program) *A program is specified by a tuple (X,I,T,F), where*

- *X is a finite, non-empty set of* variables*. Each variable x has an associated domain of values, $dom(x)$.*
- *$I(X)$ is the* initial condition*, specified as a predicate on X.*
- *$T(X,X')$ is the* transition relation*, specified as a predicate on $X \cup X'$, where X' is a set of "next-state" variables that is in 1-1 correspondence with X.*
- *$F(X,X')$ is the* fairness condition*, specified as a boolean combination of the basic fairness condition "infinitely-often p", for a predicate p over $X \cup X'$.*

The semantics of such a program is given by a TS defined as follows. The state space is the Cartesian product of the domains of the variables. The value of an expression e in state s is denoted by $e(s)$; this can be defined by induction on the expression syntax. An initial state is one for which the initial state predicate evaluates to *true*. The transition relation is defined as follows: there is a transition

$s \longrightarrow t$ iff $T(s,t)$ is true. The state labeling function L is defined by: for each atomic predicate P, $P \in L(s)$ iff $P(s) = \mathit{true}$. The fairness condition of the fair TS is obtained in a straightforward manner from the condition F.

We develop our algorithm under the assumption that the set of predicates is effectively closed under the application of the "weakest liberal precondition" transformer [Dij75], denoted by $\mathit{wlp}(P)$ (in terms of the μ-calculus, $\mathit{wlp}(P) = \neg\langle\rangle(\neg P)$).

Typical programming constructs can be rewritten into the program syntax presented above. For example, a guarded command [Dij75], which has the form $g(X) \hookrightarrow U := e(X)$, defines the transition relation $(g(X) \wedge (\bigwedge i : x_i \in U : x'_i = e_i(X)) \wedge (\bigwedge i : x_i \in X \backslash U : x'_i = x_i)$. For guarded commands, $\mathit{wlp}(P)$ can be calculated by simple substitution as $(g(X) \Rightarrow P[U \leftarrow e(X)])$, where $P[U \leftarrow e(X)]$ is the predicate obtained by replacing each occurrence of $x_i \in U$ by e_i in P, for all i. Programs with external, non-deterministic inputs can be defined by partitioning the set of variables X into the input variables Y, which are unconstrained, and state variables Z, whose next-state values are constrained by the transition relation. Using the syntax $[Y] : g(X) \hookrightarrow U := e(X)$ for describing such transitions, $\mathit{wlp}(P)$ is given by $(\forall Y :: g(X) \Rightarrow P[U \leftarrow e(X)])$. In this case, it may be necessary to use a quantifier-elimination procedure, such as that for Presburger arithmetic, to re-write $\mathit{wlp}(P)$ as a predicate.

3 The Abstraction Algorithm

We motivate and describe our algorithm for the simpler case of programs that exhibit *bounded* non-determinism. For such programs, the transition relation $T(X, X')$ is equivalent to a bounded disjunction $(\bigvee i :: T_i(X, X'))$, where each T_i is *deterministic*, i.e., for any state s, there is at most one state t such that $T_i(s,t)$ holds. We assume that the transition relation is given in this form and refer to each T_i as an *action*. The key property that we exploit is that for a deterministic action a, wlp_a distributes over all boolean operators [2].

We do not consider fairness conditions in this section. The extensions needed to handle fairness and unbounded nondeterminism are described in Section 4. The results established here carry over, essentially unchanged, to the general case.

3.1 A Motivating Example: the 2-process Bakery protocol

Consider the 2-process "Bakery" mutual exclusion protocol [Lam74] presented in Figure 1 as a finite collection of guarded commands. The specification of mutual exclusion in *CTL* is $\mathsf{AG}(\neg(st_1 = C \wedge st_2 = C))$. To verify this property, an abstraction of the protocol needs to preserve at least the atomic predicates $st_1 = C$ and $st_2 = C$, as well as those in the initial condition: $st_1 = N$, $st_2 = N$, $y_1 = 0$, $y_2 = 0$. We may choose to retain the variables st_1, st_2, as they have

