Using Dynamic Symbolic Execution to Improve Deductive Verification

Dries Vanoverberghe, Nikolaj Bjørner, Jonathan de Halleux, Wolfram Schulte, and Nikolai Tillmann

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
{t-drivan*,nbjorner,jhalleux,schulte,nikolait}@microsoft.com

Abstract. One of the most challenging problems in deductive program verification is to find inductive program invariants typically expressed using quantifiers. With strong-enough invariants, existing provers can often prove that a program satisfies its specification. However, provers by themselves do not find such invariants. We propose to automatically generate executable test cases from failed proof attempts using dynamic symbolic execution by exploring program code as well as contracts with quantifiers. A developer can analyze the test cases with a traditional debugger to determine the cause of the error; the developer may then correct the program or the contracts and repeat the process.

1 Introduction

Many modern specification and verification systems such as Spec# [3], JML [25] and so forth, use a design-by-contract approach [27], where the specification language is typically an extension of the underlying programming language. The verification of these contracts often uses verification condition generation (VCG). The verification conditions are first-order logical formulas whose validity implies the correctness of the program. The formulas are then fed to interactive or automatic theorem provers.

In practice, there are two limitations of this VCG and proving approach. First, most program verification tools do not by themselves produce sufficiently strong contracts. Such contracts need to be crafted or synthesized independently, usually by a human being. Second, if the program verification tool fails to prove the desired properties, then discovering the mismatch between the contracts and the code, or why the contracts were not strong enough, remains a difficult task. In practice, the program verification tools offer only little help.

Most of the above mentioned program verification tools employ an automated solver to prove program properties. These solvers must typically be able to handle a combination of domains, such as integers, bit-vectors, arrays, heaps, and data-types, which are often found in programming languages. In addition most interesting contracts involving functional correctness and the heap involve quantifiers, which solvers must reason about, as well. Solvers, that combine various theories are called SMT (Satisfiability Modulo Theories). Such solvers have recently gained a lot of attention, see for instance

* Permanent email address: Dries.Vanoverberghe@cs.kuleuven.be

K. Havelund, R. Majumdar, and J. Palsberg (Eds.): SPIN 2008, LNCS 5156, pp. 9–25, 2008.

Simplify [16], CVC3 [4], Fx7 [28], Verifun [19], Yices [18], and Z3 [15]. To support quantifiers, a commonly used technique for SMT solvers is to use pattern matching to determine relevant quantifier instantiations. Pattern matching happens inside of the solver on top of the generated verification condition. When a proof attempt fails, pinpointing the insufficient annotation within the context of the SMT solver is obscured by the indirection from the program itself. To give good feedback to the developer in such a case, the SMT solver should provide a human-readable counter-example, i.e. a model of the failed proof attempt. However, producing (informative) models from quantified formulas remains an open research challenge. Producing models for quantifier-free formulas, is on the other hand easy and relatively well understood.

Symbolic execution [24] is a well-known technique to generate test cases. In particular, test cases that expose errors help a developer in debugging problems. Symbolic execution analyzes individual execution traces, where each trace is characterized by a path condition, which describes an equivalence class of test inputs. A constraint solver is used to decide the feasibility of path conditions, and to obtain concrete test inputs as representatives of individual execution paths. Note that constraints can also be solved by SMT solvers with model generation capabilities. Recently symbolic execution has been extended to deal with contracts, even contracts involving quantifiers over (sufficiently small) finite domains [30].

In this paper, we propose an extension of symbolic execution for programs involving contracts with quantifiers over very large, and potentially unbounded domains. This is of benefit for debugging failed proof attempts. If a deductive proof fails due to insufficient quantified assertions, we use symbolic execution to generate concrete test cases that exhibit mismatches between the program and its contracts with quantifiers. Quantifiers are furthermore instantiated using symbolic values encountered during a set of exhibited runs. In this setting, quantifier instantiation is limited to values supplied to or produced by a symbolic execution. With a sufficient set of instances, we can derive test cases that directly witness limitations of the auxiliary assertions. The SMT solver no longer needs to handle these quantifiers.

In particular, we handle branch conditions with quantifiers as follows: When a branch condition of the program involves an unbounded universal quantifier, we first recover the quantified formula $\phi(x)$ (which must be embedded by the compiler into the code), we introduce a Boolean variable t that represents whether the quantifier holds, and we introduce a branch in the program over t. Conceptually, program execution forks at this branch. If $t = \texttt{false}$, we introduce another additional test input c that represents the bound variable, and explore the quantified formula until we find a c such that $\neg\phi(c)$, If $t = \texttt{true}$, then we proceed with the program. Quantifier instantiations are identified lazily using pattern matching over the symbolic trace.

