Verification Condition Generation for Permission Logics with Abstraction Functions

Author(s):
Heule, Stefan; Kassios, Ioannis T.; Müller, Peter; Summers, Alexander J.

Publication Date:
2012

Permanent Link:
https://doi.org/10.3929/ethz-a-007313727

Rights / License:
In Copyright - Non-Commercial Use Permitted
Verification Condition Generation for Permission Logics with Abstraction Functions

Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers

ETH Zurich, Switzerland
sthule@ethz.ch,
{ioannis.kassios,peter.mueller,alexander.summers}@inf.ethz.ch

Abstract. Abstract predicates are the primary abstraction mechanism for program logics based on access permissions, such as separation logic and implicit dynamic frames. In addition to abstract predicates, it is often useful to also support classical abstraction functions, for instance, to encode side-effect free methods of the program and use them in specifications. However, combining abstract predicates and abstraction functions in a verification condition generator leads to subtle interactions, which complicate reasoning about heap modifications. Such complications may compromise soundness or cause divergence of the prover in the context of automated verification. In this paper, we present an encoding of abstract predicates and abstraction functions in the verification condition generator Boogie. Our encoding is sound and handles recursion in a way that is suitable for automatic verification using SMT solvers. It is implemented in the automatic verifier Chalice.

1 Introduction

Program logics based on access permissions such as separation logic [17] and implicit dynamic frames [19] are the foundation of many program verifiers for heap-manipulating programs [2, 6, 8, 13, 18]. They associate an access permission with each heap location and enforce that a method accesses a location only if it has the permission to do so. To enable modular verification, each method specification states which permissions the method requires from its caller (in its precondition) and returns to its caller (in its postcondition). Upon a call, the caller relinquishes the required permissions (we say the caller exhales the precondition) and transfers them to the callee (the callee inhales them). Conversely, when a method terminates, the method exhales its postcondition, while its caller inhales it. This technique simplifies framing; as long as a method holds on to the permission for a location (that is, does not exhale it), no other method can access that location and, thus, its value remains unchanged.

Abstract Predicates. Simply enumerating all locations for which a method requires or returns permissions would violate information hiding and would not be possible for recursive data structures. For instance, a method that traverses
class List {
    var value: int;
    var next: List;

    predicate valid { acc(value) && acc(next) && (next ≠ null ⇒ next.valid) }

    function length(): int
        requires valid;
        ensures result > 0;
    { unfolding valid in next = null ? 1 : 1 + next.length() }

    function itemAt(i: int): int
        requires valid && 0 ≤ i;
    { unfolding valid in i = 0 || next = null ? value : next.itemAt(i-1) }

    method set(i: int, v: int)
        requires valid && 0 ≤ i && i < length();
        ensures valid && itemAt(i) = v;
    { unfold valid;
        if (i = 0) { value := v; }
        else { call next.set(i-1, v); }
        fold valid;
    }
}

Fig. 1. A Chalice [13] implementation of a singly-linked list. To support modular verification, methods have preconditions (keyword requires) and postconditions (keyword ensures). In addition to regular methods, Chalice supports side-effect free functions, which may be used in specifications. An access permission to a field x.f is denoted by acc(x.f), which corresponds to x.f ↦ _ in separation logic. The Chalice conjunction && treats permissions multiplicatively (i.e., requiring the sum of the permissions in each conjunct), similarly to the separating conjunction * of separation logic. The recursive abstract predicate valid represents the memory locations of the list structure. The unfold and fold ghost statements replace a predicate by its body and vice versa. The ghost expression construct unfolding...in is analogous to an unfold-fold block and can be used in functions and in specifications, where statements cannot occur.

a linked list such as method itemAt in Fig. 1 would require permission to access this.value, this.next, this.next.value, this.next.next, and so on. To solve this problem, Parkinson and Bierman [14] introduced abstract predicates. An abstract predicate is defined in terms of concrete memory locations with their values and permissions, and possibly other predicates. Due to this recursion, abstract predicates represent a possibly unbounded number of permissions. For instance, class List in Fig. 1 defines an abstract predicate valid that represents the permissions for value, next and, if next is different from null, the permissions in next.valid. Just as with permissions to field locations, a method may require predicates from its caller. It may access a location if it possesses the corresponding permission, either directly or as part of a predicate. For instance, method itemAt requires the predicate valid to get access to all locations of the list structure.
In this paper, we employ ghost heap locations to describe the predicates held for a particular object (cf. Sec. 3). We use the term *location* for both concrete field locations and predicate locations. We use the term *permission* to describe permissions to both kinds of locations.

