VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java

Bart Jacobs, Jan Smans*, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens

Department of Computer Science, Leuven, Belgium
firstname.lastname@cs.kuleuven.be

Abstract. VeriFast is a prototype verification tool for single-threaded and multithreaded C and Java programs. In this paper, we first describe the basic symbolic execution approach in some formal detail. Then we zoom in on two technical aspects: the approach to permission accounting, including fractional permissions, precise predicates, and counting permissions; and the approach to lemma function termination in the presence of dynamically-bound lemma function calls. Finally, we describe three ongoing efforts: application to JavaCard programs, integration of shape analysis, and application to Linux device drivers.

1 Introduction

VeriFast is a prototype verification tool for single-threaded and multithreaded C and Java programs annotated with preconditions and postconditions written in separation logic. To enable rich specifications, the programmer may define inductive datatypes, primitive recursive pure functions over these datatypes, and abstract separation logic predicates. To enable verification of these rich specifications, the programmer may write *lemma functions*, i.e., functions that serve only as proofs that their precondition implies their postcondition. The verifier checks that lemma functions terminate and do not have side-effects. Since neither VeriFast itself nor the underlying SMT solver need to do any significant search, verification time is predictable and low. VeriFast comes with an IDE that enables interactive annotation insertion and symbolic debugging and is available for download at http://www.cs.kuleuven.be/~bartj/verifast/.

For an introduction to VeriFast, we refer to earlier work [1]; furthermore, a tutorial text is available on the web site. In this invited paper, we take the opportunity to zoom in on three aspects of VeriFast that have not yet been covered in the same level of detail in earlier published work: in Section 2 we present in some formal detail the essence of VeriFast's symbolic execution algorithm; in Section 3 we present VeriFast's support for permission accounting; and in Section 4 we present our approach for ensuring termination of lemma functions that perform dynamically bound calls. Additionally, in Section 5, we briefly discuss some of the projects currently in progress at our group.

* Jan Smans is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

2 Symbolic Execution

In this section, we present the essence of VeriFast's verification algorithm in some formal detail.

2.1 Symbolic Execution States

VeriFast modularly verifies a C or Java program by symbolically executing each routine (function or method) in turn, using other routines' contracts to verify calls. A symbolic execution state is much like a concrete execution state, except that terms of an SMT solver, containing logical symbols, are used instead of concrete values. For example, at the start of the symbolic execution of a routine, each routine parameter's value is represented using a fresh logical symbol.

Specifically, a symbolic state $\sigma = (\Sigma, h, s)$ consists of a *path condition* Σ, a *symbolic heap* h, and a *symbolic store* s. The path condition is a set of formulae of first-order logic that constrain the values of the logical symbols that appear in the symbolic heap and the symbolic store. The symbolic heap is a multiset of *heap chunks*. Each heap chunk is of the form $[f]p\langle\overline{\tau}\rangle(\overline{t})$, where f is the *coefficient*, p the *predicate name*, $\overline{\tau}$ the *type arguments*, and $\overline{t}$ the *arguments* of the chunk. The coefficient f is a term representing a real number; if it is different from 1, the chunk represents a fractional permission (see Section 3). The predicate name is a term that denotes the predicate of which the chunk is an instance; it is either the symbol associated with a built-in predicate, such as a struct or class field predicate, or a user-defined predicate, or it is a *predicate constructor application*, which is essentially a partially applied predicate (see [2] for more information), or it is some other term, which typically means the predicate name was passed into the function as a value. VeriFast supports type parameters on user-defined predicates; hence the type arguments, which are VeriFast types. Finally, each chunk specifies argument terms for the predicate's parameters. The symbolic store maps local variable names to terms that represent their value.

2.2 Algorithm: Preliminary Definitions

To describe the essence of the symbolic execution algorithm formally, we define a highly stylized syntax of assertions a, commands c, and routines r (given an unspecified syntax for arithmetic expressions e and boolean expressions b, and given a set of variables x):

$$\begin{aligned} a &::= [e]e(e, ?x) \mid b \mid a * a \mid \textbf{if } b \textbf{ then } a \textbf{ else } a \\ c &::= x := r(\overline{e}) \mid (c; c) \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \\ \mathit{rdef} &::= \textbf{routine } r(\overline{x}) \textbf{ req } a \textbf{ ens } a \textbf{ do } c \end{aligned}$$

We assume all predicates have exactly two parameters, and we consider only predicate assertions where the first argument is an expression and the second argument is a pattern.

We will give the semantics of a symbolic execution step by means of symbolic transition relations, which are relations from initial symbolic states σ to outcomes o. An outcome is either a final symbolic state or the special outcome **abort**, which signifies that an error was found. A given initial state may be related to multiple outcomes due to case splitting, and it may be related to no outcomes if all symbolic execution paths are found to be infeasible.