An extended form of pattern matching, called E-matching [12], is used for instantiating quantifiers in SMT solvers. E-matching admits matching modulo a set of equalities. We lift E-matching to combine run-time information with symbolic execution. To this end, we use *dynamic* symbolic execution, which executes the program for particular test inputs in order to obtain derived run-time values. Run-time values are used to determine which symbolic terms *may* be equal. Thus, pattern matching is performed on the

Algorithm 2.1. Dynamic symbolic execution

Set $J := \mathit{false}$	*intuitively, J is the set of already...*
loop	*...analyzed program inputs*
Choose program input i such that $\neg J(i)$	*stop if no such i can be found*
Output i	
Execute $P(i)$; record path condition C	*in particular, $C(i)$ holds*
Set $J := J \vee C$	
end loop	

symbolic trace, where concrete run-time values are used to supply a set of alternative views of values appearing in a trace.

We implemented a prototype of our approach as an extension to the dynamic symbolic execution platform Pex [34,32] for .NET, which we develop at Microsoft Research. Pex contains a complete symbolic interpreter for safe programs that run in the .NET virtual machine. Pex uses Z3 [15,14] as a constraint solver, using Z3's ability to compute models for satisfiable constraint systems. Pex has been used within Microsoft to test core .NET components developed at Microsoft. Pex is integrated with Microsoft Visual Studio.

The rest of the paper is structured as follows: Section 2 gives an overview of dynamic symbolic execution. Section 3 walks through a simple example of how our approach generates test cases for a program with contracts involving quantifiers. Section 4 describes in detail how we extend dynamic symbolic execution to handle quantifiers. Section 5 discusses related work. Section 6 concludes.

2 Dynamic Symbolic Execution

2.1 Introduction

Dynamic symbolic execution [22,5] is a variation of conventional static symbolic execution [24]. Dynamic symbolic execution consists in executing the program, starting with arbitrary inputs, while performing a symbolic execution in parallel to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution. Then a constraint solver is used to compute variations of the previous inputs in order to steer future program executions along different execution paths. In this way, all execution paths will be exercised eventually.

Algorithm 2.1 shows the general dynamic symbolic execution algorithm.

The advantage of dynamic symbolic execution over static symbolic execution is that the abstraction of execution paths can leverage observations from concrete executions, and not all operations must be expressed and reasoned about symbolically. Using concrete observations for some values instead of fully symbolic representations leads to an under-approximation of the set of feasible execution paths, which is appropriate for testing. Such cases can be detected, e.g. when a function is called that is defined outside of the scope of the analysis. Our tool reports them to the user.

2.2 Symbolic State Representation

In concrete execution, the program's state is given by a mapping from program variables to concrete values, such as 32 or 64-bit integers and heap-allocated (object) pointers. In symbolic execution, the state is a mapping from program variables to terms built over symbolic input values, together with a predicate over symbolic input values, the so-called path condition.

The terms represent symbolically which computations were performed over the symbolic inputs. For example, if x, y, z are the symbolic inputs for the program variables `x`, `y`, `z`, then the statement

```
u = x * (y + z);
```

causes the program variable `u` to be mapped to the term $x * (y + z)$. If the concrete inputs for `x`, `y`, `z` are 2, 3, and 4, respectively, then the concrete value of `u` will be 14.

The path condition is the conjunction of all the guards of all conditional branches that were performed to reach a particular point in an execution trace. For example, when the function

```
void foo(int x) {
    if (x>0) {
        int y = x*x;
        if (y==0) {
            // target
        }
    }
}
```

reaches the *target*, then the path condition consists of two conjuncts, $(x > 0)$ and $(x * x == 0)$. Note that the symbolic state is always expressed in terms of the symbolic input values, that is why the value of the local variable `y` was expressed with the symbolic input x.

2.3 Test Inputs and Non-deterministic Programs

Dynamic symbolic execution determines a set of concrete test inputs for a program. In practice, this means that we determine parameter values for a designed top-level function. In addition to the immediate parameter values, our dynamic symbolic execution platform Pex [34] allows the code-under-test to call the generic function `Choose` in order to obtain an additional test input. In C# syntax, the function has the following signature:

```
T Choose<T>();
```

Each invocation of this function along an execution trace provides the program with a distinct additional symbolic test input. In the following, we will also refer to the functions `ChooseTruth` and `ChooseBoundVariable`. They work just as `Choose`, and their main purpose is to easily distinguish between different choices.

2.4 Making Basic Contracts Executable

Most design-by-contract languages support function pre-conditions and post-conditions, class invariants and loop invariants. In the following, we describe how most contracts

Program 2.1. Implementation of Assume and Assert in C# syntax

```
void Assume(bool b) {
    if (!b) throw new AssumptionException();
}
void Assert(bool b) {
    if (!b) throw new AssertionException();
}
```

can be turned into executable code using `Assert` and `Assume`. This code can then be explored by symbolic execution.

As shown in Program 2.1, the `Assume` and `Assert` functions contain a conditional branch over their Boolean parameter, and they throw an exception when the argument is `false`. An `AssumptionException` is treated by the symbolic execution engine as a filter: test inputs that cause this exception to be thrown are not shown to the user. An `AssertException` indicates an error, since a mismatch between the program under test and its contracts has been found. Test inputs that cause this exception are shown to the user.