**Folding and Unfolding.** Whenever a method attempts to access a location \(x.f\), the verifier needs to check whether the method has the access permission for \(x.f\), either directly or as part of a predicate. However, since the definitions of predicates may be recursive, a verifier cannot statically compute all permissions stored in a predicate. It is also not useful to let an SMT solver reason about recursive definitions because the solver might unfold a recursive definition infinitely often (in a so-called *matching-loop*) and thus not terminate.

Some verifiers \[8, 13, 19\] solve this problem by distinguishing between a predicate and its body. Instead of unfolding predicate definitions automatically, they require programmers to use the ghost statements `unfold` and `fold` to replace a predicate by its body and vice versa. A permission that has not been folded into another predicate is called *direct*; the others are called *folded*. Accessing a field, unfolding a predicate, or exhaling a predicate all require appropriate direct permissions.

Method `set` in Fig. 1 illustrates these concepts. The method body first inhales its precondition; in particular, the predicate `valid`. After the inhale, it holds `valid` as a direct predicate, whereas the permissions to `value` and `next` (as well as the fields of the rest of the list) are folded. Consequently, before accessing these locations, the method must unfold `valid`. The `fold` statement at the end of the method is necessary to re-obtain direct permission to `valid`, which gets exhaled as part of the postcondition.

**Abstraction Functions.** Permission logics often employ parametrised abstract predicates, where the parameter represents an abstracted value of the data structure. For instance, one could replace the `valid` predicate for linked lists by a predicate `List(l)`, where \(l\) is a mathematical sequence. The definition of the predicate would then not only contain the access permissions, but also relate the values stored in the concrete list to \(l\).

Parametrised abstract predicates are a powerful abstraction mechanism. However, there are several advantages to supporting *classical abstraction functions* in place of parametrised abstract predicates. Most data structure implementations include side-effect-free inspector methods (or *functions* in Chalice) such as `itemAt` and `length` in our example. It is convenient to re-use these methods in specifications; side-effect free methods can be encoded naturally as abstraction functions \[5\]. Furthermore, specifications written without abstraction functions typically use logical variables for the parameters of an abstract predicate, which can then be used in postconditions to describe how the abstract value has changed. Finding witnesses for these logical variables is not supported well by SMT solvers. By contrast, abstraction functions can be used within `old` expressions to refer to their pre-state evaluation, avoiding logical variables.
In a framework supporting abstraction functions and non-parameterised predicates, such as that of Chalice, the primary role of predicates such as \texttt{valid} in Fig. 1 is to abstract over access permissions, whereas functions such as \texttt{length} and \texttt{itemAt} abstract over the contents of a data structure.

**Contributions.** The main contribution of this paper is an encoding of abstract predicates and abstraction functions for verification condition generators that is sound and amenable to automation via SMT solvers. In particular, our approach avoids giving recursive axioms to the prover, which would lead to matching loops. Some verifiers based on symbolic execution \cite{19,9} support both abstract predicates and abstraction functions, and many more support only the former (e.g., \cite{6,8,2}). However, in the equally important domain of verifiers based on verification condition generation, no solution currently exists that both avoids matching loops and is sound. We present our solution for implicit dynamic frames \cite{19}, but it also applies to other permission logics, such as separation logic. We have implemented our solution in a new version of Chalice.

