Lifting Temporal Proofs Through Abstractions

Kedar S. Namjoshi

Bell Labs, Lucent Technologies
kedar@research.bell-labs.com

Abstract. Model checking is often performed by checking a transformed property on a suitable finite-state abstraction of the source program. Examples include abstractions resulting from symmetry reduction, data independence, and predicate abstraction. The two programs are linked by a structural relationship, such as simulation or bisimulation, guaranteeing that if the transformed property holds on the abstract program, the property holds on the original program.
Recently, several algorithms have been developed to automatically generate a *deductive proof* of correctness from a model checker. A natural question, therefore, is how to 'lift' a deductive proof that is generated for an abstract program back into the original program domain. In this paper, we show how this can be done for general temporal properties, relative to several types of abstraction relationships between the two programs. We develop simplifications of the lifting scheme for common types of abstractions, such as predicate abstraction. We also show how one may generate easily checkable lifted proofs, which find use in applications such as proof-carrying code, and in the use of model checkers as decision procedures in theorem proving.

1 Introduction

Model Checking, introduced in [CE81,QS82], has enabled routine automatic verification of programs, especially in hardware and, more recently, in software. A key component of the application of model checking is the use of *abstraction* methods, which reduce the problem of checking programs with large (possibly infinite) state-spaces to checking a transformed property on a suitable small, finite-state abstraction of the original program. Typically, the two programs are related by a structural relationship, such as simulation [Mil71], or bisimulation [Par81]. This guarantees that if the transformed property holds on the abstract program, then the original property holds on the concrete program. Common examples of abstraction are those resulting from symmetry reduction [ES93,CFJ93], data independence [Wol86], and predicate abstraction [GS97].

Recently, several algorithms [RRR00,Nam01,PPZ01,TC02] have been developed that automatically generate a *deductive proof* of correctness from a model checker; such proofs have many applications, which are mentioned in these papers. In the context of model checking with abstraction, however, there remains a missing link: how to 'lift' a deductive proof generated automatically for the abstract program back to the concrete program domain. While it is well-known that property satisfaction can be lifted back through abstraction [HM85,BCG88], these results rely on the set-based semantics of properties, and do not indicate how to lift deductive proofs of satisfaction.

In this paper, we tackle this question for temporal properties in the Mu-calculus [Koz82], and several types of abstractions. The Mu-calculus is a very expressive temporal logic, which subsumes commonly used logics such as CTL, CTL*, and linear temporal logic. We use a deductive proof system developed in [Nam01] to represent proofs of correctness for mu-calculus properties. This proof system relies on invariant assertions (for showing safety) and rank functions (for showing progress). It is based on a direct translation of the mu-calculus to alternating tree automata [EJ91]; thus, the proof steps closely follow the syntax of the formula. We show how the invariance assertions and rank functions in a valid proof on the abstract program can be lifted to the concrete program domain to constitute a valid proof in that domain. We define how such lifting works for abstractions based on simulation (preserving universal properties), bisimulation (which preserves all properties), and abstraction with 3-valued logics (which preserves all properties, but with a potential loss of completeness).

We then explore several applications of this lifting scheme. We develop simplified versions of the scheme for common abstractions, such as the encoding of non-boolean types with bit vectors in symbolic model checkers, and predicate abstraction. We also examine the question of generating easily checkable proofs, for applications to proof-carrying code [NL96] and the addition of model checking as a trusted decision procedure to theorem provers. We show that it is possible to combine a checkable proof of the abstraction relationship and a checkable proof on the abstract program into a checkable proof showing correctness of the property on the source program.

Related Work There is not, to the author's knowledge, an earlier systematic study of how proofs may be lifted from abstract to concrete programs, but several interesting and important instances of such lifting can be mentioned.

In [APR$^+$01], an automatically generated invariant for a small instance of a parameterized system is heuristically (and automatically) lifted to a candidate invariant for the entire system, which is subsequently checked for validity. This check is required as it is not known in advance – unlike in our setting – whether there is an abstraction relationship between the full system and its small instances. The BLAST toolkit for verifying sequential C code [HJM$^+$02] includes a method for lifting a checkable linear-time invariance proof through the abstraction computed by their algorithm into a checkable proof of correctness for the original program. The results in this paper apply to more general properties, including liveness and branching-time properties, and to several other types of abstractions. An alternative to proof-carrying code, called model-carrying code, is suggested in [SRRS02]. In one version, code is sent together with an abstract program and a proof of abstraction. The recipient must re-validate the property by model checking the abstract program. Our results show that it is possible to avoid this potentially expensive model checking step by combining the abstraction proof with an automatically generated deductive proof of the property.

2 Background

In this section, we define the mu-calculus and its universal fragment. For completeness' sake, we reproduce the deductive proof system from [Nam01]. In the following section, we show how to lift proofs that are developed in this system.

The mu-calculus The mu-calculus [Koz82] is a branching time temporal logic that subsumes commonly used logics such as LTL, ω-automata, CTL, and CTL^* [EL86]. It is parameterized with respect to Σ (state labels), Γ (action labels), and V (the set of fixpoint variables). Formulas of the logic are defined using the following grammar, where l is in Σ, a is in Γ, Z is in V, and μ is the least fixpoint operator: $\Phi ::= l \mid Z \mid \langle a \rangle \Phi \mid \neg\Phi \mid \Phi \wedge \Phi \mid (\mu Z : \Phi)$.

We assume that Σ and Γ are fixed in the rest of the paper. A formula must have each variable under the scope of an even number of negation symbols. It is *closed* iff every variable is bound by a fixpoint operator. Formulas can be converted to positive normal form by introducing the operators $\Phi_1 \vee \Phi_2 ::= \neg(\neg(\Phi_1) \wedge \neg(\Phi_2))$, $[a]\Phi ::= \neg\langle a \rangle(\neg\Phi)$ and $(\nu Z : \Phi) ::= \neg(\mu Z : \neg\Phi(\neg Z))$, and using de Morgan rules to push negations inwards. The result is a formula where negations are applied only to elements of Σ. A *universal* formula is one where its positive form does not contain an $\langle a \rangle$ operator, for any a in Γ.