We reduce a class invariant to both a pre- and post-condition of each affected function (see e.g. [27]), and a loop invariant to a call to an `Assert` function with the positive condition at the loop entry and with the negative condition at the loop exit. Given a designated top-level function to explore, we turn its pre-conditions into calls to the `Assume` function, and all pre-conditions of called functions into calls of the `Assert` function which are placed at the beginning of the functions. We turn all post-conditions into calls to the `Assert` function which are placed at the end of the function. We discuss the treatment of universal quantifiers in depth in Section 4.

3 Example

Program 3.1 shows an implementation of a swap function. In this example we use C# syntax extended with pre-conditions and post-conditions (`requires` and `ensures`), and `old` expressions.[1] It has two pre-conditions: the array `a` must not be `null`, and the indices must be within the bounds of the array. The three post-conditions express that the elements at index `lo` and `hi` in the array `a` are swapped, and that all remaining elements of the array are identical to the old elements in the array. (Note that the last `if` statement introduces an error into program.)

Program 3.2 shows the translation of the `swap` function, including its pre and post-condition as described in 2.4. The `old` expression is realized by creating a copy of the referenced values in the initial state. The `Forall<int>(i => p(i))` expression refers to a generic function `Forall` that takes a predicate expression `p(i)` as an argument. Intuitively, it represents $\forall i.p(i)$. shows the translated program.

[1] In fact, the syntax we use is Spec# [3], except for our `Forall` function that does not involve bounds. In contrast, the universal quantifier in Spec# must state a finite enumeration of possible values for the bound variable.

Program 3.1. Swap Example

```
public void Swap(byte[] a, int lo, int hi)
    requires a != null;
    requires 0 <= lo && lo <= hi && hi < a.Length;

    ensures a[hi] == old(a[lo]);
    ensures a[lo] == old(a[hi]);
    ensures Forall<int>(i =>
                !(i >= 0 && i < a.Length && i !=lo && i !=hi)
                || a[i] == old(a[i]));
{
    byte tmp = a[hi];
    a[hi] = a[lo];
    a[lo] = tmp;
    if (lo != 0 && hi != 0)
        a[0] = 42;
}
```

When symbolic execution of this program reaches the `Assert(Forall(...))` statement, our treatment of the quantifier (that we describe in detail in Section 4) will consider the case in which the quantifier does not hold. To this end, our technique explores the body of the quantifier using symbolic execution with the intention to find test inputs that make the asserted quantifier true and false.

Case 1: Let's assume the quantifier does not hold. The following code snippet represents a test case that was generated during the search.

```
IPexOracleRecorder oracle = PexOracle.NewTest();
oracle.OnComprehension(0)
    .ChooseTruth(false)
    .ChooseBoundVariable(1610612732);
byte[] bs0 = new byte[2];
TestSwap2(bs0, 1, 1);
```

In this code, `oracle` is initialized by a call to `OnComprehension`, which indicates which quantifier is about to be initialized (here, `0` indicates that it is the first quantifier in the execution trace). Calls to `ChooseTruth` and `ChooseBoundVariable`, set the truth value of the quantifier and the value of the bound variable (in this case, the truth value `false` indicates that the quantifier should not hold, and the bound variable `i` gets the value `1610612732`). The argument for the parameter `a` is an array of size 2, so the index `i`, chosen earlier, is outside of the range of the array. With these assignments the body of the quantifier evaluates to `true`. Since we are looking for a case where the quantifier does not hold, these test inputs get pruned and the search for a counterexample continues.

When the exploration finds the case in which `i` is zero, it discovers that the body of the quantifier evaluates to `false`. In this case, the `Assert` statement fails and a failing test case has been found:

Program 3.2. Translated Swap Example

```
public void Swap(byte[] a, int lo, int hi) {
    Assume(a != null);
    Assume(0 <= lo && lo <= hi && hi < a.Length);
    byte[] old_a = a.Clone();

    byte tmp = a[hi];
    a[hi] = a[lo];
    a[lo] = tmp;
    if (lo != 0 && hi != 0)
        a[0] = 42;

    Assert(a[hi] == old_a[lo]);
    Assert(a[lo] == old_a[hi]);
    Assert(Forall<int>(i =>
                  !(i >= 0 && i < a.Length && i != lo && i !=hi)
                  || a[i] == old_a[i]));
}
```

```
    IPexOracleRecorder oracle = PexOracle.NewTest();
    oracle.OnComprehension(0)
        .ChooseTruth(false)
        .ChooseBoundVariable(0);
    byte[] bs0 = new byte[3];
    TestSwap2(bs0, 1, 2);
```

Case 2: Let's assume the quantifier holds. In this case, the quantifier is instantiated lazily using pattern matching when relevant constraints become available. The execution continues normally at first. Whenever the program tries to observe a value that should not exist according to the asserted quantifier, the path will be pruned. For example, suppose that we add the code in Program 3.3 at the end of Program 3.2.

In this case, the code checks if the value of `a` at an arbitrary index `j` satisfying `j>=0 && j < a.Length && lo != j && hi != j` is still equal to its old value.

At this point our pattern matching engine will detect a match with the quantifier, and thus the engine instantiates the quantifier. However, the instantiated quantifier together with the current path condition is unsatisfiable. We detect that the path is infeasible, stop its execution, and prune the path. The `Assert(false)` statement will never be executed.