[2] In the case of negation, $\mathit{wlp}_a(\neg P) \equiv \mathit{wlp}_a(\mathit{false}) \vee \neg \mathit{wlp}_a(P)$

var $st_1, st_2 : \{N, W, C\}$
(* N="Non-critical", W="Waiting", C="Critical" *)
var y_1, y_2 : **natural**
initially $(st_1 = N) \wedge (y_1 = 0) \wedge (st_2 = N) \wedge (y_2 = 0)$

action $wait_1$	$st_1 = N \hookrightarrow st_1, y_1 := W, y_2 + 1$
action $enter_1$	$st_1 = W \wedge (y_2 = 0 \vee y_1 \leq y_2) \hookrightarrow st_1 := C$
action $release_1$	$st_1 = C \hookrightarrow st_1, y_1 := N, 0$
action $wait_2$	$st_2 = N \hookrightarrow st_2, y_2 := W, y_1 + 1$
action $enter_2$	$st_2 = W \wedge (y_1 = 0 \vee y_2 < y_1) \hookrightarrow st_2 := C$
action $release_2$	$st_2 = C \hookrightarrow st_1, y_2 := N, 0$

Fig. 1. The 2-process Bakery mutual exclusion algorithm.

small finite domains, but we would want to abstract y_1, y_2, as they have infinite domains, retaining only those predicates on y_1, y_2 which are necessary to preserve the control flow of the protocol. We do so by introducing auxiliary boolean variables b_1 and b_2 which represent $y_1 = 0$ and $y_2 = 0$, respectively. The initial condition can now be expressed as $(st_1 = N) \wedge b_1 \wedge (st_2 = N) \wedge b_2$.

To preserve the correspondence between b_1 and $y_1 = 0$, we need to compute an update to b_1 for each action. For a deterministic action a, b_1 is true after an update to y_1 exactly if $wlp_a(y_1 = 0)$ is true before the update. For the action *wait₁*, the syntactic *wlp* calculation yields $((st_1 = N) \Rightarrow (y_2 + 1 = 0))$. Now, however, we have a new predicate $(y_2 + 1 = 0)$, which can be simplified to *false*, as y_2 is a natural number. Hence, the modified action is:

action $wait_1 \quad st_1 = N \hookrightarrow st_1, y_1, b_1 := W, y_2 + 1, (st_1 = N \Rightarrow \mathit{false})$

We may simplify this further by replacing occurrences of the guard expression by *true* in the assignment, to get:

action $wait_1 \quad st_1 = N \hookrightarrow st_1, y_1, b_1 := W, y_2 + 1, \mathit{false}$

The result of $wlp_{enter_1}(y_1 = 0)$ is $((st_1 = W \wedge (y_2 = 0 \vee y_1 \leq y_2)) \Rightarrow (y_1 = 0))$. In this case, we have a new atomic predicate, $y_1 \leq y_2$, which must therefore be tracked by a new boolean variable, b_3. Repeating the steps above, we get:

action $enter_1 \quad st_1 = W \wedge (b_2 \vee b_3) \hookrightarrow st_1, b_1 := C, b_1$

This iterative process of computing weakest preconditions and collecting new predicates terminates for our example program; i.e., after a finite number of iterations, no new predicates are generated. Hence, it suffices to consider only these predicates to verify the property. As the auxiliary boolean variables track the predicates exactly, the infinite-domain variables y_1 and y_2 are unnecessary and can be removed, resulting in the *finite-state* abstraction shown in Figure 2, which is *bisimulation-equivalent* to the original with respect to the initial predicate set $\{st_1 = C, st_2 = C\}$. From Theorem 0, the mutual exclusion property is true of the original program if and only if it is true of the abstraction.

