Applications of Craig Interpolants in Model Checking

K.L. McMillan

Cadence Berkeley Labs

Abstract. A Craig interpolant for a mutually inconsistent pair of formulas (A, B) is a formula that is (1) implied by A, (2) inconsistent with B, and (3) expressed over the common variables of A and B. An interpolant can be efficiently derived from a refutation of $A \wedge B$, for certain theories and proof systems. We will discuss a number of applications of this concept in finite- and infinite-state model checking.

1 Introduction

A Craig interpolant for a mutually inconsistent pair of formulas (A, B) is a formula that is (1) implied by A, (2) inconsistent with B, and (3) expressed over the common variables of A and B. An interpolant can be efficiently derived from a refutation of $A \wedge B$, for certain theories and proof systems. For example, interpolants can be derived from resolution proofs in propositional logic, and for systems of linear inequalities over the reals [8, 14]. These methods have been recently been extended [10] to combine linear inequalities with uninterpreted function symbols, and to deal with integer models. One key aspect of these procedures is that they yield quantifier-free interpolants when the premises A and B are quantifier-free.

This paper will survey some recent applications of Craig interpolants in model checking. We will see that, in various contexts, interpolation can be used as a substitute for image computation, which involves quantifier elimination and is thus computationally expensive. The idea is to replace the image with a weaker approximation that is still strong enough to prove some property.

For example, interpolation can be used as an alternative to image computation in model checking, to construct an inductive invariant. This invariant contains only information actually deduced by a prover in refuting counterexamples to the property of a fixed number of steps. Thus, in a certain sense, this method abstracts the invariant relative to a given property. This avoids the complexity of computing the strongest inductive invariant (i.e., the reachable states) as is typically done in model checking, and works well in the the case where a relatively localized invariant suffices to prove a property of a large system.

This approach gives us a complete procedure for model checking temporal properties of finite-state systems that allows us to exploit recent advances in SAT solvers for the proof generation phase. Experimentally, the method is found to be quite robust for industrial hardware verification problems, relative to other

N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440, pp. 1–12, 2005.
© Springer-Verlag Berlin Heidelberg 2005

model checking approaches. The same approach can be applied to infinite-state systems, such as programs and parameterized protocols (although there is no completeness guarantee in this case). For example, it is possible to verify systems of timed automata in this way, or simple infinite-state protocols, such as the N-process "bakery" mutual exclusion protocol.

In addition, interpolants derived from proofs can be mined to obtain predicates that are useful for predicate abstraction, as is done in the Blast software model checker [7]. This approach has been used to verify properties of C programs with in excess of 100K lines of code. Finally, interpolation can be used to approximate the transition relation of a system relative to a given property. This approach can be applied to finite-state model checking and can also be useful in predicate abstraction, where constructing the exact abstract transition relation can be prohibitively costly.

1.1 Outline of the Paper

The next section of the paper introduces the technique of deriving Craig interpolants from proofs. Section 3 then describes the method of interpolation-based model checking, section 4 covers the extraction of predicates for predicate abstraction from interpolants, and section 5 deals with transition relation abstraction.

2 Interpolants from Proofs

Given a pair of formulas (A, B), such that $A \wedge B$ is inconsistent, an *interpolant* for (A, B) is a formula $\hat{A}$ with the following properties:

- A implies $\hat{A}$,
- $\hat{A} \wedge B$ is unsatisfiable, and
- $\hat{A}$ refers only to the common symbols of A and B.

Here, "symbols" excludes symbols such as $\wedge$ and $=$ that are part of the logic itself. Craig showed that for first-order formulas, an interpolant always exists for inconsistent formulas [5]. Of more practical interest is that, for certain proof systems, an interpolant can be derived from a refutation of $A \wedge B$ in linear time. For example, a purely propositional refutation of $A \wedge B$ using the resolution rule can be translated to an interpolant in the form of a Boolean circuit having the same structure as the proof [8, 14].

In [10] it is shown that linear-size interpolants can be derived from refutations in a first-order theory with with uninterpreted function symbols and linear arithmetic. This translation has the property that whenever A and B are quantifier-free, the derived interpolant $\hat{A}$ is also quantifier-free.[1] This property will be exploited in the applications of Craig interpolation that we describe below.

