Shared Boxes: Rely-Guarantee Reasoning in VeriFast

Jan Smans *Dries Vanoverberghe*
Dominique Devriese *Bart Jacobs*
Frank Piessens

Report CW 662, May 2014

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Shared Boxes: Rely-Guarantee Reasoning in VeriFast

Jan Smans *Dries Vanoverberghe*
Dominique Devriese *Bart Jacobs*
Frank Piessens

Report CW 662, May 2014

Department of Computer Science, KU Leuven

Abstract

VeriFast is a verifier for single-threaded and multithreaded C and Java programs. It takes a C or Java program annotated with preconditions and postconditions in a separation logic notation, and verifies statically that these preconditions and postconditions hold, using symbolic execution. In plain separation logic, a thread either has full ownership of a memory location and knows the value at the location, or it has no ownership and no knowledge of the value of the location. Existing work proposes a marriage of rely-guarantee reasoning and separation logic to address this. In this document, we describe the shared boxes mechanism, which marries separation logic and rely-guarantee reasoning in VeriFast.

We introduce and motivate the shared boxes mechanism using a minimalistic example and a realistic example. The minimalistic example is a *counter* program where one thread continuously increments a counter and other threads check that the counter does not decrease. For the realistic example, we verify functional correctness of the Michael-Scott queue, a lock-free concurrent data structure. We define the syntax and semantics of a simple C-like programming language, and we define a separation logic with shared boxes and prove its soundness. We discuss the implementation in VeriFast and the examples we verified using our VeriFast implementation.

Shared Boxes: Rely-Guarantee Reasoning in VeriFast

Jan Smans Dries Vanoverberghe Dominique Devriese
Bart Jacobs Frank Piessens
iMinds-DistriNet, Dept. Comp. Sci., KU Leuven, Belgium
firstname.lastname@cs.kuleuven.be

Abstract

VeriFast is a verifier for single-threaded and multithreaded C and Java programs. It takes a C or Java program annotated with preconditions and postconditions in a separation logic notation, and verifies statically that these preconditions and postconditions hold, using symbolic execution. In plain separation logic, a thread either has full ownership of a memory location and knows the value at the location, or it has no ownership and no knowledge of the value of the location. Existing work proposes a marriage of rely-guarantee reasoning and separation logic to address this. In this document, we describe the shared boxes mechanism, which marries separation logic and rely-guarantee reasoning in VeriFast.

We introduce and motivate the shared boxes mechanism using a minimalistic example and a realistic example. The minimalistic example is a *counter* program where one thread continuously increments a counter and other threads check that the counter does not decrease. For the realistic example, we verify functional correctness of the Michael-Scott queue, a lock-free concurrent data structure. We define the syntax and semantics of a simple C-like programming language, and we define a separation logic with shared boxes and prove its soundness. We discuss the implementation in VeriFast and the examples we verified using our VeriFast implementation.

1 A Minimalistic Example

Consider the following program:

$$\begin{array}{l}\mathsf{c} := \mathbf{cons}(0);\\ \mathbf{fork}\ (\mathbf{while\ true\ do}\ \langle \mathsf{n} := [\mathsf{c}]; [\mathsf{c}] := \mathsf{n}+1\rangle);\\ \mathbf{while\ true\ do\ fork}\ (\langle \mathsf{m} := [\mathsf{c}]\rangle; \langle \mathsf{m}' := [\mathsf{c}]\rangle; \mathbf{assert}\ \mathsf{m} \leq \mathsf{m}')\end{array}$$

We use the usual notation for heap-manipulating programs from the separation logic literature. The command $x := \mathbf{cons}(\overline{e})$ allocates a sequence of consecutive heap locations and initializes them with the values of the expressions $\overline{e}$; the example allocates a single cell (which we will refer to as the *counter cell*), and initializes it to zero. We use the following notation for concurrent programming: command **fork** c executes command c in a new thread; $\langle c \rangle$ denotes atomic execution of command c.

After allocating the counter cell and storing its address in variable `c`, the program forks one thread that repeatedly atomically increments the cell, and an unbounded number of threads that inspect the cell twice and assert that its value increases monotonically.

We wish to prove that the **assert** command never fails. Notice that this program cannot be verified in plain separation logic, since no single thread can be the exclusive owner of the counter cell. Concurrent Separation Logic (CSL), which extends separation logic with support for critical sections accessing shared resources with resource invariants, also does not support this example directly, since a resource invariant could only state that the cell's value is nonnegative and could not describe the *evolution* of the counter cell's value.

A combination of CSL and *ghost cells* with *fractional permissions* [1, 6] could verify this example, using a dynamic form of the resource invariant-based Owicki-Gries method [8, 6] for reasoning about concurrent programs: the shared resource containing the counter cell could be extended with a *ghost linked list* built from ghost cells. Each inspector thread, during its first inspection of the counter cell value, would add a ghost node to the end of this list, containing as its value the observed counter cell value. The thread would retain in its local state a fraction of the linked list, from the head up to the thread's node. The resource invariant would state that each node value is a lower bound for the current counter cell value. Upon the second inspection, the thread could match up the linked list in the resource with its local knowledge and conclude that the new counter cell value must be at least the previously observed value.

Such building of *ghost objects* probably yields a complete proof system. Still, in the present document, we present an alternative approach, which allows the proof author to express his insights more conveniently and more directly: shared boxes.

A shared box can be thought of at a high level as a shared resource from CSL equipped with a *two-state invariant* (or, equivalently, a *rely condition*) instead of a regular single-state invariant. This enables the proof author to express directly any desired constraints on the evolution of the shared resource. Furthermore, we allow assertions in thread proof outlines to include *shared box assertions*, assertions about the state of the shared resource that are checked to be *stable* with respect to the shared box's rely condition.

