Eager Abstraction for Symbolic Model Checking

Kenneth L. McMillan

Microsoft Research

Abstract. We introduce a method of abstraction from infinite-state to finite-state model checking based on eager theory explication and evaluate the method in a collection of case studies.

1 Introduction

In constructing decision procedures for arithmetic formulas and other theories, a successful approach has been to separate propositional reasoning and theory reasoning in a modular way. This approach is usually called Satisfiability Modulo Theories, or SMT [1]. There are two primary approaches to SMT: *eager* and *lazy* theory explication. Both approaches abstract the formula in question by constructing its propositional skeleton, that is, converting each atomic predicate to a corresponding free Boolean variable. Obviously, propositional abstraction loses a great deal of information. The eager approach compensates for this by conjoining tautologies of the theory to the formula before propositional abstraction. In abstract interpretation terms, we can think of this as a *semantic reduction*: it makes the formula more explicit without changing its semantics. The lazy approach, on the other hand, performs the propositional abstraction first, then retroactively adds tautologies of the theory to rule out infeasible propositional models.

In this paper, we will consider applying the same concepts to the symbolic model checking problem (SMC). In this problem, we are given a Kripke model M that is expressed implicitly using logical formulas, and a temporal formula ϕ, and we wish to determine whether $M \models \phi$. The states of the Kripke model are structures of a logic L over a given vocabulary, while the set of initial states I and the set of transitions T are expressed, respectively, by one- and two-vocabulary formulas. The atomic propositions in ϕ are also presumed to be expressed in L.

In the case where L is propositional logic, the Kripke model is finite-state, the SMC problem is PSPACE-complete, and many well-developed techniques are available to solve it in a heuristically efficient way. On the other hand, if L is a richer logic (say, Presburger arithmetic) SMC is usually undecidable. Here, we propose to solve instances of this problem by separating propositional reasoning and theory reasoning in a modular way, as in SMT. Given an SMC problem (I, T, ϕ), we will form its propositional abstraction by computing the propositional skeletons of I, T and ϕ. This abstraction is sound, and allows us to apply well-developed tools for propositional SMC, however it loses a great deal of information. To compensate for this loss, we will use incomplete eager theory explication. By controlling theory explication, the user controls the abstraction. We will call this general approach *eager symbolic model checking*, or ESMC.

Related work Because of the propositional abstraction, ESMC may at first seem to be a form of predicate abstraction [9]. This is not the case, however. Predicate abstraction uses a vocabulary of predicates to abstract the state, but does not abstract the theory itself. As a result, a decision procedure for the theory is needed to compute the best abstract transformer. This is problematic if the logic is undecidable, and in any event requires an exponential number of decision procedure calls in the worst case. In ESMC, the abstraction is performed in a purely syntactic way. One controls the abstraction by giving a set of axiom schemata to be instantiated and by introducing prophecy variables, as opposed to giving abstraction predicates. One effect of this is that the abstraction may depend on the precise syntactic expression of the transition relation.

The technique of "datatype reductions" [18] is also closely related. This method has been used to verify various parameterized protocols and microarchitectures using finite-state model checking [6, 5, 20, 12, 19]. The technique also abstracts an infinite-state SMC problem to a finite-state one syntactically. Though it does not do this by explicating the theory, we will see that the abstraction it produces can be simulated by ESMC. Compared to this method, ESMC is user-extensible and allows both a simpler theoretical account and a simpler implementation. Moreover, it uses a smaller trusted computing base, since the tautologies it introduces can be mechanically checked.

The methods of Invisible Invariants [25] and Indexed Predicate Abstraction [14] use different methods to compute the least fixed point in a finite abstract domain of quantified formulas. This requires decidability and incurs a relatively high cost for computing an extremal fixed point, limiting scalability (though IPA can approximate the best transformer in the undecidable case). The abstractions are also difficult to refine in practice.

Road map After preliminaries in the next section, we introduce our schema-based class of abstractions in Section 3. The next section gives some useful instantiations of this class. Section 5 describes a methodology for exploiting the abstraction in proofs of infinite-state systems, as implemented in the IVy tool. In Section 5, we evaluate the approach using case studies.

2 Preliminaries

Let $FO_=(\mathbb{S}, \Sigma)$ be standard sorted first-order logic with equality, where $\mathbb{S}$ is a collection of first-order sorts and Σ is a vocabulary of sorted non-logical symbols. We assume a special sort $\mathbb{B} \in \mathbb{S}$ that is the sort of propositions. Each symbol $f^S \in \Sigma$ has an associated sort S of the form $D_1 \times \cdots \times D_n \to R$, where $D_i, R \in \mathbb{S}$ and $n \geq 0$ is the *arity* of the symbol. If $n = 0$, we say f^S is a *constant*, and if $R = \mathbb{B}$ it is a *relation*. We write $\text{vocab}(t)$ for the set of non-logical symbols occurring in term t.

Given a set of sorts $\mathbb{S}$, a *universe* U maps each sort in $\mathbb{S}$ to a non-empty set (with $U(\mathbb{B}) = \{\top, \bot\}$). An *interpretation* of a vocabulary Σ over universe U maps each symbol $f^{D_1 \times \cdots \times D_n \to R}$ in Σ to a function in $U(D_1) \times \cdots \times U(D_n) \to U(R)$. A Σ-structure is a pair $\mathcal{M} = (U, \mathcal{I})$ where U is a universe and $\mathcal{I}$ is an interpretation of Σ over U. The structure is a *model* of a proposition ϕ in $FO_=(\mathbb{S}, \Sigma)$ if ϕ evaluates to $\top$ under $\mathcal{I}$ according to the standard semantics of first-order logic. In this case, we write $\mathcal{M} \models \phi$. Given an interpretation $\mathcal{J}$ with domain disjoint from $\mathcal{I}$, we write $\mathcal{M}, \mathcal{J}$ to abbreviate the structure $(U, \mathcal{I} \cup \mathcal{J})$.