The following code snippet shows a test case for which the quantifier holds:

```
    IPexOracleRecorder oracle = PexOracle.NewTest();
    oracle.OnComprehension(0)
        .ChooseTruth(true);
    byte[] bs0 = new byte[3];
    TestSwap2(bs0, 1, 2);
```

Program 3.3. Extra code

```
int j = Choose<int>();
if (j>=0 && j < a.Length &&
    lo != j && hi != j && a[j] != old(a[j]))
    Assert(false);
```

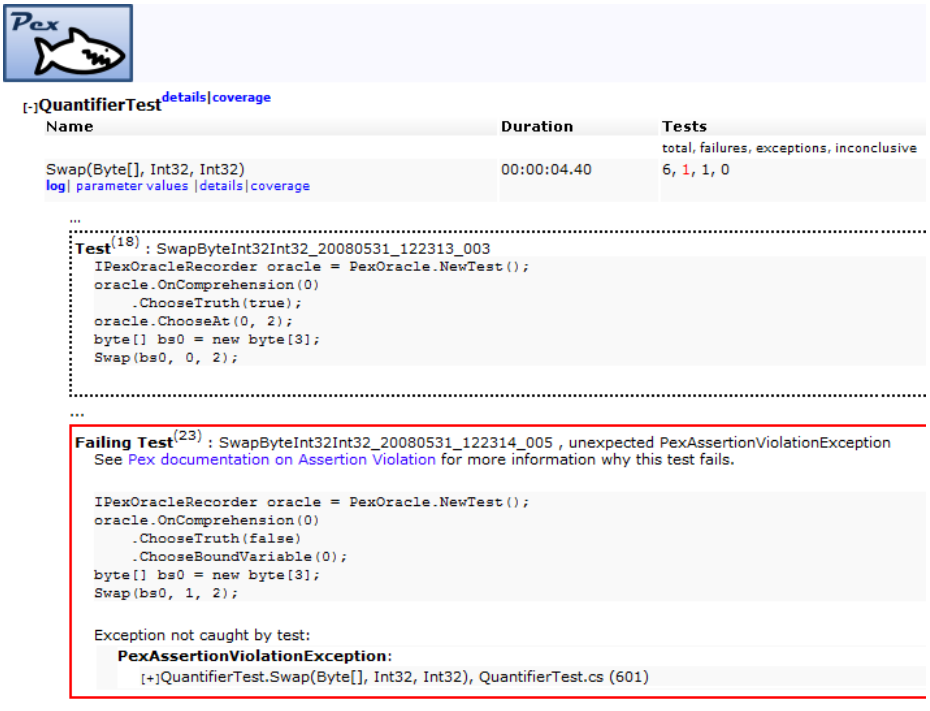

Fig. 1. Example Report of Pex

When exploring the `if` statement of Program 3.3, the exploration observes the term `a[j]`. Since this term matches with a subterm of the quantifier body, the body of the quantifier gets instantiated and added as an extra constraint (`!(j >= 0 && j < a.Length && j != lo && j != hi) || a[j] == old(a[j])`). After entering the body of the `if` test, we know that the left disjunct of this constraint is `false`, the right disjunct is `true`. This contradicts the `if` test that `a[j] == old(a[j])` and therefore this path is infeasible.

Figure 1 shows a partial report that is the result of running Pex on the example (including the extra code). The failing test is the same as the one we discussed earlier.

4 Quantifiers

4.1 Introduction

Universal and existential quantifiers are a noteworthy example of an extension of a programming language to support design-by-contract specifications. To make quantifiers executable, existing approaches typically require the bound variables to be in a range or from a set. A significant drawback is that executing these quantifiers for all elements within a range is often impractical. For instance, a contract that involves bound variables that ranges over all records in a database or over all 64-bit integers will require significant resources and be impractical to check repeatedly at run-time. Our approach does not require such bounds.

In the following, we assume that the quantifier body is a pure expression, i.e. that it does not have side-effects. Also, we do not consider nested quantifiers, and leave it for future work to explore several alternative ways of handling nested quantifiers (such as, relying on the SMT solver for these, using prenex forms, or executing the body in a separate run so that we can apply the techniques recursively).

4.2 Compiling a Quantifier to a Non-deterministic Program

We describe how quantifiers within contracts can be transformed into executable, non-deterministic code. We focus on the treatment of universal quantifiers. Since $\exists x.\varphi(x)$ is equivalent to $\neg\forall\, x.\neg\varphi(x)$, we can use the same approach for existential quantifiers.

Program 4.1 shows the executable version of a universal quantifier. It is implemented as a library function `Forall<T>` where `T` is the type of the bound variable. It takes the body of the quantifiers as input as a predicate `p` (`Predicate<T>` is the type of a function which takes an input of type `T` and returns a Boolean value).