VeriFast sometimes makes arbitrary choices. Specifically, it arbitrarily chooses a matching chunk when consuming a predicate assertion that has multiple matching chunks (a situation we call an *ambiguous match*). To model this, we define the semantics of a symbolic execution step not as a single transition relation, but as a set of transition relations. Each element of this set makes different choices in the event of ambiguous matches. The soundness theorem states that if, for a given initial symbolic state, there is any transition relation where this initial state does not lead to an **abort**, then all concrete states represented by this initial symbolic state are safe. It is possible that some choices cause VeriFast to fail (i.e., lead to an **abort**), while others do not. It is up to the user to avoid such unfortunate matches, for example by wrapping chunks inside predicates defined just for that purpose to temporarily hide them.

A note about picking fresh logical symbols. We will use the function

$$\mathsf{nextFresh}(\Sigma) = (u, \Sigma')$$

which given a path condition Σ returns a symbol u that does not appear free in Σ, and a new path condition $\Sigma' = \Sigma \cup \{u = u\}$, which is equivalent to Σ but in which u appears free. Since path conditions are finite sets of finite formulae, and there are infinitely many logical symbols, this function is well-defined. We will also use this function to generate sequences of fresh symbols.

We use the following operations on sets of transition relations. Conjunction $W \wedge W'$ denotes the pairwise union of relations from W and W':

$$W \wedge W' = \{R \cup R' \mid R \in W \wedge R' \in W'\}$$

Similarly, generalized conjunction $\bigwedge i \in I.\ W(i)$ denotes the set where each element is obtained by taking the union of one element of each $W(i)$:

$$(\bigwedge i \in I.\ W(i)) = \{\bigcup i \in I.\ \psi(i) \mid \forall i \in I.\ \psi(i) \in W(i)\}$$

We omit the range I if it is clear from the context. Sequential composition $W; W'$ denotes the pairwise sequential composition of relations from W and W':

$$W; W' = \{R; R' \mid R \in W \wedge R' \in W'\}$$

where the sequential composition of transition relations $R; R'$ is defined as

$$R; R' = \{(\sigma, \mathbf{abort}) \mid (\sigma, \mathbf{abort}) \in R\} \cup \{(\sigma, o) \mid (\sigma, \sigma') \in R \wedge (\sigma', o) \in R'\}$$

We denote the term or formula resulting from evaluation of an arithmetic expression e or boolean expression b under a symbolic store s as $s(e)$ or $s(b)$, respectively. We abuse this notation for sequences of expressions as well.

2.3 The Algorithm

A basic symbolic execution step is an *assumption step* $\mathsf{assume}(b)$, defined as follows:

$$\mathsf{assume}(b) = \{\{((\Sigma, h, s), (\Sigma \cup \{s(b)\}, h, s)) \mid \Sigma \not\vdash_{\mathrm{SMT}} \neg s(b)\}\}$$

It consists of a single transition relation, which adds b to the path condition, unless doing so would lead to an inconsistency, in which case the symbolic execution path ends (i.e., the initial state does not map to any outcome).

Symbolic execution of a routine starts by *producing* the precondition, then verifying the body, and finally *consuming* the postcondition. Producing an assertion means adding the chunks and assumptions described by the assertion to the symbolic state:

$$\begin{array}{l}
\mathsf{produce}([e]e'(e'', ?x)) \;= \\
\quad \{\{((\Sigma, h, s), (\Sigma', h \uplus \{[s(e)]s(e')(s(e''), u)\}, s[x := u])) \mid \\
\qquad (u, \Sigma') = \mathsf{nextFresh}(\Sigma)\}\} \\
\mathsf{produce}(b) = \mathsf{assume}(b) \\
\mathsf{produce}(a * a') = \mathsf{produce}(a); \mathsf{produce}(a') \\
\mathsf{produce}(\textbf{if } b \textbf{ then } a \textbf{ else } a') = \mathsf{assume}(b); \mathsf{produce}(a) \wedge \mathsf{assume}(\neg b); \mathsf{produce}(a')
\end{array}$$

Conversely, consuming an assertion means removing the chunks described by the assertion from the symbolic heap, and checking the assumptions described by the assertion against the path condition.

$$\begin{array}{l}
\mathsf{consume}([e]e'(e'', ?x)) \;= \\
\qquad \mathsf{choice}(\{\mathsf{matches}(\Sigma, h, s) \mid \mathsf{matches}(\Sigma, h, s) \neq \emptyset\}) \\
\quad \wedge \{\{((\Sigma, h, s), \textbf{abort}) \mid \mathsf{matches}(\Sigma, h, s) = \emptyset\}\} \\
\quad \text{where } \mathsf{choice}(C) = \{\{\psi(c) \mid c \in C\} \mid \forall c \in C.\ \psi(c) \in c\} \\
\quad \text{and } \mathsf{matches}(\Sigma, h, s) \;= \\
\qquad \{((\Sigma, h, s), (\Sigma, h', s[x := t'])) \mid \\
\qquad\quad h = h' \uplus \{[f]p(t, t')\} \wedge \Sigma \vdash_{\mathrm{SMT}} s(e, e', e'') = f, p, t\} \\
\mathsf{consume}(b) \;= \\
\quad \{\{((\Sigma, h, s), (\Sigma, h, s)) \mid \Sigma \vdash_{\mathrm{SMT}} s(b)\} \cup \{((\Sigma, h, s), \textbf{abort}) \mid \Sigma \not\vdash_{\mathrm{SMT}} s(b)\}\} \\
\mathsf{consume}(a * a') = \mathsf{consume}(a); \mathsf{consume}(a') \\
\mathsf{consume}(\textbf{if } b \textbf{ then } a \textbf{ else } a') \;= \\
\quad \mathsf{assume}(b); \mathsf{consume}(a) \wedge \mathsf{assume}(\neg b); \mathsf{consume}(a')
\end{array}$$