var $st_1, st_2 : \{N, W, C\}$		
(* N="Non-critical", W="Waiting", C="Critical" *)		
var b_1, b_2, b_3 : **boolean**		
(* $b_1 = (y_1 = 0), b_2 = (y_2 = 0), b_3 = (y_1 \leq y_2)$ *)		
initially $(st_1 = N) \wedge b_1 \wedge (st_2 = N) \wedge b_2 \wedge b_3$		
action $wait_1$	$st_1 = N \hookrightarrow st_1, b_1, b_2, b_3 := W, \mathit{false}, b_2, \mathit{false}$	
action $enter_1$	$st_1 = W \wedge (b_2 \vee b_3) \hookrightarrow st_1, b_1, b_2, b_3 := C, b_1, b_2, b_3$	
action $release_1$	$st_1 = C \hookrightarrow st_1, b_1, b_2, b_3 := N, \mathit{true}, b_2, \mathit{true}$	
action $wait_2$	$st_2 = N \hookrightarrow st_2, b_1, b_2, b_3 := W, b_1, \mathit{false}, \mathit{true}$	
action $enter_2$	$st_2 = W \wedge (b_1 \vee \neg b_3) \hookrightarrow st_2, b_1, b_2, b_3 := C, b_1, b_2, b_3$	
action $release_2$	$st_2 = C \hookrightarrow st_1, b_1, b_2, b_3 := N, b_1, \mathit{true}, b_1$	

Fig. 2. Abstraction of the 2-process Bakery mutual exclusion algorithm.

3.2 The Algorithm

The Bakery example introduced the key ingredients of our algorithm: starting from the atomic predicates in the specification formula, predicates of the original program are represented by boolean variables in the abstraction, exact updates for these boolean variables are computed by a syntactic *wlp* computation, possibly introducing new predicates to be examined, and simplifications are performed to avoid introducing new predicates that are syntactically distinct but semantically identical to predicates generated earlier. The algorithm is presented in its entirety in Figure 3.

The algorithm maintains a correspondence table, C, which relates syntactic atomic predicates to corresponding boolean variables. The algorithm also maintains two sets of atomic predicates: *oldPred*, which consists of those predicates for which *wlp* has been calculated, and *newPred*, which consists of the unexamined predicates. Initially (step 1), *oldPred* is empty, and *newPred* contains the atomic predicates of the specification formula, the initial condition, and the actions. In each iteration (steps 2a-2c), *wlp* is calculated for each predicate in *newPred* and the result is massaged (described below) to extract new predicates, for which new boolean variables are introduced. In steps 3, the abstract transition relation is defined by updating each boolean variable with the expression formed by massaging *wlp* for the corresponding predicate.

Although the process of generating new predicates terminates for the Bakery example, in general, it may not terminate. To ensure termination, we iterate this process for K steps, where K is a parameter to the algorithm. After K iterations, however, the predicates in *newPred* have not been processed by a *wlp* computation. Boolean *input* variables are introduced for these predicates, which are constrained by Φ to valuations that are consistent relative to their corresponding predicates. Similarly, Ψ constrains the initial values of boolean variables corresponding to the predicates in *oldPred*. In the definition of $T_{\mathcal{A}}$, the *wlp* operator is applied only to predicates from *oldPred*, so the result is in terms of *oldPred* $\cup$ *newPred* and gets massaged into an expression on $X_{\mathcal{A}}$.

It is not necessary to add Φ and Ψ to obtain a program that is a conservative approximation; these are needed only to establish the completeness results. If the atomic predicates come from a class with quantifier elimination (such as Presburger arithmetic), Φ and Ψ can be computed syntactically. In general, the computability of Φ and Ψ is equivalent to the decidability of satisfiability for predicates – in the worst case, Φ (similarly, Ψ) can be computed by checking the predicate in the scope of the $(\exists X)$ quantifier for satisfiability for each valuation of the boolean free variables. The decidability of this satisfiability question is also assumed for the symbolic minimization algorithms [BFH90,LY92,HHK95].