In the sequel, we take the vocabulary Σ to be a disjoint union of four sets: Σ_S, the *state* symbols, Σ'_S the *primed* symbols, Σ_T the *temporary* symbols, and Σ_B, the *background* symbols. We take $(\cdot)'$ to be a bijection $\Sigma_S \to \Sigma'_S$ and extend it in the expected way to terms and interpretations. We write $\text{unprime}(t)$ for the term u such that $u' = t$, if u exists.

A *transition system* is a pair (I, T) where I is a proposition over $\Sigma_S \cup \Sigma_B$ and T is a proposition over Σ. Let $\mathcal{M}_B = (U, \mathcal{I}_B)$ be a Σ_B-structure (that is, fix the universe and the interpretation of the background symbols). A *U-state* of the system is an interpretation of Σ_S (the state symbols) over U. A *$\mathcal{M}_B$-run* of the system is an infinite sequence $s_0, s_1, \ldots$ of U-states such that:

- $\mathcal{M}_B, s_0 \models I$, and
- for all $0 \leq i$, there exists and interpretation $\mathcal{I}_T$ of Σ_T over U such that $\mathcal{M}_B, s_i, \mathcal{I}_T, s'_{i+1} \models T$.

That is, under the background interpretation, the initial state must satisfy the initial condition, and for every successive pair of states, there must be an interpretation of the temporary symbols such that the transition condition is satisfied. The temporary symbols are used, for example, to model local variables of procedures, and may also be Skolem symbols. Because they can have second-order sort, we cannot existentially quantify them within the logic, so instead we quantify them implicitly in the transition system semantics. Given a background theory $\mathcal{T}$ over Σ_B, a *$\mathcal{T}$-run* is any $\mathcal{M}_B$-run such that $\mathcal{M}_B \models \mathcal{T}$.

A *linear temporal formula* over Σ applies the operators of $FO_=(\mathbb{S}, \Sigma)$ plus the standard strict until operator $\mathcal{U}$ and strict since operator $\mathcal{S}$. We define $\bigcirc\phi = \bot\,\mathcal{U}\,\phi$, $\Box\phi = \phi \wedge \neg(\top\,\mathcal{U}\,\neg\phi)$ and also $\mathcal{H}\phi = \phi\,\mathcal{S}\,\bot$, meaning "always ϕ in the strict past". We fix $\mathcal{T}$ and say $(I, T) \models \phi$ if every $\mathcal{T}$-run of (I, T) satisfies ϕ under the standard LTL semantics. The symbolic model checking problem SMC is to determine whether $(I, T) \models \phi$.

3 A schema-based abstraction class

An *atom* is a proposition in which every instance of $\{\wedge, \vee, \neg, \mathcal{U}, \mathcal{S}\}$ occurs under a quantifier. The *propositional skeleton* of a proposition ϕ is obtained by replacing each atom in ϕ by a corresponding propositional constant. The propositional skeleton is an abstraction, in the sense that for every model M of ϕ we can construct a model of its propositional skeleton from the truth values of each atomic proposition in M. We will use propositional skeletons here to convert an infinite-state model checking problem to a finite-state one.

We assume that each vocabulary Σ_B, Σ_S and Σ_T contains a countably infinite set of propositional constants. This allows us to construct injections $\mathcal{A}_B$, $\mathcal{A}_S$, $\mathcal{A}_T$ from atomic propositions of the logic to propositional constants in Σ_B, Σ_S and Σ_T respectively.

In defining the propositional skeleton of a transition formula we must consider atomic propositions containing symbols from more than one vocabulary. To which vocabulary should we map such an atom in the propositional skeleton? Here, we take a simple solution that is sound, though it may lose some state information. That is, for any atomic proposition ϕ, we say

– if $\text{vocab}(\phi) \subseteq \Sigma_B$, then $\mathcal{A}(\phi) = \mathcal{A}_B(\phi)$,
– else if $\text{vocab}(\phi) \subseteq \Sigma_B \cup \Sigma_S$ then $\mathcal{A}(\phi) = \mathcal{A}_S(\phi)$
– else if $\text{vocab}(\phi) \subseteq \Sigma_B \cup \Sigma'_S$ then $\mathcal{A}(\phi) = \mathcal{A}_S(\text{unprime}(\phi))'$
– else $\mathcal{A}(\phi) = \mathcal{A}_T(\phi)$

That is, pure background propositions are abstracted to background symbols, state propositions are abstracted to state symbols and next-state propositions are abstracted to the primed version of the corresponding state proposition. Everything else is abstracted to a temporary symbol (which is existentially quantified in the abstract transition relation).

We then extend $\mathcal{A}$ to non-atomic formulas in the obvious way, such that $\mathcal{A}(\phi \wedge \psi) = \mathcal{A}(\phi) \wedge \mathcal{A}(\psi)$, $\mathcal{A}(\bigcirc\phi) = \bigcirc\mathcal{A}(\phi)$ and so on. The following theorem shows that we can use propositional skeletons to convert infinite-state to finite-state model checking problems in a sound (but incomplete) way:

Theorem 1. *For any symbolic transition system (I, T) and linear temporal formula ϕ, if $(\mathcal{A}(I), \mathcal{A}(T)) \models \mathcal{A}(\phi)$ then $(I, T) \models \phi$.*

Intuitively, this holds because we can convert every concrete counterexample to an abstract one by simply extracting the truth values of the atomic propositions.