[1] Note that the Craig theorem does not guarantee the existence of quantifier-free interpolants. In general this depends on the choice of interpreted symbols in the logic.

Heuristically, the chief advantage of interpolants derived from refutations is that they capture the facts that the prover derived about A in showing that A is inconsistent with B. Thus, if the prover tends to ignore irrelevant facts and focus on relevant ones, we can think of interpolation as a way of filtering out irrelevant information from A.

3 Model Checking Based on Craig Interpolation

We now consider an application of Craig interpolation as a replacement for the costly image operator in symbolic model checking. In effect, interpolation allows us to filter information out of the image that is not relevant to proving the desired property.

3.1 Representing Systems Symbolically

In symbolic model checking, we represent the transition relation of a system with a formula. Here, we assume we are given a first-order signature S, consisting of individual variables and uninterpreted n-ary functional and propositional constants. A *state formula* is a first-order formula over S, (which may include various interpreted symbols, such as $=$ and $+$). We can think of a state formula ϕ as representing a set of states, namely, the set of first-order models of ϕ. We will express the proposition that an interpretation σ over S models ϕ by $\phi[\sigma]$. We also assume a first-order signature S', disjoint from S, and containing for every symbol $s \in S$, a unique symbol s' of the same type. For any formula or term ϕ over S, we will use ϕ' to represent the result of replacing every occurrence of a symbol s in ϕ with s'. Similarly, for any interpretation σ over S, we will denote by σ' the interpretation over S' such that $\sigma' s' = \sigma s$. A *transition formula* is a first-order formula over $S \cup S'$. We think of a transition formula T as representing a set of state pairs, namely the set of pairs (σ_1, σ_2), such that $\sigma_1 \cup \sigma_2'$ models T. Will will express the proposition that $\sigma_1 \cup \sigma_2'$ models T by $T[\sigma_1, \sigma_2]$.

Given two state formulas ϕ and ψ, we will say that ψ is *T-reachable* from ϕ (in k steps) when there exists a sequence of states $\sigma_0, \ldots, \sigma_k$, such that $\phi[\sigma_0]$ and for all $0 \leq i < k$, $T[\sigma_i, \sigma_{i+1}]$, and $\psi[\sigma_k]$.

3.2 Bounded Model Checking

The fact that ψ is reachable from ϕ for bounded k can be expressed symbolically. For all integers i, let S_i be a first-order signature (representing the state of the system at time i) such that for every $s \in S$, there is a corresponding symbol s_i in S_i of the same type. If f is a formula, we will write f_i to denote the result of substituting s_i for every occurrence of a symbol s, and s_{i+1} for every occurrence of a symbol s', in f. Thus, assuming T is total, ψ is T-reachable within k steps from ϕ when this formula is consistent:

$$\phi_0 \wedge T_0 \wedge \cdots T_{k-1} \wedge (\psi_0 \vee \cdots \vee \psi_k)$$

We will refer to this as a *bounded model checking* formula [2], since by testing satisfiability of such formulas, we can determine the reachability of one condition from another within a bounded number of steps.

3.3 Symbolic Model Checking

Let us define the *strongest postcondition* of a state formula ϕ with respect to transition formula T, denoted $\mathrm{sp}_T(\phi)$, as the strongest proposition ψ such that $\phi \wedge T$ implies ψ'. We will also refer to this as the *image* of ϕ with respect to T.

A *transition system* is a pair (I, T), where the initial condition I is a state formula and T is a transition formula. We will say that a state formula ψ is *reachable* in (I, T) when it is T-reachable from I, and it is an *invariant* of (I, T) when $\neg\psi$ is not reachable in (I, T). A state formula ϕ is an *inductive invariant* of (I, T) when I implies ϕ and $\mathrm{sp}_T(\phi)$ implies ϕ (note that an inductive invariant is trivially an invariant).

The strongest invariant of (I, T) can be expressed as a fixed point of sp_T, as follows:

$$R(I, T) = \mu Q.\ I \vee \mathrm{sp}_T(Q)$$

We note that the fixed points with respect to Q are exactly the inductive invariants. To prove the existence of the least fixed point, *i.e.*, the strongest inductive invariant, we have only to show that the transformer sp_T is monotonic.