**Outline.** A sound and practical verification technique must address the following two challenges:

1. **What to forget?** When a method exhales the permission for a field, it must invalidate any information about the value stored in the location. Since another method may obtain the permission and modify the location, it would be unsound to retain any information. A verification condition generator must invalidate enough information to be sound, even though it cannot determine precisely which permissions are folded into a (recursive) predicate.

2. **What to remember?** A practical verifier has to preserve as much information about the values of locations and functions as (soundly) possible. The framing of locations is difficult because a verifier might invalidate information too aggressively. The framing of functions is difficult because a verifier cannot determine precisely which locations a function depends on.

We present our solution to these challenges informally in Secs. 2 and 3, respectively. The details of our encoding are explained in Sec. 4, and we argue why our solution is sound in Sec. 5. We discuss related work in Sec. 6 and conclude in Sec. 7.

2 **What to Forget: Havocing**

In this section, we describe how we address the first challenge listed above: how to invalidate sufficient information when permissions are exhaled.
**Total Heaps.** In permission-based verification techniques, a method has access to those heap locations for which it holds permission. This semantics can be modelled via *partial heaps* [15] that map only these locations to values. Since SMT solvers support total functions, verification condition generators cannot directly use partial heaps. Instead, they use a *total heap*, which maps all heap locations to values, together with a *permission mask*, which maps heap locations to the permissions that the current method holds [12]. Proof obligations ensure that the method accesses only those locations in the total heap for which the permission mask contains a permission.

In contrast to partial heaps, a total heap contains values even for those locations to which a method invocation does not currently have permission. Therefore, when a method exhales the permission for a location $x.f$, the verifier has to explicitly invalidate any recorded information about $x.f$ since other methods may now obtain the permission and modify $x.f$. The information is invalidated by assigning an arbitrary value to $x.f$ in the total heap when the permission to $x.f$ is exhaled; we say that $x.f$ is *havoced*.

**Havocing with Folded Permissions.** In the absence of abstraction functions, explicit folding and unfolding is sufficient to handle recursive predicates. A verifier can then ignore folded permissions entirely. In particular, it can havoc a location $x.f$ whenever the method to be verified releases the direct permission to $x.f$, including the case in which the method folds the permission into a predicate. Consequently, when the predicate is exhaled later, no additional havocing is necessary because all locations that are (directly or transitively) folded into the predicate have already been havoced.

However, abstraction functions provide a way of inspecting memory locations whose permissions have been folded into a predicate. For instance, function `itemAt` yields the value of a `value` field whose permission is folded (possibly deeply) into the `valid` predicate. Consequently, havocing a location when its permission gets folded into a predicate is no longer useful. A function that inspects this location afterwards will return an arbitrary value, which defeats the purpose of abstraction functions. For instance, the call to `itemAt` in the postcondition of `set` would yield an undefined value, and the postcondition could not be verified.

Abstraction functions prevent havocing locations when their *direct* permissions are folded. On the other hand, soundness requires that they get havoced at least when their *folded* permissions are exhaled. However, as we explained above, a verification condition generator based on SMT solvers cannot precisely determine all the permissions that are (transitively) folded into a predicate and, thus, which locations to havoc when the predicate is exhaled. We solve this dilemma by havocing very aggressively. Each time a predicate is exhaled, we havoc all memory locations for which the method does not retain a direct permission. This solution is sound since it havoc all locations whose permissions are folded, not just the ones that are inside the predicate given away. We will discuss in the next section how to make this crude approximation more precise, by recovering field and function values to make the verification sufficiently complete.
3 What to Remember: Framing Locations and Functions

Aggressively havocing all information about heap locations to which no direct permission is held, is sound but too restrictive. When a method holds direct permission to a location and observes its value, it is important to preserve this heap information when the permission is folded into a predicate, as long as the predicate is held by the method. This information is for instance necessary to prove postconditions that depend on the value via function applications or unfolding expressions. Furthermore, since function preconditions are typically stated in terms of predicates, function values depend on heap locations to which folded permission is held. So, when aggressively havocing in this way, all information about such function values is lost. In this section, we tackle these two sources of incompleteness.