Theory explication While propositional skeletons are sound, they lose a great deal of information. For example, suppose our transition relation is $y' = x$. Given a predicate p, we would like to infer that $p(x) \Rightarrow \bigcirc p(y)$. However, in the propositional skeleton, the transition relation $\mathcal{A}(T)$ is just $\mathcal{A}_T(y' = x)$. In other words, it is just a free propositional symbol with no relation to any other proposition. Thus, we cannot prove the abstracted property $\mathcal{A}(p(x)) \Rightarrow \bigcirc\mathcal{A}(p(y))$.

To mitigate this loss of information, we use *theory explication*. That is, before abstracting T, we conjoin to it tautologies of the logic or the background theory. This doesn't change the semantics of T, and thus the set of runs of the transition system remains unchanged. It does, however, change the propositional skeleton. For example, $y' = x \wedge p(x) \Rightarrow p(y')$ is a tautology of the theory of equality. If we conjoin this formula to T in the above example, the abstract transition relation becomes $\mathcal{A}_T(y' = x) \wedge (\mathcal{A}_T(y' = x) \wedge \mathcal{A}_S(p(x)) \Rightarrow \mathcal{A}_S(p(y))')$ which is strong enough to prove the abstracted property.

In general, theory explication adds predicates to the abstraction. This is the only mechanism we will use to add predicates; we will not supply them manually, or obtain them automatically from counterexamples. The following theorem justifies model checking with eager theory explication:

Theorem 2. *For any symbolic transition system (I, T), linear temporal formula ϕ, $\Sigma_B \cup \Sigma_S$ formula ψ_I and Σ formula ψ_T, if $\mathcal{T} \models \psi_I \wedge \psi_T$ then $(I \wedge \psi_I, T \wedge \psi_T) \models \phi$ iff $(I, T) \models \phi$.*

The question, of course, is how to choose the tautologies in ψ_I and ψ_T. This is not just a question of capturing the transition relation semantics, since theory explication also determines the FO predicates representing state of the finite abstraction. Thus, complete theory explication is at least as hard as predicate discovery in predicate abstraction. Our goal is not to solve this problem, but to find an effective incomplete strategy that is useful in practice. It is important that the resulting finite-state model checking problems be easily resolved by a modern model checker, and that in case the strategy fails, a human can use the resulting counterexample and effectively refine the abstraction.

Schema-based theory explication The basic approach we will use to controlling theory explication is a restricted case of the pattern-based quantifier instantiation method introduced in the Simplify prover [8]. That is, we are given a set of axioms, and for each axiom a set of triggers. A trigger is a term (or terms) containing all of the free variables in the axiom. The trigger is matched against all ground subterms in the formula being explicated. Each match induces an instance of the axiom.

In our example above, suppose we have the axiom $Y = X \wedge p(X) \Rightarrow p(Y)$ with a trigger $Y = X$ (here and in the sequel, capital letters will stand for free variables). The trigger $Y = X$ matches the ground term $y' = x$ in T which generates the ground instance $y' = x \wedge p(x) \Rightarrow p(y')$. Since we match modulo the symmetry of equality, we also get $x = y' \wedge p(y') \Rightarrow p(x)$.

A risk of trigger-based instantiation is the matching loop. For example, if we have the axiom $f(X) > X + 1$ with a trigger $f(X)$, then we can generate an infinite sequence of instantiations: $f(y) > y + 1$, $f(f(y)) > f(y) + 1$ and so on. A simple approach to prevent this is to bound the number of generations of matching. In practice, we will use just one generation and expand the set axioms in cases where more than one generation is needed. This has the benefit of keeping the number of generated terms small, which limits the size of the SMC problem and also makes it easier for users to understand counterexamples.

To avoid having to write a large number of axioms, we specify the axioms using general schemata. A schema is a parameterized axiom. It takes a list of sorts and symbols as parameters and yields an axiom. In the sequel we will use s and t to stand for sort parameters. As an example, here is a general congruence schema that can be used in place of our axiom above:

$$\frac{f : s \to t}{X = Y \Rightarrow f(X) = f(Y) \quad \{X = Y\}}$$

The trigger is in curly braces. We first instantiate the axiom schemata for all possible parameter valuations using the sorts and symbols of the concrete system. Then we ground the resulting axioms using pattern-based instantiation.

One further technique is needed, however, to ground the quantifiers occurring in the formula being explicated. Quantifiers usually occur in the transition relations of parameterized systems either in the guards of guarded commands or in state updates. As an example, suppose a given command sets the state of process p to 'ready'. This would appear in the transition formula as a constraint such as the following:

$$\forall x.\ \mathrm{state}'(x) = \mathrm{ready\ if\ } x = p \mathrm{\ else\ state}(x)$$

If this quantifier is not instantiated, then all information about process state will be lost. To avoid this, we would like to apply the following schema:

$$\frac{y : s,\ p : s \rightarrow \mathbb{B}}{(\forall X.\ p(X)) \Rightarrow p(y) \quad \{\forall X.\ p(X)\}}$$

Here we intend that p should match *any* predicate with one free variable and not just a predicate symbol (including non-temporal sub-formulas of the property to be proved). However, rather than implement a general second-order matching scheme, it is simpler to build this particular schema into the theory explication process. There is some question as to which ground terms to supply for the parameter y. As with other schemata, only constants are used in the current implementation. This appears to be adequate, but it might also be useful to allow the user to supply explicit triggers for quantifiers in the transition system or property.