Now, suppose that we have a method of symbolically computing the strongest postcondition. For example, in the case of propositional logic, the strongest postcondition is given by

$$\mathrm{sp}_T(\phi)' = \exists S.(\phi \wedge T)$$

Thus, we can compute it using well-developed methods for Boolean quantifier elimination [3, 11]. This means that we can compute the strongest inductive invariant (also known as the reachable state set) by simply iterating this operator to a semantic fixed point, a procedure known as symbolic reachability analysis.[2] To verify that some formula ψ is unreachable in (I, T), we have only to show that it is inconsistent with the strongest inductive invariant.

3.4 Approximate Image Based on Interpolation

The disadvantage of the above approach is that it can be quite costly to compute the strongest inductive invariant, yet this invariant may be much stronger than what is needed to prove unreachability of ψ. By carefully over-approximating the image (strongest postcondition), we may simplify the problem while still proving ψ unreachable. An over-approximate image operator is an operator $\bar{\mathrm{sp}}$, such that, for all predicates ϕ, $\mathrm{sp}_T(\phi)$ implies $\bar{\mathrm{sp}}_T(\phi)$. Using $\bar{\mathrm{sp}}$, we can compute an over-approximation $R'(I, T)$ of the reachable states. We will say that an over-approximate image operator $\bar{\mathrm{sp}}$ is *adequate* with respect to ψ when, for any ϕ that cannot reach ψ, $\bar{\mathrm{sp}}_T(\phi)$ also cannot reach ψ. In other words, an adequate

[2] Note, convergence of this iteration is guaranteed for finite- but not infinite-state systems.

over-approximation does not add any states to the strongest postcondition that can reach a bad state. If $\bar{\text{sp}}$ is adequate, then ψ is reachable exactly when it is consistent with $R'(I,T)$, the over-approximated reachable states. The question, of course, is how to compute an adequate $\bar{\text{sp}}$. After all, if we knew which states could reach a bad state, we would not require a model checker.

One answer is to bound our notion of adequacy. Let's say that a k-adequate image operator is an $\bar{\text{sp}}$ such that, for any ϕ that cannot reach ψ, $\bar{\text{sp}}_T(\phi)$ cannot reach ψ *within k steps*. We note that if k is greater than the diameter of the state space, then k-adequate is equivalent to adequate, since by definition any state that can be reached can be reached within the diameter.

The advantage of this notion is that we can use bounded model checking and interpolation to compute a k-adequate image operator. We set up a bounded model checking formula to determine whether a given state formula ϕ can reach ψ in from 1 to $k+1$ steps. However, we break this formula into two parts:

$$A \doteq \phi_0 \wedge T_0$$
$$B \doteq T_1 \wedge \cdots \wedge T_k \wedge (\psi_1 \vee \cdots \vee \psi_{k+1})$$

Now suppose $A \wedge B$ is unsatisfiable and let $\hat{A}$ be some interpolant for (A, B) (which we can derive from the refutation of $A \wedge B$). Note that the symbols common to A and B are in S_1 (the symbols representing the state at time 1) thus $\hat{A}$ is over S_1. Dropping the time subscripts in $\hat{A}$, we obtain a state formula, which we will take as the over-approximate image of ϕ. That is, let

$$\bar{\text{sp}}_T(\phi) \doteq \hat{A}\langle S/S_1\rangle$$

The properties of interpolants guarantee that $\bar{\text{sp}}$ defined in this way is a k-adequate image over-approximation. Note that, since $\phi_0 \wedge T_0$ implies $\hat{A}$, it follows that every state in $\bar{\text{sp}}_T(\phi)$ is reachable from ϕ in one step, hence $\bar{\text{sp}}$ is an over-approximation. Further, since $\hat{A}$ is inconsistent with B, it follows that no state in $\bar{\text{sp}}_T(\phi)$ can reach ψ within k steps. Hence $\bar{\text{sp}}$ is k-adequate. [3] One way to think about this is that the interpolant is an abstraction of A containing just the information from A that the prover used to prove that ϕ cannot reach ψ in 1 to $k+1$ steps. Thus, it is in a sense an abstraction of the image relative to a (bounded time) property.

Now suppose we use this k-adequate image operator to compute an over-approximation $R'(I,T)$ of the reachable states. If we find that $R'(I,T) \wedge \psi$ is inconsistent, we know that ψ is unreachable. If not, it may be that we have over-approximated too much. In this case, however, we can simply try again with a larger value of k. Note that if the bounded model checking formula $A \wedge B$ turns out to be satisfiable in the first iteration (when $\phi = I$) then ψ is in fact reachable and we terminate with a counterexample.