3.1 Framing of Locations

To preserve observed location values across an exhale, even if the method holds only folded permission to them, we introduce an under-approximation of the folded permissions that can be computed by a verification condition generator. This under-approximation contains those folded permissions that the current method has folded itself, either through a fold statement or through an unfolding expression. We call these permissions known-folded permissions.

Our technique guarantees that in any state, the known-folded permissions are a subset of the folded permissions held by the current method. It is therefore sound to havoc less aggressively when exhaling a predicate. We havoc only those locations for which the method holds neither direct nor known-folded permission.

To maintain the known-folded permissions, we adapt the behaviour of the fold and unfold statements, as well as of the exhale operation as follows.

Folding a predicate \( p \) checks that the method holds the direct permissions in the body of \( p \), then moves these permissions from the direct permissions to the known-folded permissions, and finally adds direct permission to \( p \). The evaluation of an unfolding expression has the same effect: all permissions that were temporarily unfolded are moved to the known-folded permissions.

Unfolding a predicate \( p \) checks that the method holds direct permission to \( p \), removes this permission, and then adds the direct permissions in the predicate body. In case the permissions in the body are known-folded, they are removed from the known-folded permissions since they are no longer folded.

Exhaling a predicate \( p \) must not only remove direct permission to \( p \), but also remove all permissions (transitively) folded into \( p \) from the known-folded permissions. Determining these permissions for recursive predicates is tricky and requires additional book-keeping whenever the verification condition generator manipulates the known-folded permissions, as we explain in Sec. 4.

Fig. 2 illustrates how fold, unfold, exhale, and inhale operate on some of the possible states of the valid predicate. The upper row depicts states in which the values of fields value and next have not been observed; in the upper right trapezoid, the permissions in the body of valid are folded, but not known-folded.
Fig. 2. Each trapezoid depicts all permissions held at a particular point in a method invocation, where the white regions contain direct permissions and the gray areas depict folded permissions. Known-folded permissions appear in the light gray region with dashed border.

In the lower row the values of these fields have been observed and permission to them is either direct or known-folded, which protects the fields from being havoced by an arbitrary exhale. For instance, the set method of Fig. 1 starts in the top-left state and goes through all other states in a clock-wise manner. The method does not go back to the upper row until the predicate valid is exhaled, in which case the direct permission for valid and the known-folded permissions for its contents are removed.

Tracking known-folded permissions and excluding them from the havocing upon exhales removes the incompleteness described at the beginning of this section for location values, as the following example illustrates. Consider two distinct List objects x and y for which the valid predicate is held at the beginning of the following code:

```plaintext
var i: int := unfolding x.valid in x.value;
var j: int := unfolding y.valid in y.value;
call y.set(0,10);
assert unfolding x.valid in (i = x.value); // succeeds
assert unfolding y.valid in (j = y.value); // correctly fails to verify
```

The unfolding expression at the beginning of the code makes the permission to x.value known-folded. The call to y.set exhales y.valid and therefore havoc the state, but preserves locations for which the method holds known-folded permission. This makes the first assertion at the end of the code succeed.

On the other hand, exhaling y.valid removes the permission to y.value from the known-folded permissions, since y.value is folded under y.valid. During the havoc phase of the exhale, the value of field y.value is forgotten, and the second assertion correctly fails to verify.

### 3.2 Framing of Functions

Intuitively, the value of a function can be framed if none of the heap locations on which the function depends are modified. These locations are a subset of the
locations for which the function requires permission in its precondition. However, if the function requires a recursive predicate then a verification condition generator is not able to determine this set of locations.

Existing tools such as Spec# [1] and VeriCool [18] handle this problem by abstracting over the locations a functions depends on via versioning. The idea is as follows: if we can be sure that such predicates have been neither unfolded nor exhaled since an earlier program point, we know that all locations nested inside the predicates are also unmodified. We label the predicate with the same version to identify this case.