The theory explication process thus has three steps:

1. Instantiate quantifiers in the formulas using the quantifier schema above.
2. Generate axioms from the user axiom schemata, supplying symbols from the formulas as parameters.
3. Instantiate the axioms using triggers for one generation.

Notice this is a slight departure from the policy of one generation of matching, since terms generated in step 1 can be used to match axioms in step 3. This is important in practice since without grounding the quantifiers there may be no ground terms to match in step 3.

4 Example abstractions in the class

A typical approach to verifying parameterized protocols with finite-state model checking is to track the state of a representative fixed collection of processes and abstract away the state of the remaining processes. In this approach, introduced in [17], a small collection background constants (typically two or three) is used to identify the tracked processes. For each process identifier in the system, the abstraction records whether it is equal to each of the tracked ids, but carries no further information. For each function f over process ids, the abstraction maintains the value of $f(x)$ only if x is equal to one of the background constants. This approach has been used, for example, to verify processor micro-architectures [17, 16, 12] and cache coherence protocols [6, 5, 19].

This abstraction can be implemented using schema-based instantiation. The high-level idea is to create a set of schemata that make it possible to abstractly evaluate terms in a bottom-up manner.

For example, consider an occurrence $t = u$ of the equality operator where t and u are terms of sort s. The abstract value of this term is $\top$ if t and u are both equal to some background constant c, $\bot$ if $t = c$ and $u \neq c$, and otherwise is unknown. To implement this abstraction, we use the following schemata:

$$\frac{c : s}{X = c \wedge Y = c \Rightarrow X = Y \;\; \{X = Y\}} \qquad \frac{c : s}{X = c \wedge Y \neq c \Rightarrow X \neq Y \;\; \{X = Y\}}$$

The triggers of these two schemata cause them to be applied to every occurrence of an equality operator in the formula being abstracted.

For an application $f(t)$ of a function symbol, the abstract value is the abstraction of $f(c)$ if t is equal to background constant c, and is otherwise unknown. This fact could be captured by chaining the congruence schema above with the above two equality schemata. That is, matching the congruence schema, we obtain $t = c \Rightarrow f(t) = f(c)$. Then matching the equality operator schemata with this result, we obtain (in the contrapositive) $f(t) = f(c) \wedge f(c) = d \Rightarrow f(t) = d$ and $f(t) = f(c) \wedge f(c) \neq d \Rightarrow f(t) \neq d$ (for any background constants c, d). Recall, however, that we allow only one generation of matching, so this second matching step will not occur. Instead, we write the above two facts explicitly as a schema:

$$\frac{c : s,\; d : t,\; f : s \to t}{X = c \Rightarrow (f(X) = d \Leftrightarrow f(c) = d) \;\; \{f(X)\}}$$

This schema is matched for every application of a symbol of arity one in the formula. We also specify similar schemata for arities greater than one. Notice that this schema also applies to relation symbols if we treat $\top$ and $\bot$ as background constants of sort $\mathbb{B}$. However, for relations and functions to finitely enumerated sorts, it is more efficient to use the congruence schema, since it produces fewer instances.

Finally, we need one additional schema to guarantee that the abstract values are consistent with the equality relation on the background constants:

$$\frac{c : s,\; d : s}{X = c \Rightarrow (X = d \Leftrightarrow c = d) \;\; \{X\}}$$

Notice that this axiom is instantiated for every term in the formula (though in practice not for propositions). Though it doesn't affect satisfiability of formulas, it is also helpful to add reflexivity, symmetry and transitivity over the background constants as it makes the resulting counterexamples easier to understand.

These schemata produce an abstraction of the formula that is at least as strong as the datatype reduction for scalarset types described in [18]. In fact, this is true if we restrict the application of the schemata to constants c and d in the set of background constants, which we do in practice. The cost of the abstraction is moderate, since the number of axiom instances is directly proportional to the size of the formula and to the number of background constants.

An advantage of the schema-based explication approach is that we can use it to construct abstractions for various datatypes and even use different abstractions of the same datatype for different applications. As an example, consider an abstraction for totally ordered datatypes such as the integers. We want the abstraction to track, for any term t of this sort, whether it is equal to, less than or greater than each background constant. The abstract value of a term t is captured by the values of the predicates $t < c$ and $t = c$ for background constants c. We begin with the abstract semantics of equality given above. The abstract semantics of the $<$ relation can be given by the following schemata (where $t \leq c$ is an abbreviation for $t < c \vee t = c$):

$$\frac{c : s}{X \leq c \wedge c < Y \Rightarrow X < Y \quad \{X < Y\}} \qquad \frac{c : s}{X < c \wedge c \leq Y \Rightarrow X < Y \quad \{X < Y\}}$$

$$\frac{c : s}{Y \leq c \wedge \neg(X < c) \Rightarrow \neg(X < Y) \quad \{X < Y\}}$$

By chaining the congruence schema with these, we can obtain the abstract semantics of function application, but again we wish to limit the number of matching generations to one. Thus, as with equality, we write an explicit schema combining the two steps:

$$\frac{c : s,\ d : t,\ f : s \rightarrow t}{X = c \Rightarrow (f(X) < d \Leftrightarrow f(c) < d) \quad \{f(X)\}}$$

We also require that the abstract value of every term be consistent with the interpretation of $=$ and $<$ over the background constants. This gives us:

$$\frac{c : s}{\neg(X = c \wedge X < c) \quad \{X\}} \qquad \frac{c : s,\ d : t}{X \leq d \wedge \neg(X < c) \Rightarrow c \leq d \quad \{X\}}$$

With the equality schemata, these imply that the background constants are totally ordered. As an extension, if the totally ordered sort has a least element 0, we can add it as a background constant along with the axiom $\neg(X < 0)$.