Program 4.1. Executable Version of Universal Quantifier

```
bool Forall<T>(Predicate<T> p) {
    bool q_holds = ChooseTruth<bool>();
    if (q_holds) {              // Quantifier holds
        AssumeForall<T>(p);
    } else {                    // Quantifier does not hold.
        T val = ChooseBoundVariable<T>();
        Assume(!p(val));
    }
    return q_holds;
}
```

The implementation first obtains a value by calling `ChooseTruth` and stores it in the variable `q_holds`. The value represents whether the quantifier holds. The implementation branches over this value. This is a non-deterministic choice.

Case 1: When `q_holds` is `true`. In this case, the rest of the computation should assume that the quantifier indeed is `true` for all values. This is represented by a call to the function `AssumeForall<T>(p)`, which we will explain in detail in Section 4.3. In a nutshell, the quantifier is added to a list of active quantifiers, and the symbolic execution engine will look out for *relevant values* that appear in the program.

For all such relevant values $v_1, \ldots, v_n$, the instantiated quantifier $\texttt{p}(v_i)$ represents a condition that the test inputs must fulfill. All execution paths in which the quantifier does not hold for these values will be cut off.

In other words, the statement

```
AssumeForall<T>(p);
```

conceptually just represents a set of statements

```
Assume(p(v1));
Assume(p(v2));
...
Assume(p(vn));
```

for all relevant values $v_1, \ldots, v_n$.

However, since the relevant values might only be discovered as the program execution continues, these additional `Assume(p(...))` clauses are realized by our symbolic execution engine.

Case 2: When `q_holds` is `false`. In this case, the implementation will attempt to obtain a value `val` for which the predicate `p` is `false`. This is realized by performing another non-deterministic choice to obtain a value `val`, checking whether the predicate holds on that value, and pruning all execution paths where `p(val)` did not hold by calling `Assume(!p(val))`. The underlying dynamic symbolic execution engine will attempt to obtain values for `val` such that the execution proceeds beyond the call to `Assume`. When no such value is found, all execution paths starting from the assumption that `q_holds` is `false` will be effectively cut off.

Illustration. Figure 2 shows an execution tree for evaluating a quantifier whose branches represent the choices introduced by the `ChooseTruth` and `ChooseBoundVariable` functions. The first node with the outgoing branches `false` and `true` represents the program branch over `q_holds`. If `q_holds` is `true`, the predicate $p(i)$ is added to the list of quantifiers; If `q_holds` is `false`, the body of the quantifier `p` is explored in order to find a value for which the predicate does not hold.

4.3 Pattern Based Quantifier Instantiation

SMT solvers based on integrations of modern sat-solving techniques [20], theory lemmas and theory combination [13] have proven highly scalable, efficient and suitable for integrating theory reasoning. However, numerous applications from program analysis and verification require furthermore to handle proof-obligations involving quantifiers. As we notice here, quantifiers are often used for capturing frame conditions over loops, summarizing auxiliary invariants over heaps. Quantifiers can also be used

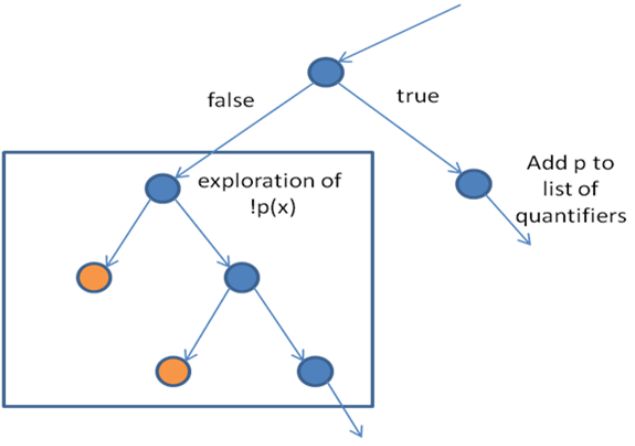

Fig. 2. Example Exploration Tree

for supplying axioms of theories that are not already equipped with solvers. A well known approach for incorporating quantifier reasoning with ground decision procedures uses an E-matching algorithm that works with the set of ground equalities asserted during search to instantiate bound variables. The ground equalities are stored in a data-structure called an *E-graph*. E-matching is used in several theorem provers: Simplify [16], CVC3 [4], Fx7 [28], Verifun [19], Yices [18], Zap [2], and Z3 [12].

We will now describe the quantifier instantiation process and E-matching problem in more detail.

Quantifiers and SAT-Solvers. Suppose φ is a quantifier free formula we wish to show unsatisfiable (conversely $\neg\varphi$ is valid), then φ can be converted into an equi-satisfiable set of clauses [35] of the form $(\ell_1 \vee \ell_2 \vee \ell_2) \wedge (\ell_4 \vee \ldots) \wedge \ldots$, where each literal ℓ_i is an atom or a negation of an atom, each atom is either an equality $t \simeq s$, a predicate symbol P, or some other relation applied to ground arguments. Then SAT solving techniques are used for searching through truth assignments to the atoms [29]. An equality $t \simeq s$ assigned to *true* cause as a side-effect a partition of ground terms in the E-graph for φ to be collapsed. When an equality $t \simeq s$ is assigned to false, the theory solvers check that s and t do not appear in the same partition; otherwise, the assignment is contradictory. If φ contains quantifiers, then quantified sub-formulas are treated as atomic predicates. Thus, if φ contains a sub-formula of the form $\forall x.\psi$, then this sub-formula is first replaced by a predicate $p_{\forall x.\psi}$. The SAT solver core, and the E-graph structure can then work in tandem to find a satisfying assignment to $\varphi\,[\forall x.\psi \leftarrow p_{\forall x.\psi}]$. Suppose first the SAT solver core chooses to set $p_{\forall x.\psi}$ to *false*; it means that under the current assignment of truth values to sub-formulas of φ, it must be the case that $\neg\forall x.\psi$, or in other words, there is some (Skolem) constant sk, such that $\neg\psi[x \leftarrow sk]$. Thus we may add the additional fact