Notice that consuming a predicate assertion generates one transition relation for each choice function ψ that picks one match for each initial state that has matches.

Verifying a routine call means consuming the precondition (under the symbolic store obtained by binding the arguments), followed by picking a fresh symbol to represent the return value, followed by producing the postcondition, followed by binding the return value into the caller's symbolic store. The other

commands are straightforward.

$$
\begin{aligned}
&\mathsf{verify}(x := r(\overline{e})) \;=\\
&\quad \bigwedge s.\ \{\{((\Sigma, h, s), (\Sigma, h, [\overline{x} := s(\overline{e})]))\}\}; \mathsf{consume}(a);\\
&\qquad \bigwedge r.\{\{((\Sigma, h, s'), (\Sigma', h, s'[\mathsf{result} := r])) \mid (r, \Sigma') = \mathsf{nextFresh}(\Sigma)\}\};\\
&\qquad\quad \mathsf{produce}(a'); \{\{((\Sigma, h, s''), (\Sigma, h, s[x := r]))\}\}\\
&\quad \text{where } \mathbf{routine}\ r(\overline{x})\ \mathbf{req}\ a\ \mathbf{ens}\ a'\\
&\mathsf{verify}((c; c')) = \mathsf{verify}(c); \mathsf{verify}(c')\\
&\mathsf{verify}(\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \mathsf{assume}(b); \mathsf{verify}(c) \wedge \mathsf{assume}(\neg b); \mathsf{verify}(c')
\end{aligned}
$$

Verifying a routine means binding the parameters to fresh symbols, then producing the precondition, then saving the resulting symbolic store s', then verifying the body under the original symbolic store, then restoring the symbolic store s' and binding the result value, and then finally consuming the postcondition. The routine is valid if in at least one transition relation, the initial state does not lead to **abort**.

$$
\begin{aligned}
&\mathsf{valid}(\mathbf{routine}\ r(\overline{x})\ \mathbf{req}\ a\ \mathbf{ens}\ a'\ \mathbf{do} c) =\\
&\quad \exists R \in W.\ ((\Sigma_0, \emptyset, [\overline{x} := \overline{u}]), \mathbf{abort}) \notin R\\
&\quad \text{where } (\overline{u}, \Sigma_0) = \mathsf{nextFresh}(\emptyset)\\
&\quad \text{and } W = \bigwedge s.\ \{\{((\Sigma, h, s), (\Sigma, h, s))\}\}; \mathsf{produce}(a);\\
&\qquad \bigwedge s'.\ \{\{((\Sigma, h, s'), (\Sigma, h, s))\}\}; \mathsf{verify}(c);\\
&\qquad\quad \bigwedge s''.\ \{\{((\Sigma, h, s''), (\Sigma, h, s'[\mathsf{result} := s''(\mathsf{result})]))\}\}; \mathsf{consume}(a')
\end{aligned}
$$

A program is valid if all routines are valid.

2.4 Soundness Proof Sketch

We now sketch an approach for proving the soundness of this algorithm. First, we define *abstracted execution* operations aproduce, aconsume, and averify, that differ from the corresponding symbolic execution operations only in that they use concrete values instead of logical terms in heap chunks and store bindings. We then prove that the relation between an abstracted state and a symbolic state that represents it (through some interpretation of the logical symbols) is a simulation relation: if some symbolic state represents some abstracted state, then for every transition relation in the symbolic execution, there is a transition relation in the abstracted execution such that if the abstracted state aborts, then the symbolic state aborts, and if the abstracted state leads to some other abstracted state, then the symbolic state either aborts or leads to some other symbolic state that represents this abstracted state. It follows that if a program is valid under symbolic execution, it is valid under abstracted execution.

We then prove two lemmas about abstracted execution. Firstly, we prove that all abstracted execution operations are *local*, in the sense that heap contraction is a simulation relation: for state $(s, h \uplus h_0)$ and contracted state (s, h), for every transition relation R_2 there is a transition relation R_1 such that if $(s, h \uplus h_0)$ aborts in R_1, then (s, h) aborts in R_2, and otherwise if $(s, h \uplus h_0)$ leads to a state

(s', h') in R_1, then either (s, h) aborts in R_2 or $h_0 \subseteq h'$ and (s, h) leads to state $(s', h' - h_0)$ in R_2.