It is easy to show that, for finite-state systems, if we keep increasing k, this procedure must terminate with either a proof or a counterexample. That is, if

[3] Note that if $A \wedge B$ is satisfiable, then ϕ can reach ψ, so our image operator can yield any over-approximation, the simplest being the predicate TRUE.

we keep increasing k, either we will obtain a counterexample, or k will become greater than the diameter of the state space. In the latter case, our k-adequate image operator is in fact an adequate image operator, so our reachability answer must be correct. In practice, we find that the k values at which we terminate are generally smaller than the diameter. This diameter-based termination bound contrasts with the termination bound for the k-induction method [16] which is length of the shortest simple path in the state space (also called the recurrence diameter). The shortest simple path can be exponentially longer than the diameter.

3.5 Practical Experience

In the case of hardware verification, a system is made up of Boolean gates, hence we can model it with a transition formula T which is purely propositional. We can therefore use an efficient Boolean satisfiability (SAT) solver [13, 17] to solve the bounded model checking formulas. Modern SAT solvers use heuristics designed to focus the proof on relevant facts, and are quite robust against the addition of irrelevant constraints. The solvers are also easily modified to produce refutations by resolution in the unsatisfiable case [12].

As an example, the performance of the interpolation-based model checking procedure using a SAT solver was tested on a set of benchmark problems [12] derived from the PicoJava II microprocessor from Sun Microsystems. The properties in this benchmark suite are localizable, meaning that only a relatively small subset of the components of a large design are needed to prove the properties. Thus, the ability to filter out irrelevant information is crucial to verifying these properties. In fact, the SMV model checker based on Binary Decision Diagrams is unable to verify any of the properties, since it performs exact reachability analysis.

On the other hand, the interpolation-based method using a SAT solver can verify 19 out of the 20 problems. It is also interesting to compare the method with another abstraction technique that uses refutations from bounded model checking formulas to identify a subset of system components that are relevant to the property, and then uses standard BDD-based methods to verify this subset [12]. This method is called *proof-based abstraction*.

Figure 1 shows a run-time comparison of the interpolation-based method against the proof-based abstraction method for the PicoJava-II benchmark set. In the figure, each point represents one benchmark problem, with the value on the X axis representing the time in seconds required for the earlier proof-based abstraction method, and the time on the Y axis representing the time in seconds taken by the interpolation-based method. A time value of 1000 indicates a timeout after 1000 seconds. Points below the diagonal therefore indicate an advantage for the interpolation method. We observe 16 wins for interpolation and 3 for proof-based abstraction, with one problem solved by neither method. In five or six cases, the interpolation method wins by two orders of magnitude. As it turns out, the performance bottleneck in both methods is bounded model checking. The interpolation method, however, tends to terminate at smaller values of k, and thus runs faster on average. This trend has been verified on a large set (about 1000 problems) of benchmark problems from industrial applications.

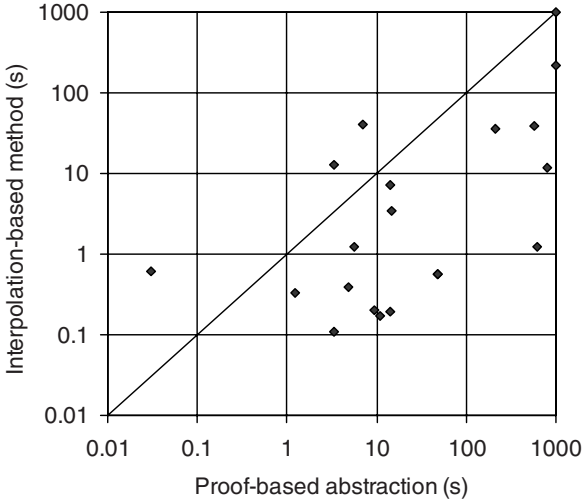

Fig. 1. Run times on PicoJava II benchmarks

3.6 Infinite-State Systems

It is also possible to apply the interpolation method to infinite-state systems whose transition formulas are first-order formulas. In this case, we can use a first-order decision procedure to check satisfiability of the bounded model checking formulas (provided the procedure can produce refutations in a suitable proof system). In this case, the procedure is not guaranteed to terminate.[4] However, using this approach it is possible to verify safety of some simple infinite-state protocols, such as Fischer's timed mutual exclusion protocol, or a simple version of Lamport's "bakery" mutual exclusion algorithm. The method has also been applied to software model checking, though it is not yet clear whether the approach is more efficient than methods based on predicate abstraction [1, 18].