The massaging step $(\mathit{massage} : (e, C) \mapsto (\overline{e}, \mathit{newC}, \mathit{fP}))$ simplifies the expression e and replaces atomic predicates by corresponding boolean variables, defining new boolean variables for predicates not already in C. These new predicates are collected in fP, and C is updated to *newC* by adding the new predicate-variable correspondences. The resulting expression is denoted by $\overline{e}$. We also use this notation to let $\overline{S}$ represent the set of boolean variables corresponding to the atomic predicates in set S. The simplifications accelerate the convergence of the algorithm and are thus necessary in practice, but not in theory, as shown in Theorems 3 and 4. Examples of simplification rules are: $((\underline{\mathsf{if}}\ c\ \underline{\mathsf{then}}\ e\ \underline{\mathsf{else}}\ f) \le g) = \underline{\mathsf{if}}\ c\ \underline{\mathsf{then}}\ (e \le g)\ \underline{\mathsf{else}}\ (f \le g)$, $(\mathit{true} \wedge x) = \mathit{true}$, $(x = x) = \mathit{true}$, $(x + y) \le (x + z) = (y \le z)$, $(\underline{\mathsf{if}}\ c\ \underline{\mathsf{then}}\ e\ \underline{\mathsf{else}}\ c) = (c \wedge e)$. For example, if the expression e is $(\underline{\mathsf{if}}\ x = u\ \underline{\mathsf{then}}\ y\ \underline{\mathsf{else}}\ x) = u$ and the current correspondence table is $\{(x = u, b)\}$, the massaging step produces the new table $\{(x = u, b), (y = u, c)\}$ and the massaged expression $\overline{e} = (b \wedge c)$.

There are several interesting claims that can be made about the algorithm, despite its simplicity. The algorithm is *sound*, in that the abstract program is guaranteed to simulate the original, with respect to the initial set of atomic predicates, AP. A more interesting fact is that the algorithm is also *complete*, in that, if the TS of the original program has a finite simulation (bisimulation) quotient, then iterating the main loop (step 2) of the algorithm sufficiently many times (i.e., with a large enough value for K) results in an abstract program whose TS is simulation-equivalent (bisimulation-equivalent) to the original with respect to AP. These propositions are stated precisely below. Due to space limitations, we present only a sketch of the proof. We use the following notation: for an abstract state t (which is always defined over $\overline{\mathit{oldPred}}$), $\uparrow t$ (read as "up t") denotes the set of concrete states that agree with t on the valuations of the atomic predicates in *oldPred*; precisely, $\uparrow t = \{s | (\forall P : P \in \mathit{oldPred} : P(s) \equiv \overline{P}(t))\}$. Let $\mathcal{A}$ be the TS of the abstract program, and $\mathcal{C}$ the TS of the concrete program.

Lemma 0 (Invariance Lemma) *For every state t of $\mathcal{A}$, $\uparrow t$ is a non-empty set of concrete states.*

Proof Sketch. The formula Ψ ensures this for initial states, while Φ and the *wlp* computation ensure that the invariant holds. $\square$

1. The initial set of atomic predicates consists of those in the specification formula, the initial condition of the program, and the transition relation. This forms the set of unexamined predicates, *newPred*. The set *oldPred*, of already examined predicates is initially empty. The initial correspondence table C is empty.
2. While *newPred* is non-empty, and the iteration bound K is not reached,
   (a) Initialize *freshPred* to the empty set
   (b) For each predicate P in *newPred* and each action a, compute $(\overline{e}, C, fP) := massage(wlp_a(P), C)$, and add the predicates in fP to *freshPred*.
   (c) Compute $newPred, oldPred := freshPred, oldPred \cup newPred$
3. The abstract program $(X_{\mathcal{A}}, I_{\mathcal{A}}, T_{\mathcal{A}}, F_{\mathcal{A}})$ is formed as follows.
   - $X_{\mathcal{A}} = b \cup c$, where b is the set of boolean variables corresponding to *oldPred*, and c is the set of boolean variables corresponding to *newPred*.
   - $I_{\mathcal{A}}(b)$ is defined as $massage(I, C) \wedge \Psi$, where $\Psi = (\exists X :: I(X) \wedge (\bigwedge i : P_i \in oldPred : b_i \equiv P_i(X)))$.
   - $T_{\mathcal{A}}(bc, b'c')$ is defined as $(\bigvee a :: \Phi \wedge (\bigwedge i : P_i \in oldPred : b'_i \equiv massage(wlp_a(P_i), C)))$, where $\Phi = (\exists X :: (\bigwedge i : Q_i \in newPred : c_i \equiv Q_i(X)))$.
   - As we are not considering fairness, both F and $F_{\mathcal{A}}$ are *true*.