Secondly, we prove the soundness of abstracted assertion production and consumption. Specifically, we prove that if we consume an assertion a in a state (h, s), then this either aborts or we obtain some state (h', s'), and for every such final state it holds that producing a in some state (h'', s) leads to state $(h'' \uplus (h - h'), s')$.

Finally, given a big-step operational semantics of the programming language, we prove that if all routines are valid, then concrete execution is simulated by abstracted execution: for every initial state, if concrete execution leads to some outcome, then in each transition relation either abstracted execution aborts or leads to the same or a contracted outcome.[1] We detail the case of routine call.

Consider a routine call $x := r(\overline{e})$ started in a state (s, h). Now, consider an arbitrary transition relation of consumption of r's precondition in state $([\overline{x} := s(\overline{e})], h)$. Either this aborts, in which case abstracted execution of the routine call aborts and we are done. Otherwise, it leads to a state (s', h'). Then, by the second lemma, production of the precondition in state $([\overline{x} := s(\overline{e})], h')$ leads to state (s', h). Now, consider the execution of the body of r in state $([\overline{x} := s(\overline{e})], h)$. If this aborts, then by the induction hypothesis, we have that abstracted execution of the body aborts in all transition relations when started in the same state. By locality, it follows that production of the precondition in state $([\overline{x} := s(\overline{e})], \emptyset)$ leads to state $(s', h - h')$ and abstracted execution of the body in state $(s', h - h')$ aborts. This contradicts the assumption that the routine is valid.

Now consider the case where execution of the body of the routine, when started in state $([\overline{x} := s(\overline{e})], h)$, leads to some state (s'', h''). Consider an arbitrary transition relation of consumption of r's postcondition in state $(s'[\mathsf{result} := s''(\mathsf{result})], h'')$. Either consumption aborts, in which case, by locality, the routine is invalid and we obtain a contradiction. Or it leads to some state (s''', h'''). Then, by the second lemma, production of r's postcondition in state $(s'[\mathsf{result} := s''(\mathsf{result})], h')$ leads to state $(s''', h' \uplus (h'' - h'''))$. By locality, we have $h' \subseteq h'''$; as a result, we have $h' \uplus (h'' - h''') \subseteq h''' \uplus (h'' - h''') = h''$, so the abstracted execution leads to a contraction of the final concrete execution state $(s[x := s''(\mathsf{result})], h'')$.

3 Permission Accounting

This section presents VeriFast's support for permission accounting. Specifically, to enable convenient sharing of heap locations, mutexes, and other resources among multiple threads, and for other purposes, VeriFast has built-in support for fractional permissions (Section 3.1), and library support for counting permissions (Section 3.4). To facilitate the application of fractional permissions to

[1] A contracted outcome (i.e., with a smaller heap) occurs in the case of routine calls if heap chunks remain after the routine's postcondition is consumed. When verifying a C program, VeriFast signals a leak error in this case; for a Java program, however, this is allowed.

user-defined predicates, VeriFast has special support for *precise predicates* (Section 3.2). Finally, to facilitate unrestricted sharing of resources in case reassembly is not required, VeriFast supports *dummy fractions* (Section 3.3).

3.1 Fractional Permissions

VeriFast has fairly convenient built-in support for the fractional permissions system proposed by Bornat et al. [3]. The basics of this support consist of the following elements: a coefficient term in each heap chunk, a relaxed proof rule for read-only memory accesses, fractional assertions, opening and closing of fractional user-defined predicate chunks, and autosplitting. Some more advanced features are explained in later subsections.

Fractional Heap Chunks and Memory Reads As mentioned in Section 2, in VeriFast's symbolic heap data structure, each heap chunk specifies a term known as its *coefficient*. This term belongs to the SMT solver's sort of real numbers. If the real number represented by this term is different from 1, we say the chunk is a *fraction*. On any feasible symbolic execution path, the coefficient of any chunk that represents a memory location lies between 0, exclusive, and 1, inclusive, where 1 represents exclusive write access, and a smaller value represents shared read access. However, coefficients of user-defined predicates may feasibly lie outside this range.

Fractional Assertions Both points-to assertions and predicate assertions may mention a coefficient f, which is a pattern of type `real`, using the syntax `[`f`]`ℓ `|->` v or `[`f`]`$p(\overline{v})$. Just like other patterns, the coefficient pattern may be an expression, such as `1/2` or `x`, where `x` is a previously declared variable of type `real`. It may also be a question mark pattern `?`x, which existentially quantifies over the coefficient and binds it to x. Finally, it may be a dummy pattern `_`, which also existentially quantifies over the coefficient but does not bind it to any variable. Dummy coefficient patterns are treated specially; see Section 3.3. If a points-to assertion or predicate assertion does not mention a coefficient, it defaults to 1.