One interesting point to note here is that, using the interpolation procedure of [10], quantifiers occurring in the transition relation T can result in quantifiers in the interpolants (the quantifiers are used to eliminate variables that are introduced into the interpolants by quantifier instantiation). Thus, the method provides a way to synthesize invariants that contain quantifiers.

As an example, suppose we have a simple program whose state consists of an array a, with all elements initialized to 0. At each step, the program inputs a number x and sets $a[x]$ to 1. We would like to prove that, at all times, $a[z] \neq 2$. Thus, our initial condition I is $\forall j.\ a[j] = 0$, our transition condition T is

$$\forall j.\ \text{if } j = x \text{ then } a'[j] = 1 \text{ else } a'[j] = a[j]$$

[4] However, it seems likely that convergence could be guaranteed given a suitably restricted prover, if the system has a quantifier-free inductive invariant that proves the property. Convergence can also be guaranteed if the system has a finite bisimulation quotient, as in timed automata.

and our final condition ψ is $a[z] = 2$. Expanding the split bounded model checking formula for $k = 1$, we have:

$$A \doteq (\forall j.\ a_0[j] = 0) \wedge (\forall j.\ \text{if } j = x_0 \text{ then } a_1[j] = 1 \text{ else } a_1[j] = a_0[j])$$
$$B \doteq (a_1[z_1] = 2)$$

In refuting this, the prover instantiates the universal in A with $j = z_1$, yielding:

$$A \doteq a_0[z_1] = 0 \wedge \text{if } z_1 = x_0 \text{ then } a_1[z_1] = 1 \text{ else } a_1[z_1] = a_0[z_1]$$
$$B \doteq (a_1[z_1] = 2)$$

Notice the introduction of the extraneous variable z_1 into A. After refuting this pair, the interpolant $\hat{A}$ we obtain is $a_1[z_1] = 0 \vee a_1[z_1] = 1$. The extraneous variable z_1 is then eliminated using a universal quantifier. This is sound, since $\hat{A}$ is implied by the original A which does not contain z_1. This yields the quantified interpolant $\forall j.\ a_1[j] = 0 \vee a_1[j] = 1$. Dropping the subscripts, we have $\forall j.\ a[j] = 0 \vee a[j] = 1$, which is in fact an inductive invariant for our program, proving that $a[z] = 2$ is not reachable.

This approach makes it possible to verify some parameterized protocols, such as the "bakery" with an arbitrary number of processes, which requires a quantified invariant. It should be noted, however, that the technique is not well suited to protocol verification, since it is based on bounded model checking. Empirically, bounded model checking of protocols is observed to be fairly inefficient. This can be explained by the fact that protocols tend not to be localizable (*i.e.*, there is little state information that can be thrown away without breaking the protocol) and they tend to have interleaving concurrency, which limits the prover's ability to propagate implications across time frames. For such applications, it may be more effective to combine the approach with predicate abstraction, as described in the next section.

4 Predicates from Interpolants

Predicate abstraction [15] is a technique commonly used in software model checking in which the state of an infinite-state system is represented abstractly by the truth values of a chosen set of predicates. In effect, the method computes the strongest inductive invariant of the program expressible as a Boolean combination of the predicates. Typically, if this invariant is insufficient to prove the property in question, the abstraction is refined by adding predicates. For this purpose, the Blast software model checker uses interpolation in a technique due to Ranjit Jhala [7].