This abstraction is a bit weaker than the "ordset" abstraction used, for example, in [20]. We can simulate that abstraction by adding schemata that interpret the $+$ operator, and facts about numeric constants such as $0 < 1$. In general, for a given datatype, we can tailor an abstraction that captures just the properties of that type needed to prove a given system property. This extensibility makes the schema-based approach more flexible and possibly more efficient than the built-in abstractions of [18]. The above schemata have been verified by Z3.

5 Proof methodology

In the previous sections, we developed an approach to produce a sound finite-state abstraction of an infinite-state system using eager theory explication and propositional skeletons. Now we consider how to construct proofs of systems using this approach. This section is essentially a summary of some results in [18].

The first question that arises is how to obtain the set of background constants that determine the abstraction. Generally speaking these arise as prophecy variables. For example, suppose we wish to prove a mutual exclusion property of the form $\Box\forall x, y.\ p(x) \wedge p(y) \Rightarrow x = y$. To do this, we replace the bound variables x and y with fresh background constants a and b, to obtain the quantifier-free property $\Box p(a) \wedge p(b) \Rightarrow a = b$. In effect a and b are immutable prophecy variables that predict the values of x and y for which the property will fail. By introducing prophecy variables, we refine the abstraction so that it tracks the state of the pair of processes that ostensibly cause the mutual exclusion property to fail. We hope, of course, to prove that there are no such processes. We apply the following theorem to introduce prophecy variables soundly:

Theorem 3. *Let (I, T) be a symbolic transition system, $x{:}s$ a variable, $\phi(x)$ a temporal formula and $v{:}s$ a background symbol not occurring in I, T, ϕ. Then $(I, T) \models \Box\forall x.\ \phi(x)$ iff $(I, T) \models \Box\phi(v)$.*

This theorem can be applied as many times as needed to eliminate universal quantifiers from an invariance property. Further refinement can be obtained if needed by manually adding prophesy variables. For example, suppose that each process x has a ticket number $t(x)$, and we wish to track the ticket number held by process a at the time of the failure. To do this, we replace our property with the property $\Box\ c = t(a) \Rightarrow (p(a) \wedge p(b) \Rightarrow a = b)$ where c is a fresh background constant. In general, we can introduce additional prophecy variables using this theorem:

Theorem 4. *Let (I, T) be a transition system, ϕ a temporal formula and t a term. Then $(I, T) \models \Box\phi$ iff $(I, T) \models \Box\forall x.\ x = t \Rightarrow \phi$, where x is not free in ϕ.*

The theorem can be applied repeatedly to introduce as many prophecy variables as needed to refine the abstraction. The introduced quantifiers can be converted to background symbols by the preceding theorem.

Since our abstraction tracks the state of only processes a and b, a protocol step in which an untracked process sends a message to a or b is likely to produce an incorrect result in the abstraction. To mitigate this problem, we assume by induction over time that our universally quantified invariant property ϕ has always held in the strict past. This makes use of the following theorem:

Theorem 5. *Let (I, T) be a symbolic transition system, and ϕ a temporal formula. Then $(I, T) \models \Box\phi$ iff $(I, T) \models \Box\ (\mathcal{H}\phi) \Rightarrow \phi$.*

The quantifiers in ϕ will be instantiated with ground terms in T. Thus, in our mutual exclusion example, we can rely on the fact that the sender of a past message (identified by some temporary symbol) is not in its critical section if either a or b are. Using induction in this way can mitigate the loss of information in the finite abstraction. Note we can pull quantifiers out of the above implication in order to apply Theorem 3. That is, $(\mathcal{H}\forall x.\ \phi) \Rightarrow \forall x.\phi$ is equivalent to $\forall x.\ (\mathcal{H}\forall x.\ \phi) \Rightarrow \phi$.

If the above tactics fail to prove an invariant property because the abstraction loses too much information, we can strengthen the invariant by adding conjuncts to it. These conjuncts have been called "non-interference lemmas", since they serve to reduce the interference with the tracked processes that is caused by loss of information about the untracked processes. We use the following theorem:

Theorem 6. *Let (I, T) be a symbolic transition system, and ϕ, ψ temporal formulas. Then if $(I, T) \models \Box\phi \wedge \psi$ then $(I, T) \models \Box\phi$.*

The general proof approach has the following steps:

1. Strengthen the invariant property (manually) with Theorem 6.
2. Apply temporal induction with Theorem 5.
3. Add quantifiers to the invariant with Theorem 4.
4. Convert the invariant quantifiers to background symbols with Theorem 3.
5. Add tautologies to the system using Theorem 2 and specified schemata.
6. Abstract to a finite-state SMC problem using Theorem 1.
7. Apply a finite-state symbolic model checker to check the property.

Implementation in IVy This approach has been implemented in the IVy tool [15]. In IVy, the state of the model is expressed in terms of mutable functions and relations over primitive sorts. The language is procedural, and allows the expression of protocol models as interleavings of atomic guarded commands, the semantics of which is expressible in first-order logic.

To implement the approach, IVy's language was augmented with a syntax for expressing schemata. The schemata of Section 4 were added to the tool's standard library. Syntax is also provided to decorate invariant assertions with terms to be used as prophecy variables. IVy extends the above theory slightly by allowing invariant properties to be asserted not only between commands, but also in the middle of sequential commands. This can be convenient, since it allows invariants to reference local variables inside the commands.