The following simple example illustrates a common pattern:

```
int read_cell(int *cell)
    requires [?f]integer(cell, ?v);
    ensures [f]integer(cell, v) &*& result == v;
{ return *cell; }
```

The above function requires an arbitrary fraction of the `integer` chunk that permits access to the `int` object at location `cell`, and returns the same fraction.

Fractions and User-Defined Predicates The syntax of `open` and `close` ghost statements allows mentioning a coefficient: `open [`f`]`$p(\overline{v})$, `close [`f`]`$p(\overline{v})$.

By definition, applying a coefficient f to a user-defined predicate is equivalent to multiplying the coefficient of each chunk mentioned in the predicate's body by f. There is no restriction on the value of f. If no coefficient is mentioned, a `close` operation defaults to coefficient 1, and an `open` operation defaults to the coefficient found in the symbolic heap.

Autosplitting When consuming a predicate assertion, the nano-VeriFast algorithm presented in Section 2 requires a precise match between the coefficient expression specified in the predicate assertion and the coefficient term in a heap chunk. Full VeriFast is more relaxed: for assertion coefficient f_a and chunk coefficient f_c, it requires either $f_\mathrm{a} = f_\mathrm{c}$ or $0 < f_\mathrm{a} < f_\mathrm{c}$. In the latter case, consumption does not remove the chunk, but simply reduces the chunk's coefficient to $f_\mathrm{c} - f_\mathrm{a}$.

3.2 Precise Predicates

Autosplitting is sound both for built-in memory location predicates and for arbitrary user-defined predicates. For built-in memory location predicates, we also have a merge law:

$$[f_1]\ell \mapsto v_1 * [f_2]\ell \mapsto v_2 \Rightarrow [f_1 + f_2]\ell \mapsto v_1 \wedge v_2 = v_1$$

This law states not only that two fractions whose first arguments are equal can be merged into one, but also that their second arguments are equal. VeriFast automatically performs this merge operation and adds this equality to the path condition when producing a built-in predicate chunk if a matching chunk is already present in the symbolic heap. Merging of fractional permissions is important because it enables modifying or deallocating memory locations once they are no longer shared between multiple threads.

However, VeriFast does not automatically merge arbitrary predicate chunks, even if they have identical argument lists. Doing so would be unsound, as illustrated by the following pathological user-defined predicate:

```
predicate foo() = integer(_, _);
lemma void evil()
    requires integer(_, _) &*& integer(_, _);
    ensures [2]integer(_, _);
{ close foo(); close foo(); open [2]foo(); }
```

Specifically, this would violate the invariant that on feasible paths, memory location chunks never appear with a coefficient greater than 1.

Therefore, VeriFast automerges only *precise predicates*. A user-defined predicate may be declared as precise by using a semicolon in the parameter list to separate the *input parameters* from the *output parameters*. If a predicate is declared as precise, VeriFast performs a static analysis on the predicate body to check that the merge law holds for this predicate. The merge law for a predicate p with input parameters $\overline{x}$ and output parameters $\overline{y}$ states:

$$[f_1]p(\overline{x}, \overline{y}_1) * [f_2]p(\overline{x}, \overline{y}_2) \Rightarrow [f_1 + f_2]p(\overline{x}, \overline{y}_1) \wedge \overline{y}_2 = \overline{y}_1$$

For example, the static analysis accepts the following definition of the classic list segment predicate:

```
struct node { struct node *next, int value };
predicate lseg(struct node *f, struct node *l; list<int> vs) =
    f == l ? vs == nil :
    f->next |-> ?n &*& f->value |-> ?v &*& malloc_block_node(f) &*&
    lseg(n, l, ?vs0) &*& vs == cons(v, vs0);
```

As a result, the following lemma is verified automatically:

```
lemma void lseg_merge(struct node *f, struct node *l)
    requires [?f1]lseg(f, l, ?vs1) &*& [?f2]lseg(f, l, ?vs2);
    ensures [f1+f2]lseg(f, l, vs1) &*& vs2 == vs1;
{}
```

The static analysis for a predicate definition **predicate** $p(\overline{x}; \overline{y}) = a;$ checks that given fixed variables $\overline{x}$, assertion a is precise and fixes variables $\overline{y}$; formally: $\overline{x} \vdash a \rightsquigarrow \overline{y}$. The meaning of this judgment is given by a merge law for assertions:

$$[f_1]a_1 * [f_2]a_2[\overline{x}_1/\overline{x}_2] \Rightarrow [f_1 + f_2]a_1 \wedge \overline{y}_2 = \overline{y}_1$$

where a_1 is a with all free variables subscripted by 1 and a_2 is a with all free variables subscripted by 2. The static analysis proceeds according to the inference rules shown in Figure 1. Notice that the analysis allows both expressions and