Fig. 3. The abstraction algorithm

Theorem 1 (Simulation Theorem) *The finite state abstract program simulates the concrete program w.r.t. the set of predicates AP.*

Proof Sketch. The claim is proved by showing that the relation R defined by $(s,t) \in R$ iff $s \in\uparrow t$ is a simulation relation from $\mathcal{C}$ to $\mathcal{A}$. As $AP \subseteq oldPred$, this relation preserves the values of the predicates in AP. $\square$

Theorem 2 (Bisimulation Theorem) *If $newPred = \emptyset$ on termination of step 2 of the abstraction algorithm, then the finite state abstract program is bisimulation-equivalent to the concrete program w.r.t. the set of predicates AP.*

Proof Sketch. If $newPred = \emptyset$ upon termination then, for each atomic predicate P in *oldPred* and each action a of the concrete program, $wlp_a(P)$ is a predicate over *oldPred*. As each action a is deterministic, for each predicate P, the transition $\overline{P}' = massage(wlp_a(P), C)$ (step 3) exactly captures the change to P after execution of action a. Let B be the relation on the disjoint union of the abstract and concrete programs defined by $(s,t) \in B$ iff $s \in\uparrow t \vee t \in\uparrow s$. The claim is proved by showing that B is a bisimulation, under the condition $newPred = \emptyset$. $\square$

Theorem 3 (Bisimulation Completeness) *If the concrete program has a finite reachable bisimulation quotient, there is an appropriate choice for the iteration bound K such that the abstract program produced by the algorithm is bisimulation-equivalent to the concrete program w.r.t. AP.*

Proof Sketch. Suppose that the concrete program has a finite reachable bisimulation quotient. By the results in [BFH90,LY92], the states of this quotient can be calculated as a finite partition Π by a symbolic partition refinement algorithm. The classes of Π are defined by boolean combinations of a finite set of atomic predicates, $\mathcal{P}$, that includes AP. The minimization algorithm computes the formulae defining these classes by repeated *wlp* computations. As *wlp* distributes over all boolean operators, these *wlp* computations may be rewritten to apply *wlp* only to atomic predicates, which is what our algorithm does. Since the unexamined predicate set *newPred* is obtained through *wlp* computations in a *breadth-first* manner, the algorithm eventually generates every predicate necessary for describing the classes of Π. Let K be the least iteration after which $\mathcal{P} \subseteq$ *oldPred* holds.

While the classes of the quotient can be described using a finite number of atomic predicates, our algorithm considers each predicate individually, not as part of a class formula. Hence, it is possible for our algorithm to generate new predicates (even semantically new predicates) beyond the Kth iteration. If *newPred* is empty after K iterations, the claim follows by Theorem 2. Otherwise, define the relation R by $(s,t) \in R$ iff the concrete state s and the set of concrete states $\uparrow t$ are both included in the same class of Π. As the extra predicates in *oldPred*$\backslash\mathcal{P}$ are unnecessary, the relation B defined by sBt iff $(sRt \vee tRs)$ can be shown to be a bisimulation between $\mathcal{C}$ and $\mathcal{A}$. As $AP \subseteq$ *oldPred*, this bisimulation preserves the predicates in AP. □

Theorem 4 (Simulation Completeness) *If the concrete program has a finite simulation quotient, there is an appropriate choice for the iteration bound K such that the abstract program produced by the algorithm is simulation-equivalent to the concrete program w.r.t. AP.*

Proof Sketch. There is a partition refinement algorithm [HHK95] to compute the greatest simulation relation that also employs *wlp* computations in a manner similar to the bisimulation minimization algorithms. The claim then follows from arguments similar to those in the proof of Theorem 3. □