Formulas are evaluated over labeled transition systems (LTS's) [Kel76]. An LTS is a tuple $(S, \hat{s}, R, L)$, where S is a non-empty set of *states*, $\hat{s}$ in S is the *initial state*, $R \subseteq S \times \Gamma \times S$ is the *transition relation*, and $L : S \rightarrow \Sigma$ is a *labeling function* on states. We assume that R is *total*: i.e., for any s and a, there exists t such that $(s, a, t) \in R$. A set, I, of initial states can be accommodated by adding a dummy initial state with a transition to each state in I. The evaluation of a formula f, represented as $\|f\|_c$, is a subset of S, defined relative to a *context* c mapping variables to subsets of S. The evaluation rules are standard (see, e.g., [Sti95]) and are omitted here. A state s in the LTS *satisfies* a closed mu-calculus formula f iff $s \in \|f\|_\perp$, where $\perp$ maps every variable to the empty set. The LTS satisfies f iff the initial state $\hat{s}$ satisfies f.

Alternating Automata The proof system we use is based on alternating automata rather than the mu-calculus. Alternating automata have a simpler structure, which results in simple proof rules. Furthermore, it is shown in [EJ91,JW95] that there is a straightforward translation from a closed mu-calculus formula to an equivalent alternating automaton: in effect, the automaton skeleton is just the parse graph of the formula (defining the acceptance condition requires an analysis of the alternation among fixpoint operators).

An *alternating automaton* is specified by a tuple $(Q, \hat{q}, \delta, F)$, where Q is a non-empty set of states, and $\hat{q} \in Q$ is the initial state. F is a partition $(F_0, F_1, \ldots, F_n)$ of Q, which defines a *parity* acceptance condition. An infinite sequence over Q satisfies F iff the *smallest* index i for which a state in F_i occurs infinitely often on the sequence is *even*. The transition function, δ, maps a pair from $Q \times \Sigma$ to a positive boolean expression formed using the operators $\wedge$, $\vee$ applied to elements of the form $true, false, q, \langle a \rangle q$ and $[a]q$, where $a \in \Gamma$, and $q \in Q$. We assume, without loss of generality, that δ is in a simple normal form where the boolean expressions have one of the following forms: $q1 \wedge q2, q1 \vee q2, \langle a \rangle q1, [a]q1, \mathit{true}, \mathit{false}$. Conversion to an equivalent, normal form automaton can be done in linear time.

The satisfaction of an automaton property by an LTS may be defined in terms of an infinite, 2-player game [EJ91] – we summarize this game here. A configuration (s, q) (s is an LTS state, q is an automaton state) is a win for player I if $\delta(q, L(s)) = \mathit{true}$; it is a loss if $\delta(q, L(s)) = \mathit{false}$. For other values of δ, player I picks the next move iff $\delta(q, L(s))$ is either $\langle a \rangle q1$ or $q1 \vee q2$. Player I picks an a-successor for s for $\langle a \rangle q1$, or the choice of disjunct. Similarly, player II picks an a-successor for s for $[a]q1$, or the choice of conjunct for $q1 \wedge q2$. A play of the game is a win for player I iff it either ends in a winning configuration, or it is infinite and the sequence of automaton states on it satisfies F. A *strategy* is a function mapping a partial play to the next move; given strategies for players I and II, one can generate the possible plays. Finally, the LTS satisfies the automaton property iff player I has a *winning* strategy (one for which every generated play is a win for player I) for the game played on the computation tree of the LTS from the initial configuration $(\hat{s}, \hat{q})$.

Deductively Verifying Mu-calculus Properties Deductive proof systems for verifying sequential and concurrent programs rely on showing safety through invariants, and progress through a decrease of a rank function (alternative names are 'variant function', 'progress measure'). The proof system we use for verifying automaton properties (from [Nam01]) is based on the same concepts. Suppose that $M = (S, \hat{s}, R, L)$ is an LTS, and $A = (Q, \hat{q}, \delta, F)$ is a normal form automaton, where $F = (F_0, F_1, \ldots, F_{2n})$. To show that M satisfies A, one exhibits:

- for each automaton state q, an *invariant* predicate, ϕ_q, over S, expressed in some assertion language, and
- for each automaton state q, a partial *rank function*, ρ_q, which maps states in S to elements of a set W equipped with a well-order $\preceq$. The set W is the product, $W_1 \times \ldots \times W_n$, of well ordered sets $\{(W_i, \leq_i)\}$, and $\preceq$ is the lexicographic well-order obtained from $\{\leq_i\}$.

The invariants and rank functions must satisfy the proof obligations shown in Figure 1 (the predicate $l(s)$ is defined as $(L(s) = l)$, and the notation $[f]$ means that the formula f is valid.) For instance, the rule for a transition $\delta(q, l) = [a]q1$ asserts that for every state with label l that satisfies the invariant ϕ_q and has rank k, all of its successors must satisfy the invariant for $q1$ and change rank appropriately. To ensure progress towards termination of least fixpoints, the rank must change in a specific manner. This is given by a *rank change* relation, $\triangleleft_q$ (q is an automaton state), defined over $W \times W$. First, let $\prec_i$ be the restriction of $\prec$ to the first i vector components: formally, $a \prec_i b \equiv (\exists k : 1 \leq k \leq i : a[k] <_k b[k] \wedge (\forall j : 1 \leq j < k : a[j] = b[j]))$. For any a, b, and q, the relation $a \triangleleft_q b$ holds iff for the (unique, since F is a partition) index k such that $q \in F_k$, for some i, either $k = 2i$ and $a \preceq_i b$, or $k = 2i - 1$ and $a \prec_i b$. Informally speaking, the strict decrease of rank at odd indexed states ensures that in the game, player I can not get 'stuck' in such states, so the parity condition holds. In [Nam01], it is shown that this proof system is sound, and relatively complete.