$$\frac{\textbf{predicate } q(\overline{x}; \overline{y}) \qquad |\overline{e}| = |\overline{x}| \qquad \mathsf{FreeVars}(\overline{e}) \subseteq X}{X \vdash q(\overline{e}, \overline{pat}) \rightsquigarrow X \cup \overline{\mathsf{FixedVars}(pat)}} \qquad \frac{\mathsf{FreeVars}(e) \subseteq X}{X \vdash x = e \rightsquigarrow X \cup \{x\}}$$

$$X \vdash e \rightsquigarrow X \qquad \frac{\mathsf{FreeVars}(e) \subseteq X \quad X \vdash a \rightsquigarrow Y}{X \vdash [e]a \rightsquigarrow Y} \qquad \frac{X \vdash a \rightsquigarrow Y}{X \vdash [\_]a \rightsquigarrow Y} \qquad \frac{X \vdash a_1 \rightsquigarrow Y \quad Y \vdash a_2 \rightsquigarrow Z}{X \vdash a_1 * a_2 \rightsquigarrow Z}$$

$$\frac{\mathsf{FreeVars}(b) \subseteq X \qquad X \vdash a_1 \rightsquigarrow Y \qquad X \vdash a_2 \rightsquigarrow Y}{X \vdash b \ ?\ a_1 \ :\ a_2 \rightsquigarrow Y} \qquad \frac{X \vdash a \rightsquigarrow Y \qquad Y' \subseteq Y}{X \vdash a \rightsquigarrow Y'}$$

where

$$\mathsf{FixedVars}(x) = \{x\} \quad \mathsf{FixedVars}(e) = \emptyset \quad \mathsf{FixedVars}(?x) = \{x\} \quad \mathsf{FixedVars}(\_) = \emptyset$$

Fig. 1. The static analysis for preciseness of assertions

dummy patterns as coefficients (but not question mark patterns). In allowing dummy patterns, VeriFast's notion of preciseness deviates from the separation logic literature, where an assertion is precise if for any heap, there is at most

one subheap that satisfies the assertion. Indeed, in the presence of dummy fractions, there may be infinitely many fractional subheaps that satisfy the assertion; however, the merge law still holds.

3.3 Dummy Fractions for Leakable Resources

VeriFast treats dummy coefficients in predicate assertions specially, to facilitate scenarios where reassembly of fractions of a given resource is not required, and as a result the resource can be shared arbitrarily. Specifically, when consuming a predicate assertion with a dummy coefficient, VeriFast always performs an autosplit; that is, it does not remove the matched chunk but merely replaces its coefficient by a fresh symbol.

Furthermore, when verifying a C program, dummy fractions affect leak checking. In general, when verifying a C function, if after consuming the postcondition the symbolic heap is not empty, VeriFast signals a leak error. However, VeriFast does not signal an error if for all remaining resources, the user has indicated explicitly that leaking this resource is acceptable. The user can do so using a `leak` a`;` command. This command consumes the assertion a, and then reinserts all consumed chunks into the symbolic heap, after replacing their coefficients with fresh symbols and registering these symbols as *dummy coefficient symbols*. Leaking a chunk whose coefficient is a dummy coefficient symbol is allowed.

To allow this leakability information to be carried across function boundaries, dummy coefficients in assertions are considered to match only dummy coefficient symbols. That is, consuming a dummy fraction assertion matches only chunks whose coefficients are dummy coefficient symbols, and producing a dummy fraction assertion produces a chunk whose coefficient is a dummy coefficient symbol.

To understand the combined benefit of these features, consider the common type of program where the main method creates a mutex and then starts an unbounded number of threads, passing a fraction of the mutex to each thread. Each thread leaks its mutex fraction when it dies. If the user performs a `leak` operation on the mutex directly after it is created, VeriFast automatically splits the mutex chunk when a thread is started, and silently leaks each thread's fraction when the thread finishes.

VeriFast also uses dummy fractions to represent C's string literals.

3.4 Counting Permissions

Fractional permissions are sufficient in many sharing scenarios; however, an important example of a scenario where they are not applicable is when verifying a program that uses reference counting for resource management. For this scenario, another permission accounting scheme known as counting permissions [3] is appropriate.

VeriFast does not have built-in support for counting permissions. However, using VeriFast's support for higher-order predicates, VeriFast offers counting permissions support in the form of a trusted library, specified by header file

```
predicate counting<a, b>(predicate(a; b) p, a a, int count; b b);
predicate ticket<a, b>(predicate(a; b) p, a a, real frac;);

lemma void start_counting<a, b>(predicate(a; b) p, a a);
    requires p(a, ?b);
    ensures counting(p, a, 0, b);

lemma void counting_match_fraction<a, b>(predicate(a; b) p, a a);
    requires counting(p, a, ?count, ?b1) &*& [?f]p(a, ?b2);
    ensures counting(p, a, count, b1) &*& [f]p(a, b2) &*& b2 == b1;

lemma real create_ticket<a, b>(predicate(a; b) p, a a);
    requires counting(p, a, ?count, ?b);
    ensures counting(p, a, count + 1, b)
        &*& ticket(p, a, result) &*& [result]p(a, b) &*& 0 < result;

lemma void destroy_ticket<a, b>(predicate(a; b) p, a a);
    requires counting(p, a, ?count, ?b1)
        &*& ticket(p, a, ?f) &*& [f]p(a, ?b2) &*& 0 != count;
    ensures counting(p, a, count - 1, b1) &*& b2 == b1;

lemma void stop_counting<a, b>(predicate(a; b) p, a a);
    requires counting(p, a, 0, ?b);
    ensures p(a, b);
```