With this input, the tool applies the six transformation steps detailed above to produce a purely propositional SMC problem. This problem is then converted to the AIGER format [2], a standard for hardware model checking. At present, the system only handles safety properties of the form $\Box(\mathcal{H}\phi) \Rightarrow \phi$, where ϕ is non-temporal. The AIGER format does support liveness, however, and this is planned as a future extension.

The resulting AIGER file is passed to the tool ABC [4] which uses its implementation of property driven reachability [10] to check the property. The counterexample, if any, is converted back to a run of the abstract transition system. The propositional symbols in this run are converted back to the corresponding atoms by inverting the abstraction mapping $\mathcal{A}$. This yields an *abstract* *counterexample*: a sequence of predicate valuations that correspond to both the state and temporary symbols in the abstraction.

The abstract counterexample may be spurious in the sense that it corresponds to no run of the concrete transition system. In this case, the user must analyze the trace to determine where necessary information was lost and either modify the invariant or refine the abstraction by adding a prophecy variable.

6 Case studies

In this section, we consider the proof of safety properties of four parameterized algorithms and protocols. We wish to address three main questions. First, is the abstraction approach efficient? That is, if we construct an abstract model using schema-based theory explication, can the resulting finite-state problem be solved using a modern symbolic model checker? Second, is the methodology usable? That is, can a human user construct a proof using the methodology by analyzing the abstract counterexamples? Third, when is it more effective than the current best alternative, which is to write an inductive invariant manually and check it using an SMT solver, as in [11]? We will call this approach "invariant checking". We note that predicate abstraction is not suitable to these examples because the invariants require complex quantified formulas while current methods that synthesize quantified invariants for parameterized systems are unreliable in practice and do not scale well.

The last question in particular has not been well addressed in prior work on model checking approaches to parameterized verification. In most cases, either no comparison was made, or comparison was made to proofs using general-purpose proof assistants, which tend to be extremely laborious and do not make use of current state-of-the art proof automation techniques. To make a reasonably direct comparison, we construct proofs of each model using both methodologies, using the same language and tool, using the state-of-the art tools ABC [4] for model checking and Z3 [7] for invariant checking.

To apply the invariant checking method, some of the protocol models have been slightly re-encoded. In particular, it is helpful in some cases to use relations rather than functions in modeling the protocol state, as this can prevent the prover from diverging in a "matching loop" [8]. This re-encoding adds negligibly to the proof effort and is arguably harmless, since it does not appear in practice to affect the difficulty of refining the model to a concrete implementation.

Our four example models are:

1. **Tomasulo**: a parameterized model of Tomasulo's algorithm for out-or-order instruction execution, taken from [17].
2. **German**: a model of a simple directory-based cache coherence protocol from [6].
3. **FLASH**: a model of a more complex and realistic cache coherence protocol from [23, 19], based on the Stanford FLASH multiprocessor [13].
4. **VS-Paxos**: a model of Virtually Synchronous Paxos [3], a distributed consensus algorithm, from [21].

		model checking					invariant checking			
model	size	|Inv|	HVars	PVars	|Pf|	time	|Inv|	HVars	|Pf|	time
Tomasulo	1245	100	6	11	248	0.39	318	5	398	2.4
German	754	23	1	0	29	0.60	234	1	240	1.8
FLASH	2427	81	3	2	122	69	1235	1	1255	9.1
VS-Paxos	1442	224	8	34	512	23	1022	2	1101	59

Table 1. Comparison of proofs using two methodologies.

A comparison of the proofs obtained using the two methodologies is shown in Table 1. The column "size" shows the textual size of the model plus property in lexical tokens. The columns labeled |Inv| give the size of the auxiliary invariants used in the proofs, expressed in the number of lexical tokens not including the property to be proved. Since both methods require the user to supply auxiliary invariants and discovering this invariant is the largest part of the effort in both cases, this number provides a fairly direct comparison of the complexity of the proofs. In both methodologies, the user also defines history or "ghost" variables that help in expressing the invariant. The number of these variables is shown in the columns labeled HVars. In the model checking approach, the user also refines the abstraction by defining prophecy variables. These were not used in the invariant checking proofs. The closest analogy in invariant checking proofs to this type of information would be quantifier instantiations or triggers provided by the user. This was not needed, however, since the methodology of [22] was applied to ensure that all verification conditions reside in a decidable fragment of the logic. For the model checking methodology, the number of distinct terms supplied by the user as prophecy variables is shown in the column labeled PVars. The time columns show the total time in seconds for model checking or invariant checking for the completed proofs on a 2.6 GHz Intel Xeon CPU using one core. Times to produce counterexamples were generally faster.

When measuring the overall complexity of the proofs, it is unclear how to weight the three kinds of information supplied by the user. In a sense, prophecy variables are the easiest to handle, since their behavior is monotone. That is, adding a prophecy variable only increases precision so it cannot cause passing invariants to fail. Ghost variables are more conceptually difficult to introduce, since the invariants depend on them. If a ghost variable definition is changed to repair a failing invariant, this may cause a different invariant to fail. Similarly if we strengthen a passing invariant, it may fail to be proved and if we weaken a failing one it may cause other formerly passing invariants to fail. This instability can cause the manual proof search to fail to converge and is the chief cause of conceptual difficulty in constructing proofs in both methodologies. Having said this, for lack of a principled way to weight the different aspects of the proof effort, we will measure the proof size as simply the sum of the number of lexical tokens in the auxiliary invariant, the history variable definitions, and all terms used as prophecy variables. The total proof size is shown in the columns labeled |Pf|.