Predicate versions are recorded as ghost locations in the heap, and are treated as regular field locations: in particular, we can retain knowledge of a predicate version so long as we hold either direct or known-folded permission to the predicate location. Furthermore, when we hold neither direct nor known-folded permission to such a location, it will naturally be havoced during an exhale. In addition to that, the version is havoced when the predicate is unfolded.

The function framing axiom is, informally, as follows: two calls to the same function in two states evaluate to the same value if the receiver and all arguments of the function calls are the same and the two states agree on the values of all locations to which the function precondition requires direct permission, in particular, on the versions of the required predicates.

Function framing based on predicate versions removes the incompleteness described at the beginning of this section for function values, as the following example illustrates. We consider again two distinct List objects x and y for which the valid predicate is held at the beginning in the following code snippet:

```java
var i: int := x.itemAt(0);
var j: int := y.itemAt(0);
call y.set(0, 10);
assert x.itemAt(0) = i; // succeeds
assert y.itemAt(0) = j; // correctly fails to verify
```

The exhale operation of the method call gives away y.valid, thereby havocing the version of that predicate. The predicate x.valid is not affected, and keeps the same version, which allows the prover to correctly verify the first assertion, using the framing axiom. The second assertion is not necessarily true, and indeed fails to verify as the version of y.valid changed.

4 Encoding

In this section, we present an encoding of our technique in the verification condition generator Boogie [11]. Verification then consists of three steps: (1) a translator translates the source program and its specification into the Boogie language, (2) Boogie computes verification conditions, and (3) an SMT solver attempts to prove the verification conditions. Here, we focus on how the translator encodes known-folded permissions and avoids providing recursive definitions or axioms to the prover.

Our encoding represents the current heap with a variable Heap, which is a map from locations to values; that is, the heap maps field locations x.f to their
values and predicate locations \(x.p\) to their versions. The direct and known-folded permissions are tracked via two variables \(\text{DMask}\) and \(\text{KFMask}\), which are maps from locations to booleans. We call these two mask variables the direct and known-folded permission masks, respectively.

The most challenging part of the encoding is the exhale operation, which removes all known-folded permissions that are folded into the predicate that is being exhaled. To avoid having the prover reason about recursive definitions, we split the responsibility for determining which known-folded permissions to remove: the translator handles recursion, whereas the prover is used to decide aliasing and conditionals.

We have argued before that a verification condition generator cannot statically determine which permissions are (transitively) folded into a predicate. However, the situation here is different: we only have to determine which known-folded permissions are folded into a predicate. Since a method’s known-folded mask contains only those permissions that the method moved there, we can use a simple static analysis of the method implementation to resolve recursion statically.

In principle, the translator emits Boogie code that encodes an exhale operation of a predicate location \(x.p\) as follows:

1. Remove all the direct permissions in \(p\)’s body from the known-folded mask.
2. For each predicate location in the body of \(p\) for which the method holds known-folded permission, apply the translation recursively.

This translation scheme ignores two aspects. First, the translator is not able to evaluate conditionals in the program or in the definition of the predicate and therefore cannot decide for which locations the known-folded mask actually needs to be updated. Second, the translator cannot determine statically whether or not a method holds known-folded permission for a given predicate and thus does not know when to stop the recursion.

To address the first issue, we let the translator generate Boogie code that includes conditional statements to represent the various conditions that the translator cannot decide. This forces the verification condition generator to consider all possible evaluations of these conditionals.

To address the second issue, it is sufficient to find an upper bound on the recursion depth, such that the translation results in a finite sequence of Boogie statements. A suitable upper bound is the number of fold statements and unfolding expressions that occur before the exhale operation in the program text. This number is an upper bound for two reasons. First, fold and unfolding are the only operations that add known-folded permissions, and each of them adds (at most) one more predicate body to these permissions. Second, loops are verified via loop invariants that summarize the behaviour of the loop; loop bodies are verified separately. Therefore, it is sufficient to count the static occurrences of fold and unfolding in the program text, ignoring loop bodies. Note that the generated Boogie code is accurate, even if the bound is not tight. If it recurses too deeply, the check whether the method holds known-folded permission for a folded predicate will fail and, thus, no further mask updates will be performed.
fold o.p ≡
assert that body(o,p) holds in the state (Heap,DMask);
move permissions in body(o,p) from DMask to KFMask;
DMask[o,p] := true;
depth := depth + 1;