- ***Consistency:*** For each $q \in Q$, $[\phi_q \Rightarrow (\exists k : (\rho_q = k))]$ (ρ_q is defined for every state in ϕ_q)
- ***Initiality:*** $[\phi_{\hat{q}}(\hat{s}) \equiv \mathit{true}]$ (the initial state satisfies the initial invariant)
- ***Invariance and Progress:*** For each $q \in Q$, and $l \in \Sigma$, depending on the form of $\delta(q, l)$, check the following.
  - true: there is nothing to check.
  - false: $[\phi_q \Rightarrow \neg l]$ holds,
  - $q1 \wedge q2$: $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow (\phi_{q1} \wedge (\rho_{q1} \lhd_q k)) \wedge (\phi_{q2} \wedge (\rho_{q2} \lhd_q k))]$
  - $q1 \vee q2$: $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow (\phi_{q1} \wedge (\rho_{q1} \lhd_q k)) \vee (\phi_{q2} \wedge (\rho_{q2} \lhd_q k))]$
  - $\langle a \rangle q1$: $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow \langle a \rangle (\phi_{q1} \wedge (\rho_{q1} \lhd_q k))]$
  - $[a]q1$: $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow [a](\phi_{q1} \wedge (\rho_{q1} \lhd_q k))]$

Fig. 1. A deductive proof system for verifying branching-time properties.

3 Lifting Proofs

Given a LTS N that abstracts an LTS M, we show how a proof of a property f on N can be lifted to a proof of the same property on M. We consider two common notions of abstraction: *simulation* [Mil71], which preserves only universal properties, and *bisimulation* [Par81], which preserves properties of the full mu-calculus (cf. [BCG88,Sti95]). Let M and N be LTS's, with $\Gamma_M = \Gamma_N$ and $\Sigma_M = \Sigma_N$. A relation $\xi \subseteq S_M \times S_N$ is a *simulation* from M to N if, and only if:

- The initial states of M and N are related, i.e., $\hat{s}_M \xi \hat{s}_N$, and
- For every s in S_M and t in S_N such that $s \xi t$ holds:
  - $L_M(s) = L_N(t)$, and
  - for every $a \in \Gamma_M$, and every u in S_M such that $(s, a, u) \in R_M$, there exists v in S_N such that $(t, a, v) \in R_N$ and $u \xi v$ holds.

A relation ξ is a *bisimulation* if, and only if, both ξ and its converse relation, ξ^{-1}, are simulations. We say that M is *simulated by* (*bisimilar to*) N if there exists a simulation (bisimulation) relation from M to N.

Notation. In the rest of this paper, proofs are presented in a format popularized by Dijkstra and Scholten in [DS90]. Here, individual steps of a proof are linked by a transitive connective such as $\equiv$ or $\Rightarrow$, along with a hint for why the connection holds. The notation $(\mathcal{Q}x : r(x) : p(x))$ is used to represent an operation where x is the dummy variable, $r(x)$ is the range of x, and $p(x)$ is the term being operated on. Therefore, $(\exists x : r(x) : q(x))$ is the same as $(\exists x : r(x) \wedge q(x))$, while $(\forall x : r(x) : q(x))$ is the same as $(\forall x : r(x) \Rightarrow q(x))$.

3.1 Lifting Proofs through a Simulation

Suppose that M is simulated by N through a relation ξ, and f is a universal property. Since f is universal, the transition relation of the alternating automaton for f does not have any occurrence of $\langle a \rangle$, although any of the other connectives may occur. A proof, Π, that N satisfies f is given by the tuple (ϕ, ρ, W), where ϕ is the invariant function, ρ is the rank function, and W is the well-ordered set of ranks. Let $\Pi' = (\phi', \rho', W')$ be defined by:

- (lifting invariants) $\phi'_q(s) \equiv (\exists t : s\xi t : \phi_q(t))$
- (lifting rank functions) $\rho'_q(s) = (\mathbf{min}\, t : s\xi t \wedge \phi_q(t) : \rho_q(t))$, where **min** is the minimum relative to $\preceq$ (the condition $\phi_q(t)$ ensures that $\rho_q(t)$ is defined.)
- (lifting rank sets) $W' = W$, $\preceq' = \preceq$

The lifted invariant is just the concretization of the abstract invariant (this is a well-known fact from abstract interpretation theory). We make frequent use of the following lemma in the subsequent proofs. It captures all of the information we need about $\lhd_q$ in order to show that the lifted proof is valid.

Lemma 0. For any a, b, c, d of equal length, and any automaton state q, if $a \preceq b$, $b \lhd_q c$, and $c \preceq d$, then $a \lhd_q d$.

Proof. By definition, $\lhd_q$ is $\preceq_i$ or $\prec_i$, for some i. The lemma holds as the $\preceq$ relation is stronger than $\preceq_i$, and $\preceq_i$ is transitive (it is a partial order). □

Theorem 0. For a universal property f, if $\Pi = (\phi, \rho, W)$ is a valid proof that $N \models f$, and M is simulated by N through a relation ξ, then $\Pi' = (\phi', \rho', W')$, as defined above, forms a valid proof that $M \models f$.