4 Extensions to the Algorithm

4.1 Retaining a set of finite-domain control variables

The algorithm can be easily modified to retain a set of finite-domain control variables V while abstracting out the rest $(X\backslash V)$. We assume that V is closed under next-state dependencies; formally that, for any predicate $P(V)$, $wlp_a(P(V))$ does not introduce any atomic predicates over $X\backslash V$ other than those already present in the action a. During the massaging process, atomic predicates over V are *not* replaced with corresponding boolean variables. After step 2 terminates, the concrete program transitions for the V-variables are massaged and copied over to the abstract program. This modification was used in the Bakery example to retain the variables st_1, st_2. It is particularly useful when data variables are to be abstracted while retaining control variables.

4.2 Handling Fairness

There are two ways in which fairness may be specified. In the first type, one may specify fairness constraints on the actions of the program. Since the abstract program is strongly similar (bisimular) to the original, $A\mu$ (μ) properties over fair computations that hold in the abstract program also hold of the original. If fairness is specified instead by constraints on program states, we can add the atomic predicates from these constraints to *newPred* in step 1 of the algorithm and, in step 3, massage the fairness conditions to get the corresponding conditions for the abstract program.

4.3 Abstracting programs with unbounded nondeterminacy

We presented our algorithm for programs with bounded non-determinacy, which was exploited by considering each deterministic action individually. For transition relations that exhibit unbounded nondeterminacy, this partitioning is not possible, so we adopt a slightly different strategy. The key idea is to replace the computation (in step 2) of *wlp* for individual predicates with a computation of *wlp* for all clauses formed out of $oldPred \cup newPred$ (a clause is disjunction of literals, where each literal is an atomic predicate or its negation). This algorithm is both *sound* and *complete*, in the sense used earlier; however, it is probably impractical as an exponential number of clauses is generated in each iteration of step 2. We present below a simpler algorithm which is sound but not complete; however, as shown in the next section, it generalizes existing algorithms for data-insensitive programs.

Consider a partitioning of the transition relation into actions of the form $[W] : g(V,W) \hookrightarrow V := e(V,W)$, where W is a set of unbounded input variables, and V is a set of state variables. Hence, $wlp_a(P(V))$ is $(\forall W :: wlp_a^V(P))$, where $wlp_a^V(P)$ is $g(V,W) \Rightarrow P(W, e(V,W))$. The expression $wlp_a^V(P)$ may contain two types of predicates: state predicates over V and "mixed" predicates over $V \cup W$. Step 2 of the original algorithm is replaced with the following.

1. Compute $wlp_a^V(P)$ repeatedly with only the state predicates in *newPred* until no new state predicates are generated or the iteration bound K is reached.
2. If the iteration bound K is reached, proceed as before. Otherwise, define a boolean *input* variable c_i for each mixed predicate P_i in $newPred \cup oldPred$, and compute $F = (\exists W :: (\bigwedge_i :: c_i = P_i(W,V)))$, which is the condition for a c-valuation to be consistent. Quantifier-elimination results in a definition of F as a predicate on $V \cup \{c_i\}$. If there are new state predicates in F, add those to *newPred* and return to the first step, otherwise conjoin $massage(F, C)$ to the abstract transition relation.

Theorem 5 *(i) The modified algorithm always produces a conservative abstraction. (ii) If the algorithm terminates at step 2 with newPred* $= \emptyset$*, the abstract program is bisimular to the concrete program w.r.t.* AP.$\Box$

5 Applications

From the completeness results, our method can be applied to any program that has a finite quotient relative to the atomic predicates in the specification. Specific types of programs are particularly amenable to the application of this algorithm.

5.1 Data-Insensitive programs

We define *data-insensitive* programs as those that have a finite simulation quotient that preserves the values of all control variables. Hence, the control-flow is dependent on only a finite number of data predicates. Several papers describe restrictions on program syntax to ensure data-insensitivity, and provide abstraction algorithms which replace data domains by small finite domains, keeping the program actions unchanged [Wol86,HB95,ID96,Laz99]. This is justified by showing a bisimulation between the large- and small-domain instances.