unfold o.p ≡
assert DMask[o,p]; // check that the method holds the predicate
DMask[o,p] := false;
remove permissions in body(o,p) from KFMask; // no recursion
inhale body(o,p);
havoc Heap(o,p); // the recorded version of o.p

inhale o.p ≡
DMask[o,p] := true;

exhale o.p ≡
assert DMask[o,p]; // check that the method holds the predicate
DMask[o,p] := false;
remove-contents(o.p, depth); // remove any contents from KFMask
havoc Heap at all locations except where DMask or KFMask is true

remove-contents(o.p, depth) ≡
if depth > 0 then
  for each predicate location o'.p' in body(o,p):
    if KFMask[o',p'] then remove-contents(o'.p', depth-1)
remove permissions in body(o,p) from KFMask; // no recursion

Fig. 3. Encoding of the most interesting operations. The statements on grey background are evaluated by the translator, whereas for all other steps the translator generates Boogie code to perform these steps. Variable depth is used to count the number of fold and unfolding operations to bound the recursion.

The pseudo-code in Fig. 3 summarizes the translation of the most interesting operations. To illustrate their application, consider an example of two distinct non-null List objects x and y, and the following code snippet:

fold x.next.valid;
fold x.valid;
fold y.valid;
call x.set(0, 0);

Fig. 4 shows the Boogie code that is generated for the exhale of x.valid as part of the method call. Before the point where the translator produces the Boogie output for this exhale operation, three fold statements occurred, and thus the upper bound on the recursion depth (variable depth in Fig. 3) is three. Therefore the recursive definition of remove-contents is unrolled three times, even though in this particular example two unrollings would be sufficient (assuming
we know y.next ≠ x). For the third unrolling, the prover will determine that KFMask[x.next.next,valid] is false and, thus, not perform the mask update.

```c
assert DMask[x,valid];
DMask[x,valid] := false;
if (KFMask[x,valid]) {
    if (x.next ≠ null) { // from predicate definition
        if (KFMask[x.next,valid]) {
            if (x.next.next ≠ null) { // from predicate definition
                // end of recursion, depth 3 has been reached
                KFMask[x.next.next.next,valid] := false;
            }
            // remove fields under x.next.next.valid from KFMask:
            KFMask[x.next.next,value] := false;
            KFMask[x.next.next,next] := false;
        }
        KFMask[x.next.next,valid] := false;
    }
    // remove fields under x.next.valid from KFMask:
    KFMask[x.next.value] := false; KFMask[x.next.next] := false;
}
KFMask[x.next,valid] := false;
// remove fields inside x.valid from KFMask:
KFMask[x.value] := false; KFMask[x,next] := false;
}
```

Fig. 4. Translation of the exhale x.valid operation as part of the code snippet given in the text. We write o.f as a shorthand for Heap[o,f].

5 Soundness

In this section, we give an informal justification for the soundness of our approach. Our technique provides two additional ways (beyond the usual use of permissions in the direct mask) in which we preserve information: function framing and framing using the known-folded mask. To justify their soundness, we introduce the concept of a footprint.

Definition 1 (Permission and Heap Footprints). The permission footprint of a predicate in a heap is the set of heap locations defined by recursively evaluating the predicate definition in the given heap, and collecting the locations whose permissions are required by the definition.

The permission footprint of a function in a heap is the set of heap locations defined by evaluating the function’s precondition in the given heap, and collecting the locations whose permissions the precondition requires, as well as the permission footprints of all predicates the precondition requires in that evaluation.
The heap footprint of the predicate or function in a heap is the set of location-value pairs such that the location is in the corresponding permission footprint, and the value is the value stored in the heap at that location.