$$\neg p_{\forall x.\psi} \rightarrow \neg\psi[x \leftarrow sk]$$

to φ, propagate the truth assignment for $p_{\forall x.\phi}$, and have the resulting formula $\neg\psi[x \leftarrow sk]$ participate in subsequent search. Conversely, if the SAT solver core chooses to set $p_{\forall x.\psi}$ to *true*, then for the assignment to be consistent with φ, it must be the case that $\forall x.\psi$. In this case, ϕ holds for every instantiation of x. We will later describe how suitable instantiations for x are determined, but suppose for a moment that an instantiation t_1 is identified. We can then add the following fact to φ

$$p_{\forall x.\psi} \rightarrow \psi[x \leftarrow t_1]$$

while maintaining satisfiability, and use the current truth-assignment to propagate the instantiation.

The E-matching Problem. As mentioned above, a widely used algorithmic component for finding suitable instantiations consists of an E-matching algorithm. The E-matching problem is more precisely defined as: *Given* a set of ground equations E, where E is a set of the form $\{t_1 \simeq s_1, t_2 \simeq s_2, \ldots\}$, a ground term t and a term p possibly containing variables. *Provide* the set of substitutions θ, modulo E, over the variables in p, such that $E \models t \simeq \theta(p)$. Two substitutions are equivalent if their right hand sides are pairwise congruent modulo E.

When solving the E-matching problem, it is common to build a *congruence closure* based on the equalities in E. The congruence closure partitions terms in E and ground sub-terms from p and t; it is the least partition ($\equiv_C$) closed under equivalence (reflexivity, symmetry, and transitivity), and functionality: if $t_1 \equiv_C s_1, t_2 \equiv_C s_2$, and $f(t_1, t_2)$, and $f(s_1, s_2)$ are sub-terms, then $f(t_1, t_2) \equiv_C f(s_1, s_2)$. Efficient implementations of congruence closure typically use union-find data-structures and use-lists [17]. A congruence closure is the finest partition induced by the equalities E.

The E-matching problem is NP-complete, but in the context of SMT problems the harder practical problem is to handle a massive number of patterns and a dynamically changing set of patterns and equalities E. Efficient data-structures and algorithms for these situations are described in [12].

Patterns. So what do we E-match against? For most practical purposes, the answer is a set of sub-terms in the quantified formula (from the above example, ψ) that contain the bound variables (from the above example, the variable x).

Example 1. Consider one of the axioms used for characterizing arrays [26]:

$$\forall a, i, j, v \;.\; i \not\simeq j \rightarrow read(write(a, i, v), j) \simeq read(a, j) \;\;.$$

When should it be instantiated? One clear case is when the sub-term $read(write(a, i, v), j)$ matches a term in set of current ground terms. A less obvious case is when both $write(a, i, v)$ and $read(a, j)$ occur as terms. These terms combined contain all bound variables, so they can be used for instantiating the quantifier. The latter condition refers to two occurrences of a. These two occurrences can match any pair of terms in the current context as long as they belong to the same equality partition.

In Simplify [16], such patterns are annotated together with the quantifier.

$$\forall a, i, j, v \;.\; (\texttt{PATS}\ read(write(a, i, v), j)\ (\texttt{MPAT}\ write(a, i, v)\ read(a, j))) :$$
$$i \not\simeq j \rightarrow read(write(a, i, v), j) \simeq read(a, j) \;\;.$$

We here perform a partial lifting of the concept of patterns to the context of C# programs.

Example 2. A contract that assumes the array `a` to be `0` on every index `i` can be formulated as:

```
Assume(Forall<int>(i =>
                  !(i >= 0 && i < a.Length)
                  || Pattern<int>(a[i]) == 0));
```

In this contract we have used the generic function:

```
T Pattern<T>(T value);
```

The pattern specifies that the quantifier on `i` be instantiated whenever a sub-term of the form `a[i]` is created during search. Multiple occurrences of `Pattern` are treated as alternatives. Our pattern extension in Pex does not currently provide a counter-part to multi-patterns (conceptually it is a relatively easy extension, that has yet to be exercised: add a numeric argument to `Pattern`, all occurrences using the same numeral argument belong to the same multi-pattern).

Note that this does not directly limit the set of values for instantiating the quantifier. There are still 2^{32} or 2^{64} possible values of the type `int` to instantiate the quantifier. Operationally, the `Pattern` function implements the identity function.