These numbers should be taken as unreliable for several reasons that are common to any attempt to measure the effectiveness of a proof methodology. First, the size of the proof (or any other measure of the proof difficulty, such as expended time) can depend on the proficiency of the user in the particular methodology. Even if the same user produces both proofs, the user's proficiency in the two methodologies may differ, and knowledge gained in the first proof will effect the second one. Since resources were not available to train and test a statistically significant population users in both methodologies (assuming such could be found) the numbers presented here should not be considered a direct comparison of the methods. Rather, they are presented to support some observations made below about the specific case studies and proofs.

Case study: Tomasulo's algorithm This is a simple abstract model of a processor microarchitecture that executes instructions concurrently out of order. The model state consists of a register file, a set of reservation stations (RS) and a set of execution units (EU) and is parameterized on the size of each of these, as well as the data word size. The machine's instructions are register-to-register and are modeled abstractly by an uninterpreted function. Each register has a flag that records whether it is the destination of a pending instruction. If so, its tag indicates which RS is holding that instruction. Each RS stores the tags of its instruction arguments, and waits for these to be computed before issuing the instruction to an EU.

Both proofs are based on history variables that record the correct values of arguments and result for each RS. The principal invariant of both states that the arguments obtained by all RS's are correct. In the model checking case, the abstraction is refined by making the tags of these arguments and chosen EU into prophecy variables. This allows the model checker to track enough state information to prove the main invariant, though one additional "non-interference" lemma is needed to guarantee that other EU's do not interfere by producing an incorrect tag. An interesting aspect of the invariant is that it does not refer to the states of the register file or EU's. The necessary invariants of these structures can be inferred by the model checker. On the other hand, this information must be supplied explicitly in the manual invariant. As the table shows, the resulting invariant is more complex.

Case study: German's cache protocol This simple distributed directory-based cache coherence protocol allows the caches to communicate directly only with the directory. The property proved is coherence, in effect that exclusive copies are exclusive. In the model checking proof, there is one non-interference lemma, stating that no cache produces a spurious invalidation acknowledgment message. No extra prophecy variables are need, as tracking the state of just the two caches that produce the coherence failure suffices. The manual invariant on the other hand is much more detailed, in fact about an order of magnitude larger. This is because it must relate the state of all the various types of messages in the network to the cache and directory states. These relationships were inferred automatically by the model checker, resulting in a much simpler proof.

Case study: FLASH cache coherence protocol This is a much more complex (and realistic) distributed cache coherence protocol model. The increased protocol complexity derives from the fact that information can be transferred directly from one cache to another. In a typical transaction, a cache sends a request to the directory for (say) an exclusive copy of a cache line. The directory forwards the request to the current owner of the line, which then sends a copy to the original requester, as well as a response to the directory confirming the ownership transfer. Handling various race conditions in this scheme makes both the protocol and its proof complex. Again the property proved is coherence. The model checking proof is similar to [19], though there data correctness and liveness were proved.

In this case, three non-interference lemmas are used in the model checking proof, ruling out three types of spurious messages. Also two additional prophecy variables are needed. For example, one of these identifies the cache that sent an exclusive copy. This allows the abstraction to track the state of the third participant in the triangular transaction described above. Generally, protocols with more complex communication patterns require more prophecy variables to refine the abstraction.

As with German's protocol, and for the same reason, the manual invariant is an order of magnitude larger. In this case, the additional protocol complexity makes it quite challenging to converge to an invariant and a large number of strengthenings and weakenings were needed.

Case study: Virtually Synchronous Paxos This is a high-level model of a distributed consensus protocol, designed to allow a collection of processes to agree on a sequence of decisions, despite process and network failures. This model was previous proved by a manual invariant to be consistent, meaning that two decisions for a given index never disagree [21].

The protocol operates in a sequence of epochs, each of which has a leader process. The leader proposes decision values and any proposal that receives votes of a majority of processes becomes a decision. When the leader fails the protocol must move on to a new epoch. For consistency, any decisions that are possibly made in the old epoch must be preserved in the new. This is accomplished by choosing a majority of processes to start the new epoch and preserving all of their votes. Any decision having a majority of votes in the old epoch must have one voter in the new epoch's starting majority and thus must be preserved. The choice of an epoch's starting majority is itself a single-decree consensus problem. This is solved in a sequence of rounds called "stakes". A stake can be created by a majority of processes and proposes the votes of some majority to be carried to the next epoch. Each process in the stake promises not accept any lesser stake with differing votes. If a majority accepts the stake, then the votes of that stake can be passed to the next epoch.

The important auxiliary invariants of the model checking proof are these:

– At each epoch, the votes of the majority that ends the epoch are known to the leaders of all future epochs, and
– When a stake is created, every lesser stake with different votes is “dead” in the sense that a majority of nodes has promised not to accept it, and
– In any epoch, any two accepted stakes agree on their votes.

Perhaps not surprisingly, the manual invariant is much larger. The model checking proof, however, requires many extra prophecy variables. This is mainly accounted for by the fact that the model has seven unbounded sorts: process id’s, decision indices, decision values, epochs, stakes, vote sets and process sets. Typically each invariant (including the one to be proved) requires one or two prophecy variables of each sort to refine the abstraction (though some of these may not be unique).