**Soundness of Function Framing.** To justify our approach to function framing, it is sufficient to observe the following:

1. Evaluation of a function application (at runtime) reads only locations in its permission footprint. This property is enforced by a well-formedness check for each function definition. In particular, a function application’s value is a function of its receiver, arguments, and its heap footprint. Function bodies can also apply functions, but the checking of preconditions ensures that heap footprints for recursive applications are always subsets of the original one.

2. Like functions, predicate definitions are checked to ensure that they read only heap locations which fall within their permission footprint. Thus, the permission and heap footprints of a predicate are fixed by the permissions folded inside of it. Since we havoc version numbers whenever a predicate is unfolded or exhaled, the version of a predicate location at two different program points can only be known to be the same if the heap footprint of the predicate is also identical at both points.

3. Consider a situation in which our framing axiom allows the prover to equate a function value between two program points. By (1), we know that it is sufficient to know that the function’s heap footprint remains the same between both points. The heap footprint is made up of field locations to which explicit permission is required in the function’s precondition (which the axiom requires to have the same values), and the heap footprints of those predicate locations which the function’s precondition requires (which the axiom requires to have the same versions). By (2), the function value is the same in both states.

**Soundness of Known-Folded Mask Framing.** Our approach to framing heap information is based on the permissions stored in the direct and known-folded masks. The fact that it is sound to frame the values of heap locations for which we hold direct permission follows from the basic permission model without predicates and functions, which we do not address here (see [16] for a soundness proof for a similar system). To justify that it is also sound to frame information based on known-folded permissions, we need to show that the permissions stored in the known-folded mask are indeed folded (perhaps recursively) inside predicates to which the method holds a direct permission. In this case, no other method may hold these permissions and modify the corresponding locations.

We first define concretely what it means for a location or predicate to be folded (transitively) inside a predicate that a method holds. The following definition uses the notion of a direct permission footprint of a predicate, which is the set of locations (fields and predicates) that are explicitly mentioned in the definition of a predicate (without unfolding nested predicates).

**Definition 2 (Contained Locations and Permissions).** Let $P$ be a program point. Let depth be the number of fold and unfolding keywords from the beginning
of the code fragment that is currently under verification until \( P \). A location \( o_1.p_1 \) is contained at program point \( P \) and state \((\text{Heap}, \text{DMask}, \text{KFMask})\) if there exists a (possibly empty) sequence of predicate locations \( o_2.p_2, \ldots, o_n.p_n \) such that:

1. \( n \leq \text{depth} + 1 \)
2. \( \text{DMask}[o_n, p_n] \) is true.
3. \( \forall 1 \leq i < n, \text{KFMask}[o_i, p_i] \) is true.
4. \( \forall 1 < j \leq n, \text{perm}(o_j.p_j, o_{j-1}.p_{j-1}) \) is in the direct permission footprint of \( o_j.p_j \) in the given state.

We write \( \text{contains}(o_1.p_1, (\text{Heap}, \text{DMask}, \text{KFMask})) \), when this property holds.

The key property for soundness is that all permissions held in the known-folded mask are contained in the state:

**Theorem 1 (Soundness of Framing with Known-Folded Permissions).**
In a verified program, for all locations \( o.p \), if either \( \text{DMask}[o, p] \) or \( \text{KFMask}[o, p] \) is true then \( \text{contains}(o.p, (\text{Heap}, \text{DMask}, \text{KFMask})) \) holds.

**Proof Sketch.** We show that the property is preserved across the four operations that affect the known-folded mask: folding, unfolding, inhaling, and exhaling predicates.

**Folding** \( o.p \): This operation moves the permissions required by the body of \( o.p \) from \( \text{DMask} \) to \( \text{KFMask} \). It may affect \( \text{contains}(o'.p', (\text{Heap}, \text{DMask}, \text{KFMask})) \) at most for those locations \( o'.p' \) for which the sequence of predicate locations used to end with some predicate required by the body of \( o.p \). However, because we set \( \text{DMask}[o, p] \) to true and because the number \( \text{depth} \) of Def. 2 is increased by one after the fold operation, the contains-property is preserved for such cases, by extending the previous sequence with the newly-folded predicate location \( o.p \).