Fig. 2. The specification of VeriFast's counting permissions library

`counting.h`, reproduced in Figure 2. This library allows any precise predicate of one input parameter and one output parameter to be shared by means of counting permissions. VeriFast's built-in memory location predicates satisfy this constraint, so they can be used directly. Precise predicates that are of a different shape can be wrapped in a helper predicate that bundles the input and output arguments into tuples.

Once a chunk is wrapped into a `counting` chunk using the `start_counting` lemma, tickets can be created from it using the `create_ticket` lemma. This lemma not only increments the `counting` chunk's counter and produces a `ticket` chunk; it also produces an unspecified fraction of the wrapped chunk. In case of built-in memory location chunks, this allows the memory location to be read immediately. The `ticket` chunk remembers the coefficient of the produced fraction. The same fraction is consumed again when the ticket is destroyed using lemma `destroy_ticket`. When the counter reaches zero, the original chunk can be unwrapped using lemma `stop_counting`.

Notice that this library is sound even when applied to predicates that are not *unique*, i.e., predicates that can appear with a fraction greater than one. However, the existence of non-unique precise predicates does mean that we cannot assume

that the counter of a `counting` chunk remains nonnegative, as illustrated in Figure 3.

```
predicate foo(int *n; int v) = [1/2]integer(n, v);
predicate hide(int *n; int v) = counting(foo, n, 1, v);
lemma void test(int *n)
    requires integer(n, ?v);
    ensures counting(foo, n, -1, v) &*& hide(n, v);
{
    close [2]foo(n, v);
    start_counting(foo, n); create_ticket(foo, n);
    close hide(n, v);
    start_counting(foo, n); destroy_ticket(foo, n);
}
```

Fig. 3. Example where a counter decreases below zero

4 Lemma Function Termination and Dynamic Binding

VeriFast supports *lemma functions*, which are like ordinary C functions, except that lemma functions and calls of lemma functions are written inside annotations, and VeriFast checks that they have no side-effects on non-ghost memory and that they terminate. Lemma functions serve mainly to encode inductive proofs of lemmas about inductive datatypes, such as the associativity of appending two mathematical lists, or inductive proofs of lemmas about recursive predicates, such as a lemma stating that a linked list segment from node n_1 to node n_2 separately conjoined with a linked list segment from node n_2 to 0 implies a linked list segment from node n_1 to 0.

To enable such inductive proofs, lemma functions are allowed to be recursive. Specifically, to ensure termination, VeriFast allows a statically bound lemma function call if either the callee is defined before the caller in the program text, or the callee equals the caller and one of the following hold: 1) after consuming the precondition, at least one full (i.e., non-fractional) memory location predicate remains, or 2) the body of the lemma function is a switch statement over one of the function's parameters whose type is an inductive datatype, and the callee's argument for this parameter is a component of the caller's argument for this parameter, or 3) the body of the lemma function is not a switch statement and the first chunk consumed by the callee's precondition was obtained from the first chunk produced by the caller's precondition through one or more `open` operations. These three cases constitute induction on the size of the concrete heap, induction on the size of an argument, and induction on the derivation of the first conjunct of the precondition.

However, VeriFast supports not just statically bound lemma function calls, but dynamically bound calls as well. Specifically, VeriFast supports *lemma function pointers* and *lemma function pointer calls*. The purpose of these is as follows.

VeriFast supports the modular specification and verification of fine-grained concurrent data structures. It does so by modularizing Owicki and Gries's approach based on auxiliary variables. The problem with their approach is that it requires application-specific auxiliary variable updates to be inserted inside critical sections. If the critical sections are inside a library that is to be reused by many applications, this is a problem. In earlier work [4], we propose to solve this problem by allowing applications to pass auxiliary variable updates into the library in a simple form of higher-order programming. In VeriFast, this can be realized through lemma function pointers.

A simple approach to ensure termination of lemma functions in the presence of lemma function pointers would be to allow lemma function pointer calls only in non-lemma functions. However, when building fine-grained concurrent data structures on top of other fine-grained concurrent data structures, layer N needs to be able to call lemma function pointers it receives from layer $N + 1$ inside of its own lemma function, which it passes to layer $N - 1$.