The basic idea of the technique is as follows. A counterexample is a sequence of program locations (a path) that leads from the program entry point to an error location. When the model checker finds a counterexample in the abstraction, it builds a bounded model checking formula that is satisfiable exactly when the path is a counterexample in the concrete model. This formula consists of a set of

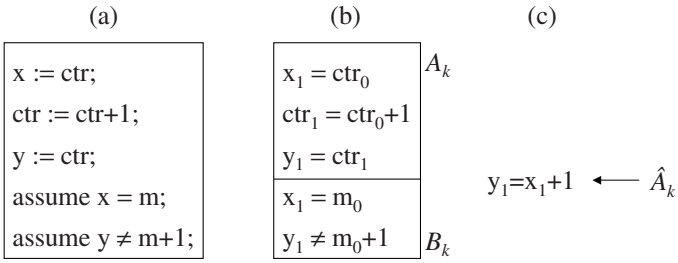

Fig. 2. Predicates from interpolants. Figure shows (a) an infeasible program path, (b) transition constraints, divided into prefix A_k and suffix B_k and (c) an interpolant $\hat{A}_k$ for (A_k, B_k)

constraints: equations that define the values of program variables in each location in the path, and predicates that must be true for execution to continue along the path from each location (these correspond to program branch conditions). As an example, Figure 2 shows a program path, and the corresponding transition constraints.

Now let us divide the path into two parts, at location k. Let A_k be the set of constraints on transitions preceding location k and let B_k be the set of constraints on transitions subsequent to location k. Note that the common variables of A and B represent the values of the program variables at location k. An interpolant for (A_k, B_k) is a fact about location k that must hold if we take the given path to location k, but is inconsistent with the remainder of the path. An example of such a division, and the resulting interpolant, is also shown in 2.

If we derive such interpolants for every location of the path from the same refutation of the constraint set, we can show that the interpolant for location k is sufficient to prove the interpolant for location $k+1$. As a result, if we add the atomic predicates occurring in the interpolants to the set of predicates defining the abstraction, we are guaranteed to rule out the given path as a counterexample in the abstract model. Note that it is important here that interpolants be quantifier-free, since the predicate abstraction method can synthesize any Boolean combination of atomic predicates, but cannot synthesize quantifiers. We can guarantee that the interpolants are quantifier-free if the transition constraints are quantifier-free.

This interpolation approach to predicate selection has the advantage that it tells us which predicates are relevant to each program location in the path. By using at each program location only predicates that are relevant to that location, a substantial reduction in the number of abstract states can be achieved, resulting in greatly increased performance of the model checker [7]. The fact that the interpolation method can handle both linear inequalities and uninterpreted functions is useful, since linear arithmetic can represent operations on index variables, while uninterpreted functions can be used to represent array lookups or pointer dereferences, or to abstract unsupported operations (such as multiplication).

Notice, finally, that the predicate abstraction requires us to solve bounded model checking instances only for particular program paths, rather than for all possible paths of a given length. Such problems are much easier for the decision procedure to solve. Thus, the predicate abstraction approach might be feasible in cases such as protocols where full bounded model checking tends not to be practical.

5 Transition Relation Abstraction Using Interpolants

Because of the expense of image computation in symbolic model checking, it is often beneficial to abstract the transition relation before model checking, removing information that is not relevant to the property to be proved. Some examples of techniques for this purpose are [4, 12].

Here, we will consider a method of abstracting the transition relation using bounded model checking and interpolation. The technique is based on the notion of a *symmetric interpolant*. That is, given an inconsistent set of formulas $A = \{a_1, \ldots, a_n\}$ a symmetric interpolant for A is a set of formulas $\hat{A} = \{\hat{a}_1, \ldots, \hat{a}_n\}$ such that each a_i implies $\hat{a}_i$, and $\hat{A}$ is inconsistent, and each $\hat{a}_i$ is over the symbols common to a_i and $A \setminus a_i$. We can construct a symmetric interpolant for A from a refutation of $\bigwedge A$ by simply letting $\hat{a}_i$ be the interpolant derived from the given refutation for the pair $(a_i, \bigwedge A \setminus a_i)$. As long as all the individual interpolants are derived from the same proof, we are guaranteed that their conjunction is inconsistent.