Proof. Consider each type of proof subgoal.
(Initiality) We have to show that $[\phi'_{\hat{q}}(\hat{s}_M) \equiv \mathit{true}]$.

$$
\begin{array}{ll}
 & \phi'_{\hat{q}}(\hat{s}_M) \\
\equiv & \quad \text{(definitions)} \\
 & (\exists t : \hat{s}_M \xi t : \phi_{\hat{q}}(t)) \\
\Leftarrow & \quad \text{(logic)} \\
 & \hat{s}_M \xi \hat{s}_N \wedge \phi_{\hat{q}}(\hat{s}_N) \\
\equiv & \quad \text{(} \xi \text{ is a simulation)} \\
 & \phi_{\hat{q}}(\hat{s}_N) \\
\equiv & \quad \text{(Initiality for the given proof)} \\
 & \mathit{true}
\end{array}
$$

(Consistency) We have to show that, for any state s, $\phi'_q(s)$ implies that $\rho'_q(s)$ is defined. From the definition of ρ', if $\phi'_q(s)$ holds, then the range of t in $(\mathbf{min}\, t)$ is non-empty; thus, $\rho'_q(s)$ is defined.
(Invariance and Progress) Based on the form of $\delta(q, l)$:
0. *true*: nothing to check.
1. *false*: given that $[\phi_q \wedge l \Rightarrow \mathit{false}]$, we have to show that $[\phi'_q \wedge l \Rightarrow \mathit{false}]$. For any s,

$$
\begin{array}{ll}
 & \phi'_q(s) \wedge l(s) \\
\equiv & \quad (\text{ definitions }) \\
 & (\exists t : s\xi t : \phi_q(t) \wedge l(s)) \\
\Rightarrow & \quad (\text{ by simulation } \xi,\ l(s) \Rightarrow l(t) \ ) \\
 & (\exists t : s\xi t : \phi_q(t) \wedge l(t)) \\
\Rightarrow & \quad (\text{ by subgoal for } N \ ) \\
 & (\exists t : s\xi t : \mathit{false}) \\
\equiv & \quad (\text{ logic }) \\
 & \mathit{false}
\end{array}
$$

2. $q1 \wedge q2$: we have to show that $[(\phi'_q \wedge l \wedge \rho'_q = k) \Rightarrow (\phi'_{q1} \wedge \rho'_{q1} \lhd_q k) \wedge (\phi'_{q2} \wedge \rho'_{q2} \lhd_q k)]$. Consider the first consequence (the second has a symmetric proof). For any s,

$$
\begin{array}{ll}
 & \phi'_q(s) \wedge l(s) \wedge \rho'_q(s) = k \\
\equiv & \quad (\text{ definitions }) \\
 & (\exists t : s\xi t : \phi_q(t)) \wedge l(s) \wedge (\mathbf{min}\, t : s\xi t \wedge \phi_q(t) : \rho_q(t)) = k \\
\Rightarrow & \quad (\text{ definition of } \mathbf{min} \ ) \\
 & (\exists t : s\xi t \wedge \phi_q(t) \wedge \rho_q(t) = k \wedge l(s)) \\
\Rightarrow & \quad (\ \xi \text{ is a simulation }) \\
 & (\exists t : s\xi t \wedge \phi_q(t) \wedge \rho_q(t) = k \wedge l(t)) \\
\Rightarrow & \quad (\text{ corresponding subgoal for } N \ ) \\
 & (\exists t : s\xi t \wedge \phi_{q1}(t) \wedge \rho_{q1}(t) \lhd_q k) \\
\Rightarrow & \quad (\text{ logic }) \\
 & (\exists t : s\xi t \wedge \phi_{q1}(t)) \wedge (\exists t : s\xi t \wedge \phi_{q1}(t) : \rho_{q1}(t) \lhd_q k) \\
\Rightarrow & \quad (\text{ definitions, Lemma 0 }) \\
 & \phi'_{q1}(s) \wedge \rho'_{q1}(s) \lhd_q k
\end{array}
$$

3. $q1 \vee q2$: this is similar to the previous case.
4. $[a](q1)$: we have to show that $[(\phi'_q \wedge l \wedge \rho'_q = k) \Rightarrow [a](\phi'_{q1} \wedge \rho'_{q1} \lhd_q k)]$. For any s and u,

$$
\begin{array}{ll}
 & (\phi'_q(s) \wedge l(s) \wedge \rho'_q(s) = k) \wedge (s,a,u) \in R_M \\
\equiv & \quad (\text{ definitions }) \\
 & (\exists t : s\xi t : \phi_q(t)) \wedge l(s) \wedge (\mathbf{min}\, t : s\xi t \wedge \phi_q(t) : \rho_q(t)) = k \wedge (s,a,u) \in R_M \\
\Rightarrow & \quad (\text{ definition of } \mathbf{min} \ ) \\
 & (\exists t : s\xi t \wedge \phi_q(t) \wedge \rho_q(t) = k \wedge l(s) \wedge (s,a,u) \in R_M) \\
\Rightarrow & \quad (\ \xi \text{ is a simulation }) \\
 & (\exists t : s\xi t \wedge \phi_q(t) \wedge \rho_q(t) = k \wedge l(t) \wedge (s,a,u) \in R_M \wedge (\exists v : (t,a,v) \in R_N : u\xi v)) \\
\equiv & \quad (\text{ rearranging }) \\
 & (\exists t, v : s\xi t \wedge \phi_q(t) \wedge \rho_q(t) = k \wedge l(t) \wedge (t,a,v) \in R_N \wedge u\xi v \wedge (s,a,u) \in R_M) \\
\Rightarrow & \quad (\text{ corresponding subgoal for } N\text{, dropping some terms }) \\
 & (\exists v : \phi_{q1}(v) \wedge \rho_{q1}(v) \lhd_q k \wedge u\xi v) \\
\Rightarrow & \quad (\text{ logic }) \\
 & (\exists v : u\xi v : \phi_{q1}(v)) \wedge (\exists v : u\xi v \wedge \phi_{q1}(v) : \rho_{q1}(v) \lhd_q k)
\end{array}
$$

$\Rightarrow$ (definitions, Lemma 0)

$\phi'_{q1}(u) \wedge \rho'_{q1}(u) \lhd_q k$

$\square$

3.2 Lifting proofs through Bisimulations

Theorem 1. For any mu-calculus property f, if $\Pi = (\phi, \rho, W)$ is a valid proof that $N \models f$, and M and N are bisimilar through a relation ξ, then $\Pi' = (\phi', \rho', W')$, as defined above, forms a valid proof that $M \models f$.