**Unfolding** \( o.p \): This operation sets \( \text{DMask}[o, p] \) to false. It falsifies the contains-property for \( o.p \), which preserves the soundness property since we no longer hold permission to this predicate location in either mask. The operation could also potentially falsify the contains-property for any \( o'.p' \) whose corresponding sequence of predicate locations terminated in \( o.p \). However, the penultimate predicate location in this sequence must, by Def. 2, have been in the direct permission footprint of \( o.p \), and thus, permission to this predicate location gets moved from \( \text{KFMask} \) to \( \text{DMask} \). Thus, by dropping the last element of the sequence, we preserve the property for all such \( o'.p' \).

**Inhaling** \( o.p \): This operation sets \( \text{DMask}[o, p] \) to true and, thus, the contains-property holds trivially for \( o.p \) (just taking a sequence of length 1). All other permissions in \( \text{DMask} \) and \( \text{KFMask} \) remain the same.

**Exhaling** \( o.p \): This operation sets \( \text{DMask}[o, p] \) to false. It falsifies the contains-property for any \( o'.p' \) whose corresponding sequence of predicate locations terminated in \( o.p \). However, our generated if-conditionals recursively traverse these folded locations, and remove them all from \( \text{KFMask} \). Points 1 and 4 of Def. 2 guarantee that the generated conditionals will cover all of these locations, while point 3 guarantees that all of the appropriate conditionals will evaluate to true.
6 Related Work

Verification Condition Generation (VCG) is a popular technique for the construction of automated verifiers [7, 11, 1, 4, 10, 12]. The concept of abstract predicates was introduced by [14] and it is used in separation logic [17] and implicit dynamic frames [19]. Abstraction functions are used together with abstract predicates in Chalice [12] and VeriCool [18].

The VCG implementation of VeriCool removes the distinction between the folded and the unfolded state of a predicate (i.e., a predicate and its body), which addresses the problem soundly but, unlike our solution, introduces the possibility of matching loops in the operation of the SMT solver, as is explained in the introduction.

The previous Chalice encoding of predicates and functions is unsound. The encoding used a single permission mask to track direct permissions. The heap was havoced *lazily*, i.e., when permission is (re-)obtained. This ensures that values of fields are preserved, but also leaves invalid information in the heap, which caused the unsound behaviour. In particular, folded locations were never havoced.

Symbolic execution is an alternative technique to VCG and is used in tools such as [2, 6, 8, 19, 9]. Typically, symbolic execution engines use partial heaps and other more elaborate data structures for the representation of the program state. In the presence of such data structures, the problem treated in this paper is not as intricate. In particular, symbolic execution engines can iterate through their heap representation to determine folded permissions, whereas for VCG with SMT solvers, this is not possible in the presence of recursive predicates. Symbolic execution forgets heap information by chopping off the corresponding part from the partial heap. Framing of function values can be sufficiently handled as in VCG by predicate versioning.

7 Conclusion

In this paper, we have discussed the problem of supporting abstraction functions and permission-based logics in a VCG setting. We have demonstrated that the problem has subtle difficulties, and presented a detailed solution. Our main idea is to ensure soundness through aggressive havocing, while supporting an extra permission mask that protects information that we have previously observed, from the havocs. To our knowledge, this is the first sound encoding which protects the underlying prover from matching loops.

Our solution is implemented in Chalice. Presently, it can be tried out online at [http://129.132.31.86:8080/wits/](http://129.132.31.86:8080/wits/). Chalice supports fractional permissions [3], and we extended our approach to support this by having numeric values for both permission masks. The additional complexity to soundly handle predicates and functions did not cause a significant decrease in performance compared to the previous (unsound) version of Chalice; for the official test suite of Chalice, the slowdown is 10.7% on average.
Acknowledgements. We would like to thank Jan Smans for explaining to us the operation of VeriCool and Malte Schwerhoff for various discussions and for making our implementation available online.

References