Now, given a transition system (I, T), and a formula ψ, let us consider the set of formulas:

$$A = \{I_0, T_0, \ldots, T_{k-1}, (\psi_0 \vee \cdots \psi_k)\}$$

Note that $\bigwedge A$ is exactly the bounded model checking formula for k steps. Suppose we refute this formula, and from the refutation, construct a symmetric interpolant $\hat{A}$. Notice that each $\hat{T}_i$ is a formula implied by the transition relation at time i. If we take the conjunction of these formulas, we have a transition formula that admits no path up to k steps from I to ψ. That is, let the abstract transition relation be

$$\hat{T} = \bigwedge \hat{T}_i \langle S/S_i \rangle \langle S'/S_{i+1} \rangle$$

If we model check unreachability of ψ in the abstract transition system $(I, \hat{T})$, we are guaranteed that there is no counterexample of up to k steps. If ϕ is in fact unreachable in $(I, \hat{T})$, we know it is unreachable in the stronger (I, T). Otherwise, we can refine $\hat{T}$ using a larger value of k. In the finite-state case, this method is guaranteed to converge, since we cannot refine $\hat{T}$ infinitely.

The advantages of $\hat{T}$ as a transition relation are that (1) it contains only facts about the transition relation used in resolving the bounded model checking problem, and (2) it contains only state-holding symbols (those that occur in I or occur primed in T). Thus, for example, free variables introduced to represent inputs of the system are eliminated. This can substantially simplify the image computation.

One potential application of this idea is in predicate abstraction. Since the image computation in predicate abstraction requires in the worst case an exponential number of calls to a decision procedure, software model checkers tend to avoid an exact computation by using approximate methods that lose correlations between predicates [1]. This approximation can lead to false counterexamples. On the other hand, if we derive the transition relation approximation from symmetric interpolants (another idea due to Ranjit Jhala) we can guarantee convergence without using an exact image computation, and at the same time focus the transition relation approximation on relevant facts. We can improve the performance by considering only bad program paths found by the model checker, as opposed to all possible paths of length k. Preliminary experiments show that this approach converges more rapidly than the approach of [6], which uses analysis of the predicate state transitions in the abstract counterexamples to refine the transition relation.

6 Conclusion

We have seen that Craig interpolants derived from proofs have a variety of applications in model checking, primarily in replacing exact image computations with approximate ones. Interpolation allows us to exploit the ability of modern SAT solvers, and decision procedures based on them, to narrow down a proof to relevant facts. We can extract as an interpolant just the information about an image or a transition relation that was actually used by the prover to refute a bounded model checking instance. This allows us in turn to weaken our computation of the strongest invariant, while still proving a given property, or to extract the building blocks from which a suitable invariant might be constructed.

A number of the potential applications of interpolation have yet to be explored. For example, interpolation-based model checking for software seems a promising approach, as does interpolation-based transition relation abstraction for hardware verification. Recently, predicate abstraction methods have been extended to the synthesis of quantified invariants in a method called *indexed predicate abstraction* [9]. It seems plausible that quantified interpolants could be used in the selection of indexed predicates in this method. It also seems plausible that interpolation could be used to good effect for transition relation abstraction in parallel program verification.

Finally, it would be useful to extend the extraction of interpolants from proofs to other theories, for example, first-order arrays and bit vectors. This would extend the utility of interpolant extraction as a tool in the verifier's toolkit.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *STTT*, 5(1):49–58, 2003.

2. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *LNCS*, pages 193–207, 1999.
3. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *VLSI '91*, Edinburgh, Scotland, 1991.
4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
5. W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symbolic Logic*, 22(3):269–285, 1957.
6. Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *LICS 2001*, pages 51–60, 2001.
7. T. A. Henzinger, R. Jhala, Rupak Majumdar, and K. L. McMillan. Abstractions from proofs. In *Principles of Prog. Lang. (POPL 2004)*, pages 232–244, 2004.
8. J. Krajíček. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symbolic Logic*, 62(2):457–486, June 1997.
9. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *LNCS*, pages 267–281. Springer, 2004.
10. K. L. McMillan. An interpolating prover. *Theoretical Computer Science*. To appear.
11. K. L. McMillan. Applying sat methods in unbounded symbolic model checking. In *Computer-Aided Verification (CAV 2002)*, pages 250–264, 2002.
12. K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, pages 2–17, 2003.
13. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.
14. P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 62(2):981–998, June 1997.
15. Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254, pages 72–83, Haifa, Israel, 1997. Springer-Verlag.
16. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer Aided Design*, 2000.
17. J. P. M. Silva and K. A. Sakallah. GRASP–a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design, November 1996*, 1996.
18. R. Majumdar T. A. Henzinger, R. Jhala and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL 2002)*, 2002.