Proof. Since we use the same lifted invariant and ranking function as in the simulation proof, the earlier proof claims carry over unchanged, and we only need add the proof for the $\langle\rangle$ case.
5. $\langle a \rangle q1$: We have to show that $[\phi'_q \wedge l \wedge \rho'_q = k \Rightarrow \langle a \rangle(\phi'_{q1} \wedge \rho'_{q1} \lhd_q k)]$ holds. For any s,

$$
\begin{array}{ll}
 & \phi'_q(s) \wedge l(s) \wedge \rho'_q(s) = k \\
\equiv & \quad (\text{ definitions }) \\
 & (\exists t : s\xi t : \phi_q(t)) \wedge l(s) \wedge (\mathbf{min}\, t : s\xi t \wedge \phi_q(t) : \rho_q(t)) = k \\
\Rightarrow & \quad (\text{ definition of } \mathbf{min}\ ) \\
 & (\exists t : s\xi t \wedge \phi_q(t) \wedge \rho_q(t) = k \wedge l(s)) \\
\Rightarrow & \quad (\ \xi \text{ is a simulation }) \\
 & (\exists t : s\xi t \wedge \phi_q(t) \wedge \rho_q(t) = k \wedge l(t)) \\
\Rightarrow & \quad (\text{ corresponding subgoal for } N\ ) \\
 & (\exists t : s\xi t \wedge (\exists v : (t,a,v) \in R_N : \phi_{q1}(v) \wedge \rho_{q1}(v) \lhd_q k)) \\
\equiv & \quad (\text{ rearranging }) \\
 & (\exists t, v : s\xi t \wedge (t,a,v) \in R_N \wedge \phi_{q1}(v) \wedge \rho_{q1}(v) \lhd_q k) \\
\Rightarrow & \quad (\ \xi \text{ is a bisimulation }) \\
 & (\exists v : (\exists u : (s,a,u) \in R_M \wedge u\xi v) \wedge \phi_{q1}(v) \wedge \rho_{q1}(v) \lhd_q k) \\
\equiv & \quad (\text{ rearranging }) \\
 & (\exists u : (s,a,u) \in R_M : (\exists v : u\xi v \wedge \phi_{q1}(v) \wedge \rho_{q1}(v) \lhd_q k)) \\
\Rightarrow & \quad (\text{ logic }) \\
 & (\exists u : (s,a,u) \in R_M : (\exists v : u\xi v \wedge \phi_{q1}(v)) \wedge (\exists v : u\xi v \wedge \phi_{q1}(v) \wedge \rho_{q1}(v) \lhd_q k)) \\
\Rightarrow & \quad (\text{ definitions, Lemma 0 }) \\
 & (\exists u : (s,a,u) \in R_M : \phi'_{q1}(u) \wedge \rho'_{q1}(u) \lhd_q k)
\end{array}
$$

$\square$

3.3 Lifting proofs through 3-valued refinement

A *modal* transition system (MTS) [LT88] is a tuple $(S, \hat{s}, R^{may}, R^{must}, L)$, where $S, \hat{s}$, and L are as in the LTS definition, but there are two transition relations, R^{may} (the *may*-transitions), and R^{must} (the *must* transitions), with the constraint that $R^{must} \subseteq R^{may}$. The interpretation of a mu-calculus property, f, on a MTS [HJS01] is given by inductively computing a pair, (f^{nec}, f^{pos}), with the property that $[f^{nec} \Rightarrow f^{pos}]$. The f^{nec} computation interprets $\langle a \rangle$ using R^{must}, and $[a]$ using R^{may}. Symmetrically, the f^{pos} computation uses R^{may} to interpret $\langle a \rangle$ and R^{must} to interpret $[a]$.

For MTS's M and N, N is an abstraction of M iff there exists a relation $\xi \subseteq S_M \times S_N$ such that ξ is a simulation relative to the R^{may} relations of M and N, and ξ^{-1} is a simulation relative to the R^{must} relations of N and M. Note that abstraction coincides with bisimulation and $[f^{pos} \equiv f^{nec}]$ when both MTS's are, in fact, LTS's (i.e., $R^{may} = R^{must}$).

Theorem 2. [HJS01] For MTS's M, N and a mu-calculus property f, if N abstracts M, then: (i) if $f_N^{nec}(\hat{s}_N)$, then $f_M^{nec}(\hat{s}_M)$ (success), and (ii) if $\neg f_N^{pos}(\hat{s}_N)$, then $\neg f_M^{pos}(\hat{s}_M)$ (failure).

Notice that it is not always possible to give a definite answer; there is a completeness gap. However, MTS refinement can be coarser than bisimulation, and yet be able to give a definite answer for arbitrary mu-calculus properties. It is necessary to modify our proof system slightly in order to apply it to MTS's. The proof system now checks whether f^{nec} holds: to do so, the $\langle a \rangle$ operator is interpreted using R^{must}, and the $[a]$ operator using R^{may}.

Theorem 3. Let M and N be MTS's such that N is an abstraction of M through a relation ξ, and let f be a mu-calculus property. If $\Pi = (\phi, \rho, W)$ is a valid proof that f^{nec} holds for N, then $\Pi' = (\phi', \rho', W')$, where the primed components are defined as before, is a valid proof that f^{nec} holds on M.

Proof. This follows immediately by inspecting the proofs given earlier for the validity of Π' in the simulation and bisimulation cases, substituting R^{may} for R in the $[a]$ proof, and R^{must} for R in the $\langle a \rangle$ proof. $\Box$

Hence, if the success case holds, a proof of success can be lifted from N to M. By the semantics, $[\neg(f^{pos}) \equiv (\neg f)^{nec}]$. Thus, failure on N is equivalent to showing success for a negated property, so that a proof of failure becomes a proof of success of the negated property. So, on failure, we can use the previous theorem to lift the proof that $(\neg f)^{nec}$ holds from N to M.