An additional complication is dealing with sets and majorities. Sets of processes are represented by an abstract data type. This type provides a predicate called ‘majority’ that indicates that a set contains more than half of the process id’s. A function ‘common’ returns a common element between two sets if both are majorities (and is otherwise undefined). For example, to prove that we cannot have two conflicting decisions, we use the majorities that voted for each decision and declare the common process between these majorities as a prophecy variable. It then suffices to show that this particular process cannot have voted for both decisions (which requires the auxiliary invariants above). Since majorities are used in several places in the protocol, this tactic is applied several times.

Because of the larger number of prophecy variables, our (admittedly arbitrary) measure of overall proof complexity does not show as much advantage for model checking in this protocol as it does for the cache protocols. In fact, getting the details right in this proof was much more difficult subjectively than for FLASH.

This difficulty may be related to the two sorts in the model that are totally ordered: epochs and stakes. For these sorts we use the schemata for totally ordered sets detailed in Section 4. The ordering of these sorts introduces some difficulty in the proof, requiring more detailed invariants. For example, suppose we want to show that the first invariant above holds at the moment when a given process leaves one epoch and enters the next. The votes received at the epoch depend on all the previous epochs. We cannot however, make all of the unboundedly many lesser epochs concrete by adding a finite number of prophecy variables. This means our property must be inductive over epochs, that is, it holds now if it held in the past at the start of some *particular* epoch we can identify (perhaps the previous one). The need to write invariants that are inductive over ordered datatypes may account for the the fact that the VS-Paxos invariant is more complex than that of the more complex FLASH protocol.

Discussion We can make several general observations about these case studies. First, the performance of the finite-state model checker was never problematic.

It always produced results in a reasonable amount of time and was not the bottleneck in constructing any of the proofs. Rather the most time-consuming task was usually analyzing the abstract counterexamples. This task proved tractable in practice, allowing the proof search process to converge.

Second, the invariants used in the model checking approach are generally much smaller than the manual ones because of the model checker's ability to infer state invariants.

This advantage may be somewhat offset by the need to provide prophecy variables to refine the abstraction, especially in the case where there are many unbounded sorts. Moreover, the need to write properties that are inductive over ordered sorts may lessen the advantage of model checking in invariant complexity. This was evident in the case of VS-Paxos and to some extent in Tomasulo as well, because of the implicit induction over the instruction stream. These criteria may be helpful in deciding which approach to take to a given proof problem.

Finally, it is interesting to note that the schemata presented in Section 4 proved adequate in all cases. That is, in no case was it necessary to add a schema to refine the abstraction of the transition relation. This indicates there is no need in practice to restrict to decidable logics or pay the cost of computing best transformers.

7 Conclusion

We have presented a method of abstracting parameterized or infinite-state SMC problems to finite-state problems based on propositional skeletons and eager theory explication. The method is extensible in the sense that users can add abstractions (or refine existing abstractions) by providing axiom schemata. It generalizes the 'datatype reduction' approach of [18] while giving both a simpler theoretical account and allowing a simpler implementation. Compared to predicate abstraction, it has the advantage that it can be applied to undecidable logics and does not require a costly decision procedure in the loop. The approach has been implemented in the IVy tool. Based on some case studies, we found that the approach is practical and requires substantially less complex auxiliary invariants than inductive invariant checking. We identified some conditions under which the approach is likely to be most effective.

Conceivably some of the tasks performed here by a human could be automated. However, the resulting system would be liable to fail unpredictably and opaquely. The present approach is an attempt to create a usable trade-off between human input and reliability.

The next step is to implement liveness. Recent work has constructed liveness proofs in IVy by an infinite-state liveness-to-safety reduction, but the proofs are complex [21]. It would interesting to compare this to an approach that leverages a finite-state model checker's ability to prove liveness.

References

1. Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsch, editors, *Handbook of Satisfiability*, chapter 12, pages 737–797. IOS Press, 2008.
2. Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, July 2011.
3. Ken Birman, Dahlia Malkhi, and Robbert van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, November 2010.
4. Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010.
5. Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan, and Ching-Tsun Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*, pages 81–88. IEEE Computer Society, 2006.
6. Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, volume 3312 of *Lecture Notes in Computer Science*, pages 382–398. Springer, 2004.
7. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
8. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
9. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.
10. Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Better generalization in IC3. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 157–164. IEEE, 2013.
11. Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015.
12. Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2001.

13. Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy. The stanford FLASH multiprocessor. In David A. Patterson, editor, *Proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago, IL, USA, April 1994*, pages 302–313. IEEE Computer Society, 1994.
14. Shuvendu K. Lahiri and Randal E. Bryant. Constructing quantified invariants via predicate abstraction. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 267–281. Springer, 2004.
15. Kenneth L. McMillan. IVy. http://microsoft.github.io/ivy/. Accessed: 2018-01-28.
16. Kenneth L. McMillan. Circular compositional reasoning about liveness. In Pierre and Kropf [24], pages 342–345.
17. Kenneth L. McMillan. Verification of infinite state systems by compositional model checking. In Pierre and Kropf [24], pages 219–234.
18. Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.
19. Kenneth L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In Tiziana Margaria and Thomas F. Melham, editors, *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, September 4-7, 2001, Proceedings*, volume 2144 of *Lecture Notes in Computer Science*, pages 179–195. Springer, 2001.
20. Kenneth L. McMillan, Shaz Qadeer, and James B. Saxe. Induction in compositional model checking. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2000.
21. Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL*, 1(OOPSLA):108:1–108:31, 2017.
22. Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630. ACM, 2016.
23. Seungjoon Park and David L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *SPAA*, pages 288–296, 1996.
24. Laurence Pierre and Thomas Kropf, editors. *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*. Springer, 1999.
25. Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2001.