Our program transformation method terminates for each of the above classes of programs. For instance, in [ID96], the only atomic predicate is =, and every assignment has the form $X \ := \ Y$. As there are n^2 distinct atomic equality predicates over the n variables $x_i \in X$, our algorithm terminates in at most n^2 steps, creating a bisimulation-equivalent abstraction by Theorem 5. The transformation of [ID96] replaces each data domain with $[0 \ldots n]$, hence the abstract state is represented by $n * \log(n)$ bits. In contrast, if only a few combinations of equality predicates are necessary, our method will result in states representable with fewer than $n * \log(n)$ bits [3]. One may also show the following theorem.

Theorem 6 *For programs without unbounded inputs where assignments are of the form $X \ := \ Y$, our abstraction algorithm terminates with a bisimulation-equivalent abstract program.*

Our algorithm also terminates for showing that the program below has an infinite computation, which is not possible to show with a finite domain method (cf. [HB95]). Thus, our algorithm is strictly more powerful than the finite-domain methods.

var x : **natural**
initially $x = 0$
action $a[i$: **natural**$]$ $\quad (x < i) \hookrightarrow x \ := \ i$

5.2 Symmetric Programs

Bisimulation reductions for semantically symmetric programs have been proposed in [ES93,CFJ93]. It is computationally difficult, however, to implement such reductions symbolically (i.e., with BDD's) [CFJ93]. Hence, [ET99] consider syntactically symmetric programs, defined using symmetric predicates such as

[3] A similar tradeoff occurs in finite-domain [PRSS99] vs. predicate abstraction [SGZ+98] approaches to verifying combinational circuits over integer variables.

$(\forall i :: P(i))$. The reduction of such a program with n processes, each with k local states, is obtained by introducing k variables $\{x_i\}$, each with a domain of $[0 \ldots n]$. By considering symmetric predicates to be atomic, our algorithm can be applied to such programs. In the worst case, it may produce all predicates of the form $x_i \geq l$ for each $l \in [0 \ldots n]$, requiring $k * (n+1)$ (correlated) boolean variables, but with a $poly(n)$ size BDD for each action. On the other hand, as our algorithm calculates only those predicates necessary for the reduction, it may also produce a program with fewer than the $k * \log(n)$ bits required by [ET99].

6 Related Work and Conclusions

Among related work, [GS97,CABN97,BLO98,CU98] also propose predicate abstraction methods. [CABN97] performs a simple syntactic transformation, but requires the use of a constraint solver during the model checking process. The methods of [GS97,BLO98,CU98] utilize general-purpose theorem proving to compute the abstract program, which is defined over boolean variables that correspond to an *a priori fixed* set of predicates. If the verification fails on the abstract program, the set of predicates is refined heuristically, and the abstraction process is repeated. In contrast, our algorithm, which has the same initial choice of predicates, both refines the set of predicates and computes the abstract program *automatically*, in a manner that is shown to be both sound and complete.

The papers [Wol86,ID96,HB95,Laz99] present algorithms for abstracting syntactically restricted programs. As shown in the previous section, our algorithm can be applied with guaranteed termination to these classes of programs, and is more generally applicable.

The symbolic minimization algorithms in [BFH90,LY92,HHK95] produce an *explicit* abstract transition system, which can be quite large and may preclude the effective application of symbolic model checking and partial order reduction methods. In contrast, our algorithm produces an *implicit* program description in the same syntax as the original program, which can then be analyzed using any model checking method. Our completeness results guarantee that our algorithm can find an equivalent finite abstract program, if one exists, given an appropriate termination bound.

The deductive model checking algorithm of [SUM99] produces an abstraction relative to a *LTL* specification by a process of iterative refinement which, however, requires significant human intervention. In [KP00], it is shown that finite state abstractions exist for programs that satisfy *LTL* properties; the completeness proof is non-constructive in general but partly utilizes predicate abstraction. Other automatic abstraction methods [DGG93,CGL94] apply only to finite state systems. Other semi-algorithms [HGD95,KMM+97,BGP97,BDG+98] directly model check infinite state systems without computing an abstract program. The algorithm in [GS92] for model checking a special type of parameterized system relies on a trace equivalent abstraction.