3.4 Rank Functions and Rank Relations

A proof, as defined, is a triple (ϕ, ρ, W), where ρ is a collection of rank functions. For verifying a proof, however, we really need the rank *relations* $\rho_q(s) = k$ and $\rho_q(s) \lhd_r k$. We write the first relation as $\rho_q^{=}(s, k)$, and the second as $\rho_q^{\lhd}(r, s, k)$. We now show how to lift these relations through abstraction. For this purpose, we need a third relation, $\rho_q(s) \succeq k$, which we represent by $\rho_q^{\succeq}(s, k)$. The lifted relations can be calculated in terms of the original relations, as shown below.

$$
\begin{array}{ll}
 & \rho_q'^{=}(s,k) \\
\equiv & \quad \text{(by definition)} \\
 & (\mathbf{min}\, t : s\xi t \,\wedge\, \phi_q(t) : \rho_q(t)) = k \\
\equiv & \quad \text{(definition of } \mathbf{min} \text{)} \\
 & (\forall t : s\xi t \,\wedge\, \phi_q(t) : \rho_q(t) \succeq k) \,\wedge\, (\exists t : s\xi t \,\wedge\, \phi_q(t) : \rho_q(t) = k) \\
\equiv & \quad \text{(definitions)} \\
 & (\forall t : s\xi t \,\wedge\, \phi_q(t) : \rho_q^{\succeq}(t,k)) \,\wedge\, (\exists t : s\xi t \,\wedge\, \phi_q(t) : \rho_q^{=}(t,k))
\end{array}
$$

$$
\begin{array}{ll}
 & \rho'^{\lhd}_q(r,s,k) \\
\equiv & \quad (\text{ by definition }) \\
 & (\mathbf{min}\, t : s\xi t \ \wedge\ \phi_q(t) : \rho_q(t)) \lhd_r k \\
\equiv & \quad (\text{ definition of } \mathbf{min}, \text{ Lemma } 0 \ ) \\
 & (\exists t : s\xi t \ \wedge\ \phi_q(t) : \rho^{\lhd}_q(r,t,k))
\end{array}
$$

$$
\begin{array}{ll}
 & \rho'^{\succeq}_q(s,k) \\
\equiv & \quad (\text{ by definition }) \\
 & (\mathbf{min}\, t : s\xi t \ \wedge\ \phi_q(t) : \rho_q(t)) \succeq k \\
\equiv & \quad (\text{ definition of } \mathbf{min}, \text{ Lemma } 0 \ ) \\
 & (\forall t : s\xi t \ \wedge\ \phi_q(t) : \rho^{\succeq}_q(t,k))
\end{array}
$$

Rank relations can be easier to compute than rank functions. For instance, if ξ, ϕ, and the abstract rank relations are expressed in Presburger arithmetic, the lifted relations are also expressible in Presburger arithmetic.

An alternative way of checking a proof is to fully expand the proof obligations, removing the dependence on the rank variable k through the 1-point rule. For instance, the fully expanded form of the $[a]q1$ obligation is $(\forall s : \phi_q(s) \ \wedge\ l(s) \Rightarrow (\forall u : R(s,a,u) : \phi_{q1}(u) \ \wedge\ \rho_{q1}(u) \lhd_q \rho_q(s)))$. Let $\gamma(q,q1,s,u) \equiv \rho_{q1}(u) \lhd_q \rho_q(s)$. It is then desirable to lift the γ rank relation; its lifted form is $\gamma'(q,q1,s,u) \equiv (\exists v : u\xi v \ \wedge\ \phi_{q1}(v) : (\forall t : s\xi t \ \wedge\ \phi_q(t) : \rho_{q1}(v) \lhd_q \rho_q(t)))$.

4 Applications

In this section, we explore the use of proof lifting in several settings where abstraction is employed. We start with perhaps the simplest possible example: the encoding of finite types with bit-vectors in symbolic model checkers.

4.1 Symbolic Model Checking

Symbolic model checkers, such as SMV [McM93] and COSPAN [HHK96], operate on programs with finite data types. However, for the model checking computation, these programs are transformed into programs with only binary variables, by encoding variables with non-binary types with bit vectors. For instance, consider the following program, with a single action, a.

```
var x,y: [0,3]      initially (x=0) and (y=0)
a: x,y := (x+1) mod 4, (y+1) mod 4
```

To model check this program, variable x is transformed to the bit-vector (x_1, x_0), and similarly for y. This transformation induces a bisimulation between the source and result programs. Now consider the problem of lifting a proof of the bit-vector invariant, $\phi \equiv (x_0 = y_0 \ \wedge\ x_1 = y_1)$, into a proof in the original notation. The bisimulation relationship can be expressed by $(x,y)\xi((x_1,x_0),(y_1,y_0)) \equiv (x \in [0,3] \wedge y \in [0,3] \wedge x = x_0 + 2x_1 \wedge y = y_0 + 2y_1)$. Applying the lifting scheme for invariants, we get the expected result, $x = y$, as a result of Presburger simplification, which may be automated with tools such as Omega [Pug92].

$$\begin{array}{ll}
 & \phi'(x,y) \\
\equiv & \quad \langle \text{ definition of lifting } \rangle \\
 & (\exists x_0, x_1, y_0, y_1 : (x,y)\xi((x_1,x_0),(y_1,y_0)) : \phi(x_0,x_1,y_0,y_1)) \\
\equiv & \quad \langle \text{ definition of } \xi \text{, and } \phi \text{ } \rangle \\
 & (\exists x_0, x_1, y_0, y_1 : (x \in [0,3] \wedge y \in [0,3] \wedge x = x_0 + 2x_1 \wedge y = y_0 + 2y_1) : \\
 & x_0 = y_0 \wedge x_1 = y_1) \\
\equiv & \quad \langle \text{ Presburger simplification } \rangle \\
 & x \in [0,3] \wedge y \in [0,3] \wedge x = y
\end{array}$$