To support this, we introduced a new kind of built-in heap chunks, called *lemma function pointer chunks*. A call of a lemma function pointer p is allowed only if the symbolic heap contains a lemma function pointer chunk for p, and this chunk becomes unavailable for the duration of the call. Non-lemma functions may produce lemma function pointer chunks arbitrarily. A lemma function may only produce lemma function pointer chunks for lemma functions that appear before itself in the program text, and furthermore, these chunks are consumed again before the producing lemma function terminates, so the pointer calls must occur within the dynamic scope of the producing lemma function.

We prove that this approach guarantees lemma function termination, by contradiction. Consider an infinite chain of nested lemma function calls. Since we have termination of statically bound calls, the chain must contain infinitely many pointer calls. Of all functions that appear infinitely often, consider the one that appears latest in the program text. It must be called infinitely often through a pointer. Therefore, infinitely many pointer chunks must be generated during the chain. However, these can only be generated by functions that appear later, which is a contradiction.

5 Ongoing Efforts

In this section, we briefly describe three projects currently proceeding in our group.

5.1 JavaCard Programs

JavaCard is a trimmed-down version of the Java Platform for smart cards such as cell phone subscriber cards, payment cards, identity cards, etc. We are ap-

plying VeriFast to a number of JavaCard programs, to prove absence of runtime exceptions and functional correctness properties.

An interesting aspect of the JavaCard execution environment is the fact that by default, objects allocated by a JavaCard program (called an *applet*) are *persistent*. That is, once a JavaCard applet is installed on a card, the applet object and objects reachable from its fields persist for the entire lifetime of the smart card. This interacts in interesting ways with the phenomenon of *card tearing*, which occurs when the user removes the smart card from the card reader while a method call on the applet object is in progress. To allow the programmer to preserve the consistency of the applet object, JavaCard offers a transaction mechanism, that ensures that modifications to objects during a transaction are rolled back if a card tear occurs before the transaction is committed.

We developed a specification of the JavaCard API that is sound in the presence of card tearing. We did not need to modify the VeriFast tool itself. In our specification, when a newly installed applet is registered with the virtual machine, the virtual machine takes ownership of the applet's state as defined by its `valid` predicate. When an applet receives a method call, the method receives a `1/2` fraction of the applet's `valid` predicate. This allows the method to inspect but not modify the applet's state. As a result, the method is forced to call the `beginTransaction` method before modifying the state. This API method produces the other half of the `valid` chunk. Conversely, API method `commitTransaction` consumes the entire `valid` chunk and produces a `1/2` fraction.

The soundness argument for this approach goes as follows. We need to show that in every execution, even one where card tears occur, at the start of each toplevel method call on the applet, the `valid` predicate is fully owned by the VM. We do so by showing that at every point during a method call, either we are in a transaction, or the VM owns half of `valid` and the method call owns the other half. When a method call terminates, either normally or due to a card tear, the method call's fraction is simply transfered to the VM. This proof explains the contract of `commitTransaction`: if `commitTransaction` merely consumed `1/2` of `valid`, it would not guarantee that the thread owned the other half.

5.2 Integrating Shape Analysis

We are in the process of integrating separation logic-based shape analysis algorithms from the literature [5, 6] into VeriFast. The goal is to enable a scenario where annotations are inserted into a program using a mixed manual-automatic process: the user writes some manual annotations; then they invoke the shape analysis algorithm, which, given the existing annotations, infers additional ones; then, the user adds further annotations where the algorithm failed; etc. We are not yet at the point where we can report if this approach works or not.

We are currently targeting the scenario where the code is not evolving, i.e., the scenario where an existing, unannotated program is annotated for the first time. In a later stage, we intend to consider the question whether shape analysis can help to adapt existing annotations to code evolution. The latter problem

seems much more difficult, especially if the generated shape annotations have been extended manually with functional information.

5.3 Linux Device Drivers

We are applying VeriFast to the verification of device drivers for the Linux operating system. These programs seem particularly suited for formal verification, because they are at the same time tricky to write, critical to the safety of the system, written by many different people with varying backgrounds and priorities, and yet relatively small and written against a relatively small API.

A significant part of the effort consists in writing specifications for the Linux kernel facilities used by the driver being verified. Part of the challenge here is that these facilities are often documented poorly or not at all, so we often find ourselves inventing a specification based on inspection of the kernel source code. Another part of the challenge is that VeriFast does not yet support all C language features required to interface with these facilities. As a temporary measure, in these cases we write a thin intermediate library that implements a VeriFast-friendly interface on top of the actual kernel interface.

We are only in the early stages of this endeavor. We have so far verified a small "Hello, world" driver that exposes a simple `/proc` file with an incrementing counter. This example, and the preliminary version of the VeriFast Linux Kernel Module Verification Kit that enables its verification, are included in the current VeriFast distribution. We are currently looking at a small USB keyboard driver.

Acknowledgements

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U. Leuven.

References

1. Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, 2010.
2. Bart Jacobs, Jan Smans, and Frank Piessens. The VeriFast program verifier: A tutorial. At http://www.cs.kuleuven.be/~bartj/verifast/, 2010.
3. Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, 2005.
4. Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, 2011.
5. Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
6. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.