4.4 Run-Time-Guided Pattern Matching

As we outlined, modern SMT solvers use E-matching for instantiating patterns. E-matching uses congruence relations between ground terms. During search for unsatisfiability or satisfiability, congruence relations encode equalities that hold under all possible interpretations of the current state of the search. We will here deviate from this use of congruence relations for finding pattern matches. The basic observation is that, instead of searching through a set of different congruence relations, we use the model produced by a concrete execution for identifying a (coarse) partition of terms appearing in the corresponding symbolic trace. Two terms in a symbolic trace are treated as potentially equal if their run-time values are equal. E-matching can now be replaced by a pattern matching function that uses run-time values. We call this version the $\mathcal{M}$-*matching* problem, where $\mathcal{M}$ refers to a model. The $\mathcal{M}$-matching problem is more precisely, *given* a model $\mathcal{M}$, that provides an interpretation for a set of terms $\mathcal{T}$, and a ground term $t \in \mathcal{T}$ and pattern p; *provide* the set of substitutions θ mapping variables in p to terms in $\mathcal{T}$, such that $\mathcal{M} \models t \simeq \theta(p)$. Notice that $\mathcal{M}$-matching is an approximation of E-matching, since $E \models t \simeq \theta(p)$ and $\mathcal{M} \models E$ implies that $\mathcal{M} \models t \simeq \theta(p)$. On the other hand, $\mathcal{M}$-matching allows going beyond congruences of uninterpreted function symbols: the model provided by a concrete run provides interpretations to arbitrary functions. $\mathcal{M}$-matching may be implemented in a way similar to E-matching, using code-trees [12], but using a model $\mathcal{M}$ instead of relying on a congruence closure.

The main steps used by the $\mathcal{M}$-matching algorithm are summarized below.

1. Let $\mathcal{T}$ be the set of all terms appearing in a symbolic state.
2. Let p be a pattern we wish to match with terms in $\mathcal{T}$.

3. Recursively, match function symbols used in p with all possible matching symbols from $\mathcal{T}$. For example, if p is of the form $f(p_1, p_2)$, then select every occurrence of $f(t_1, t_2)$ in $\mathcal{T}$ and create the sub-matching problems p_1, t_1 and p_2, t_2.
4. If, in the recursive matching step, p_i is ground, then check if the matching terms $p_i^{\mathcal{M}}$ equals $t_i^{\mathcal{M}}$. If, p_i is a bound variable x, then bind the value $t_i^{\mathcal{M}}$ to x if x has not been bound before. If x was bound before to $t_j^{\mathcal{M}}$, then check $t_i^{\mathcal{M}} = t_j^{\mathcal{M}}$.

A match succeeds if the concrete run-time values coincide. The symbolic representations may be different, but the use of run-time values ensures that every match that may be valid at a give execution point is found by using the run-time values. Thus, this process may be used for supplying a superset of useful values for parameters to quantified contracts. Our method restricts quantifier instantiation to observed values. Symbolic values that are not observed are don't cares from the point of view of the program under test, so we admit test inputs that violate contracts on unobserved values.

Example 3. Suppose we seek to match a pattern of the form:

$$w * ((w + u) + V)$$

where w is an identifier used in a program and V is the bound variable of a quantifier. Consider the program fragment:

```
y = u + 4;
w = 4;
if (x == y - u) {
     u = x * (y + z);
     . . .
}
```

We match the pattern $w * ((w + u) + V)$ against the term $x * ((u + 4) + z)$ which got built by expanding the assignment to y by $u + 4$. Matching proceeds by following the structure of the pattern:

$$\begin{aligned} &Match(w * ((w + u) + V), x * ((u + 4) + z)) = \\ &\quad Match(w, x) \text{ and} \\ &\quad Match(((w + u) + V), ((u + 4) + z)) \end{aligned}$$

Since the pattern occurs only under the if-condition $(x == y - u)$ it must be the case that the run-time value of both x and w is 4. So by using the run-time values, the first match reduces to

$$Match(4, 4)$$

which holds. The second call to $Match$ reduces to:

$$\begin{aligned} &Match((w + u), (u + 4)) \\ &Match(V, z) \end{aligned}$$

Where the first matching obligation can be solved by looking at the run-time values of w and u:

$Match(8, 8)$

The second matching obligation binds the variable V to z.

Models induced by run-time values will produce a possibly coarser partition than a corresponding congruence closure, so more terms may be identified as matches than really exist. We can compensate for the approximation by adding a side-condition to the instantiated quantifier. Namely, if

$$\forall x \,.\, pat(x) : \phi(x)$$

is a quantifier with bound variable x and pattern $pat(x)$, and the symbolic term t is identified as a run-time match of $pat(x)$, with the instantiation $x \leftarrow s$, then we can create the instantiated formula:

$$pat(s) \simeq t \rightarrow \phi(s)$$

5 Related Work

Automated testing has been used in the past to guide the refinement of invariants when proof attempts fail [11], however their work was not applied to design-by-contract specifications.