4.2 Predicate Abstraction

In Predicate Abstraction [GS97], predicates of the original program are represented with boolean variables in the abstract program. For a predicate p, we let $\bar{p}$ denote its corresponding boolean variable. Let $\mathcal{P} = \{p_1, \ldots, p_n\}$, $(n > 0)$, denote the set of predicates. We let $s \sim_{\mathcal{P}} t$ represent the fact that concrete state s and abstract state t agree on values of the predicates in $\mathcal{P}$, up to the predicate-boolean correspondence. Formally, $s \sim_{\mathcal{P}} t \equiv (\forall i : p_i(s) \equiv \bar{p}_i(t))$. The abstract program, $\mathcal{A}$, is computed from the concrete program, $\mathcal{C}$, while satisfying the following constraints.

- The abstract initial state, $\hat{s}_{\mathcal{A}}$, is such that $\hat{s}_{\mathcal{C}} \sim_{\mathcal{P}} \hat{s}_{\mathcal{A}}$
- The abstract transition relation, $R_{\mathcal{A}}(t,a,v)$, is such that $(\exists s, u : s \sim_{\mathcal{P}} t \wedge u \sim_{\mathcal{P}} v : R_{\mathcal{C}}(s,a,u)) \Rightarrow R_{\mathcal{A}}(t,a,v)$

These two constraints ensure that the relation $\sim_{\mathcal{P}}$ is a simulation relation from $\mathcal{C}$ to $\mathcal{A}$, relative to the predicates in $\mathcal{P}$. Now consider a proof of a universal property on the abstract program. The lifted proof computed by the recipe of the previous section can be simplified as follows.

$$\begin{array}{ll}
 & \phi'_q(s) \\
\equiv & \quad \langle \text{ by definition } \rangle \\
 & (\exists t : s \sim_{\mathcal{P}} t : \phi_q(t)) \\
\equiv & \quad \langle \text{ by definition of } \sim_{\mathcal{P}} \text{ } \rangle \\
 & (\exists t : t = (p_1(s), \ldots, p_n(s)) : \phi_q(t)) \\
\equiv & \quad \langle \text{ 1-point rule } \rangle \\
 & \phi_q(p_1(s), \ldots, p_n(s))
\end{array}$$

Thus, abstract invariants can be lifted simply by substituting each boolean variable with its corresponding predicate. The rank relations can be simplified in a similar manner, to obtain the formulas below.

- $\rho'^{=}_q(s,k) \equiv (\phi'_q(s) \wedge \rho^{=}_q(p_1(s), \ldots, p_n(s), k))$
- $\rho'^{\lhd}_q(r,s,k) \equiv (\phi'_q(s) \wedge \rho^{\lhd}_q(r, p_1(s), \ldots, p_n(s), k))$

– $\rho'^{\succeq}_q(s,k) \equiv (\phi'_q(s) \Rightarrow \rho^{\succeq}_q(r, p_1(s), \ldots, p_n(s), k))$

Similar substitution-based transformations can be obtained for other types of abstraction that are (as in this case) functional in nature: i.e., $s\xi t$ if, and only if, $t = g(s)$, for a total function g. Data type reductions (e.g., reducing the integer type to $\{negative, zero, positive\}$), and symmetry reductions (reducing an equivalence class to its representative) are other examples of functional abstractions.

Example. In the self-stabilizing program below, execution of the actions $\{a, r\}$ alone maintains the invariant that $(x \, mod \, 3 = 0)$. However, the environment, through action e, may invalidate this property. In this case, execution of the restoration action r sufficiently many times (maximum 2) restores the invariant. This restoration property can be written in the mu-calculus as $(\nu Y : (\mu Z : (x \, mod \, 3 = 0) \vee [r]Z) \wedge [a, r, e]Y)$.

```
var x:integer      initially x=0
a: x := x+3
r: not(x mod 3 = 0) -> x := x+1
e: x := choose integer
```

We can verify this property through an abstraction relative to the set of predicates $\{p_i \,|\, i \in [0..2]\}$, where $p_i \equiv (x \, mod \, 3 = i)$. One possible abstract program is as follows.

```
var b_0,b_1,b_2: boolean   initially b_0 and not(b_1) and not(b_2)
a: b_0,b_1,b_2 := b_0,b_1,b_2
r: not(b_0) -> b_0,b_1,b_2 := b_2, b_0, b_1
e: b_0,b_1,b_2 := a_0,a_1,a_2  // the inputs {a_i} are mutually exclusive
```

The transformed property is $(\nu Y : (\mu Z : b_0 \vee [r]Z) \wedge [a, r, e]Y)$. The alternating automaton for this property (each automaton state corresponds to a sub-formula), and its correctness proof are shown in Figure 2. From this proof, one can read off the lifted correctness proof as follows (the notation $(a?b|...)$ is read as "a, if b else ...").
Invariants: $\phi_{q_3} = p_0$, all other $\phi_{q_i} = true$
Rank Functions: $\rho_{q1} = (1?p_0|3?p_1|2); \rho_{q_4} = (0?p_0|2?p_1|1)$; and all other ranks are 0.

The set of states is $Q = \{q_0, \ldots, q_4\}$ with initial state q_0.
The transitions are:
$\delta(q_0, true) = q_1 \wedge q_2$,
$\delta(q_2, true) = [a, r, e]q_0$,
$\delta(q_1, true) = q_3 \vee q_4$,
$\delta(q_3, b_0) = true, \delta(q_3, \neg b_0) = false$, and
$\delta(q_4, true) = [r]q_1$
The parity condition is $(Q\backslash\{q_1\}, \{q_1\})$.