We have hand-simulated our algorithm for a simple data transfer protocol that transmits natural numbers with 0 representing null data. The correctness property is that data that is sent is eventually received. The abstracted protocol is verified by COSPAN using less resources (time, space) than the verification of the original protocol with data domain size 1. We are currently developing a prototype implementation to experiment with larger examples.

References

[BCG88] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59, 1988.

[BDG+98] J. Bohn, W. Damm, O. Grumberg, H. Hungar, and K. Laster. First-order-CTL model checking. In *FST&TCS*, volume 1530 of *LNCS*, 1998.

[BFH90] A. Bouajjani, J-C. Fernandez, and N. Halbwachs. Minimal model generation. In *CAV*, volume 531 of *LNCS*, 1990. Full version in *Science of Computer Programming*, vol. 18, 1992.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *CAV*, volume 1254 of *LNCS*, 1997.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *CAV*, volume 1427 of *LNCS*, 1998.

[CABN97] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *CAV*, volume 1254 of *LNCS*, 1997.

[CE81] E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *LNCS*, 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *TOPLAS*, 8(2), 1986.

[CFJ93] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *CAV*, volume 697 of *LNCS*, 1993.

[CGL94] E.M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *TOPLAS*, 1994.

[CU98] M.A. Colon and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *CAV*, volume 1427 of *LNCS*, 1998.

[DGG93] D. Dams, R. Gerth, and O. Grumberg. Generation of reduced models for checking fragments of CTL. In *CAV*, volume 697 of *LNCS*, 1993.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. In *C.ACM*, volume 18, 1975.

[EH86] E.A. Emerson and J. Halpern. "Sometimes" and "Not Never" revisited: On branching versus linear time temporal logic. *J.ACM*, 33, 1986.

[EL86] E.A. Emerson and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *LICS*, 1986.

[ES93] E.A. Emerson and A.P. Sistla. Symmetry and model checking. In *CAV*, number 697 in LNCS, 1993.

[ET99] E.A. Emerson and R.J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *CHARME*, volume LNCS, 1999.

[GL94] O. Grumberg and D. Long. Model checking and modular verification. *TOPLAS*, 16, 1994.

[GS92] S. German and A.P. Sistla. Reasoning about systems with many processes. *J.ACM*, 1992.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, 1997.

[HB95] R. Hojati and R.K. Brayton. Automatic datapath abstraction of hardware systems. In *CAV*, volume 939 of *LNCS*, 1995.

[HGD95] H. Hungar, O. Grumberg, and W. Damm. What if model checking must be truly symbolic. In *CHARME*, volume 987 of *LNCS*, 1995.

[HHK95] M.R. Henzinger, T.A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *IEEE FOCS*, 1995.

[ID96] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 1996.

[Kel76] R. M. Keller. Formal verification of parallel programs. *C.ACM*, 1976.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *CAV*, volume 1254 of *LNCS*, 1997.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 1983.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation (to appear)*, 2000. Earlier version at MFCS 1998, published in LNCS 1450.

[Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *C.ACM*, August 1974.

[Laz99] R.S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University, 1999.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. In *STOC*, 1992.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *2nd IJCAI*, 1971.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science: 5th GI-Conference, Karlsruhe*, volume 104 of *LNCS*, 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *FOCS*, 1977.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *CAV*, volume 1633 of *LNCS*, 1999.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*, 1982.

[SGZ+98] K. Sajid, A. Goel, H. Zhou, A. Aziz, S. Barber, and V. Singhal. Bdd-based procedures for a theory of equality with uninterpreted functions. In *CAV*, volume 1427 of *LNCS*, 1998.

[SUM99] H. Sipma, T. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15, 1999. Earlier version at CAV 1996, published in LNCS 1102.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL*, 1986.