The use of specifications as test oracle to decide the result for a particular test case is a well-known technique. This idea was first explored by Peters and Parnas [31]. Many approaches have followed this technique to run-time check design-by-contract specifications for JML [6], Eiffel [27] and Spec# [3]. Design-by-contract specifications have also been used for test generation using more or less random approaches to the test input generation problem, e.g. for JML [8,7] and Eiffel [9,10]. Notably, their approaches do not handle unbounded quantifiers. Unlike existing approaches, we provide a way to evaluate unbounded quantifiers (or quantifiers over an impractically large domain).

The idea of symbolic execution was pioneered by [24]. Dynamic symbolic execution was first suggested in DART [22]. Their tool analyzes C programs. Several related approaches followed [33,5,23]. They differ between each other in the extent of how much concrete information is lifted in the analysis and how much is treated symbolically, i.e. the extent of the under-approximation that they perform. We describe how to extend dynamic symbolic execution with a symbolic treatment of quantifiers.

Contracts can be used to make dynamic symbolic execution more modular and thus scalable [30]. Several attempts have been made to even infer such contracts dynamically [21,1].

Our approach relies on using the SMT solver Z3 to generate inputs that drive a program into its different reachable configurations. An overview of related work on E-matching is detailed in [12].

6 Conclusion

This paper described an approach for using dynamic testing for debugging deductive verification of contracts with quantifiers. We extended symbolic execution to handle

unbounded quantifiers. We translated quantifiers to non-deterministic programs and introduced $\mathcal{M}$-matching as a technique for finding quantifier instances among symbolic values exercised in a run. Future work includes assessing scalability and the coverage exercised by the quantifier instances.

Acknowledgments

We would like to thank Ernie Cohen, Herman Venter, and Songtao Xia for the discussions and their support.

References

1. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Proc. of TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)
2. Ball, T., Lahiri, S.K., Musuvathi, M.: Zap: Automated theorem proving for software analysis. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 2–22. Springer, Heidelberg (2005)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
5. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: CCS 2006: Proceedings of the 13th ACM conference on Computer and communications security, pp. 322–335. ACM Press, New York (2006)
6. Cheon, Y.: A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, The author's Ph.D. dissertation. (April 2003), `http://archives.cs.iastate.edu`
7. Cheon, Y.: Automated random testing to detect specification-code inconsistencies. Technical report, Department of Computer Science The University of Texas at El Paso, 500 West University Avenue, El Paso, Texas, USA (2007)
8. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Proc. 16th European Conference Object-Oriented Programming, pp. 231–255 (June 2002)
9. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Experimental assessment of random testing for object-oriented software. In: ISSTA 2007: Proceedings of the 2007 international symposium on Software testing and analysis, pp. 84–94. ACM, New York (2007)
10. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Artoo: adaptive random testing for object-oriented software. In: ICSE 2008: Proceedings of the 30th international conference on Software engineering, pp. 71–80. ACM, New York (2008)
11. Claessen, K., Svensson, H.: Finding counter examples in induction proofs. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 48–65. Springer, Heidelberg (2008)
12. de Moura, L., Bjørner, N.: Efficient E-matching for SMT Solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007)
13. de Moura, L., Bjørner, N.: Model-based Theory Combination. Electron. Notes Theor. Comput. Sci. 198(2), 37–49 (2008)
14. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver (2007), `http://research.microsoft.com/projects/Z3`

15. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963. Springer, Heidelberg (2008)
16. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
17. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. ACM 27(4), 758–771 (1980)
18. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
19. Flanagan, C., Joshi, R., Saxe, J.B.: An explicating theorem prover for quantified formulas. Technical Report HPL-2004-199, HP Laboratories, Palo Alto (2004)
20. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
21. Godefroid, P.: Compositional dynamic test generation. In: Proc. of POPL 2007, pp. 47–54. ACM Press, New York (2007)
22. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. SIGPLAN Notices 40(6), 213–223 (2005)
23. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Proceedings of NDSS 2008 (Network and Distributed Systems Security), pp. 151–166 (2008)
24. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
25. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University (June 1998)
26. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress, pp. 21–28 (1962)
27. Meyer, B.: Eiffel: The Language. Prentice Hall, New York (1992)
28. Moskal, M., Lopuszański, J.: Fast quantifier reasoning with lazy proof explication (2006), `http://nemerle.org/malekith/smt/smt-tr-1.pdf`
29. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: 38th Design Automation Conference (DAC 2001) (2001)
30. Mouy, P., Marre, B., Williams, N., Gall, P.L.: Generation of all-paths unit test with function calls. In: Proceedings of ICST 2008 (International Conference on Software Testing, Verification and Validation), pp. 32–41 (2008)
31. Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. IEEE Trans. Softw. Eng. 24(3), 161–173 (1998)
32. Pex development team. Pex (2007), `http://research.microsoft.com/Pex`
33. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proc. of ESEC/FSE 2005, pp. 263–272. ACM Press, New York (2005)
34. Tillmann, N., de Halleux, J.: Pex – white box test generation for .NET. In: Proc. of Tests and Proofs (TAP 2008), Prato, Italy, April 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
35. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970, pp. 466–483. Springer, Heidelberg (1983)