Invariants: $\phi_{q_3} = b_0$, all other $\phi_{q_i} = true$
Rank Functions: $\rho_{q_1} = (1?b_0|3?b_1|2)$,
$\rho_{q_4} = (0?b_0|2?b_1|1)$,
all other ranks are 0.

Fig. 2. Alternating Automaton and the Correctness Proof

4.3 Efficiently Checkable Lifted Proofs

In several applications of automatically generated proofs discussed in [Nam01], [HJM+02], it is important that the proof be efficiently checkable. For programs with bounded data types, the proof that is generated is *propositional* in nature: for instance, the invariants and rank relations may be represented by BDD's. Hence, it is possible to refine the individual validity checks of a temporal proof (which are instances of a co-NP complete problem) into proofs in a sound and complete propositional proof system (see, e.g., [Men97] for examples of such proof systems). Such a proof, for instance, in a Hilbert-style system, is easy to check in polynomial time: one need only check that each step is a proper substitution into an axiom schema, or a proper inference. The expanded proof may, of course, be exponential in the length of the original assertions.

However, abstraction often transforms a program with unbounded data types into a bounded program; thus, the lifted proof is over an unbounded data space. While it may seem that this makes it difficult to generate a checkable lifted proof, that is not necessarily the case. Suppose that M is simulated by N through a relation ξ, and that Π is the proof that $N \models f$. Let Π' be the lifted proof that $M \models f$, as defined in Section 3.1. The proof of Theorem 0 shows that each one of the conditions for Π' to be a valid proof is satisfied. Those proofs use, as sub-lemmas, the (given) facts that (i) ξ is a simulation relation, and (ii) Π forms a valid proof for the abstract program. Thus, if there are available checkable proofs of these facts, then the sub-proofs in Theorem 0 form a schema into which these can be inserted to make Π' a checkable proof.

For fixed abstraction schemes, such as reduction by symmetry, or predicate abstraction, the proof that ξ is a simulation can further be decomposed into a generic schema that any relation satisfying certain conditions (e.g., those given in Section 4.2) induces a valid abstraction relationship, and a proof that a given relationship satisfies these conditions. This further simplifies the burden of generating a checkable proof, and of checking it as well, since the generic schemas may be checked in advance.

5 Conclusions

We have shown that it is possible to lift deductive proofs, quite simply, through both simulation and bisimulation abstractions, for properties written in powerful temporal logics. We have also shown that it is possible to combine checkable proofs for the validity of an abstraction and for the temporal property on the abstract program into a checkable proof for the property on the original program. We have discussed simplifications in the lifted proof induced by the form of particular kinds of abstractions. It seems quite possible that other kinds of abstractions, such as reductions due to symmetry and data independence, will also be amenable to such simplification. Whether lifted proofs are of reasonable size in practice is still an open question, but the initial results of the BLAST project [HJM+02] are encouraging.

The lifted rank functions have the same domain as the abstract rank functions. This suggests that using the types of abstractions discussed here with finite-state abstract programs, one can only prove properties with 'bounded progress' proofs (as in the self-stabilizing example). Whether this is a limitation in practice remains to be seen. However, it is possible to abstract with unbounded progress measures, as shown in [KP00]; the extension of their results from linear time to branching time properties is the subject of ongoing work.

Acknowledgements: Thanks go to Dennis Dams, Patrice Godefroid, and the referees for a careful reading and many useful suggestions. Patrice Godefroid suggested the application to Modal Transition Systems.

References

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, volume 2102 of *LNCS*, 2001.

[BCG88] M. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59, 1988.

[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*. Springer-Verlag, 1981.

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

[DS90] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer Verlag, 1990.

[EJ91] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *FOCS*, 1991.

[EL86] E.A. 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*, volume 697 of *LNCS*, 1993.

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

[HHK96] R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *CAV*, volume 1102 of *LNCS*, 1996.

[HJM+02] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV*, volume 2404 of *LNCS*, 2002.

[HJS01] M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: a foundation for three-valued program analysis. In *ESOP*, number 2028 in LNCS, 2001.

[HM85] M. Hennessy and R. Milner. Algebriac laws for nondeterminism and concurrency. *J.ACM*, 1985.

[JW95] D. Janin and I. Walukiewicz. Automata for the modal mu-calulus and related results. In *MFCS*, volume 969 of *LNCS*, 1995.

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

[Koz82] D. Kozen. Results on the propositional mu-calculus. In *ICALP*, volume 140 of *LNCS*, 1982.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1), 2000.

[LT88] K.G. Larsen and B. Thomsen. A modal process logic. In *LICS*, 1988.

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

[Men97] E. Mendelson. *Introduction to Mathematical Logic*. Chapman and Hall (4th Edition), 1997.

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

[Nam01] K. S. Namjoshi. Certifying model checkers. In *CAV*, volume 2102 of *LNCS*, 2001.

[NL96] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, 1996.

[Par81] D. Park. *Concurrency and automata on infinite sequences*, volume 154 of *LNCS*. Springer Verlag, 1981.

[PPZ01] D. Peled, A. Pnueli, and L. D. Zuck. From falsification to verification. In *FSTTCS*, volume 2245 of *LNCS*, 2001.

[Pug92] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *CACM*, 35(8), 1992. web page: `http://www.cs.umd.edu/projects/omega/omega.html`.

[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.

[RRR00] A. Roychoudhury, C.R. Ramakrishnan, and I.V. Ramakrishnan. Justifying proofs using memo tables. In *PPDP*, 2000.

[SRRS02] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Model-carrying code (MCC): A new paradigm for mobile-code security. In *New Security Paradigms Workshop*, 2002.

[Sti95] C. Stirling. Modal and temporal logics for processes. In *Banff Higher Order Workshop*, volume 1043 of *LNCS*. Springer Verlag, 1995.

[TC02] L. Tan and R. Cleaveland. Evidence-based model checking. In *CAV*, volume 2404 of *LNCS*, 2002.

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