To verify the example, we put the counter cell in a shared box whose rely condition states that the cell's value may not decrease. Each inspector thread's proof outline, between the two inspections, includes a shared box assertion stating that the counter cell's value is bounded below by the first observed value.

This general idea is very similar to what has been proposed before (e.g. [9, 4, 3, 2]). However, in order to integrate this mechanism conveniently into our VeriFast verification tool, we have made a number of design decisions:

- Shared boxes can be created dynamically, but each shared box must be an instance of a statically declared *box class*.
- A box class has a *name* and a *parameter list*.
- A box class's rely condition is specified in the form of the combination of a *box invariant* and a set of *action specifications*.
- A box invariant is a VeriFast separation logic assertion that may use the

box class parameters and may bind additional logical variables, together with the box class parameters called the *box state variables*.

- An action specification consists of an action name, a parameter list, a precondition, and a postcondition. The precondition is a boolean (i.e. pure, non-spatial) expression over the action parameters and the box state variables. The postcondition is a boolean expression over the action parameters and two versions of the box state variables: the old versions and the new versions.

- Whenever a thread mutates the resources held by a box, it must specify an action name and action arguments, and VeriFast checks that the operation complies with the action precondition and postcondition.

- Shared box assertions are expressed as *box handle predicate assertions*, referring to one of a set of *box handle predicates* (or *handle predicates* for short) declared as part of the box class. A handle predicate declaration consists of a name, a parameter list, and a *handle predicate invariant*, which is a boolean expression over the handle parameters and the box state variables.

- Each handle predicate declaration must include a *preserved-by clause* for each box class action, which may state any ghost commands (such as lemma invocations) required to establish that the handle predicate invariant is preserved by the action.

We formalize the syntax of our proof system. Let $B \in \mathcal{B}$ range over box class names, $A \in \mathcal{A}$ over action names, $P \in \mathcal{P}$ over handle predicate names, $x \in \mathcal{X}$ over program variable names, and $X \in \mathcal{L}$ over logical variable names. The syntax of box classes, assertions a, and program commands c is as follows:

$$
\begin{array}{rl}
\mathit{boxClass} ::= & \textbf{boxclass}\ B(\overline{X})\ \{\ \textbf{inv}\ \exists \overline{X}.\ a\ \overline{\mathit{actionSpec}}\ \overline{\mathit{handlePred}}\ \} \\
\mathit{actionSpec} ::= & \textbf{action}\ A(\overline{X})\ \textbf{req}\ b\ \textbf{ens}\ b \\
\mathit{handlePred} ::= & \textbf{handlePred}\ P(\overline{X})\ \{\ \textbf{inv}\ b\ \} \\
e ::= & z \mid x \mid X \mid e + e \mid e - e \\
b ::= & e = e \mid e < e \mid b \wedge b \mid \neg b \\
f ::= & e/e \\
a ::= & b \mid a \vee a \mid \exists X.\ a \mid e \overset{f}{\mapsto} e \mid a * a \mid [f]B(e, \overline{e}) \mid P(e, e, \overline{e}) \\
c ::= & x := e \mid c; c \mid \textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c \mid \textbf{while}\ b\ \textbf{do}\ c \\
& \mid x := \textbf{cons}(\overline{e}) \mid x := [e] \mid [e] := e \mid \langle c \rangle \mid \textbf{fork}\ c
\end{array}
$$

We assume that logical variables X do not appear inside program commands c. Notice that we do not formalize preserved-by clauses; we will formalize the stability constraints but leave the mechanism for proving them unspecified. Our proof system supports fractional permissions f on points-to assertions $e \overset{f}{\mapsto} e$ and *box assertions* $[f]B(e, \overline{e})$. The first argument of a box assertion is the *box identifier*; the first argument of a *handle predicate assertion* $P(e, e, \overline{e})$ is the *handle identifier*, and the second argument is the box identifier.

The box class declaration and proof outline for the example program are shown in Figure 1. Notice that in the formal system, in an action postcondition, we use unprimed versions of the box invariant variables to denote the old values, and primed versions to denote the new values.

```
boxclass incrbox(c) {
  inv ∃v. c ↦ v
  action incr() req true ens v ≤ v′
  handlePred observed(val) { inv val ≤ v }
}

{true}
c := cons(0);
{c ↦ 0}
{[1]incrbox(_, c)}   CreateBox
fork (
  {[_]incrbox(_, c)}
  while true do
    ⟨n := [c]; [c] := n + 1⟩   Action incr()
);
{[_]incrbox(_, c)}
while true do (
  ⟨m := c⟩;
  {∃b. [_]incrbox(b, c) ∗ observed(_, b, m)}
  ⟨m′ := c⟩;
  {[_]incrbox(_, c) ∗ m ≤ m′}
  assert m ≤ m′
)
```

Figure 1: Proof of the minimalistic example program

$$\textsc{StablePred}\ \frac{\begin{array}{l}\textbf{boxclass}\ B(\overline{X})\ \{\\ \quad \textbf{inv}\ \exists\overline{Y}.\ I\\ \quad \cdots\\ \quad \textbf{action}\ A(\overline{Z})\ \textbf{req}\ b\ \textbf{ens}\ b'\\ \quad \cdots\\ \quad \textbf{handlePred}\ P(\overline{U})\ \{\ \textbf{inv}\ b''\ \}\\ \quad \cdots\\ \}\end{array} \qquad \begin{array}{l}\forall \overline{v}, \overline{w}, \overline{w'}, \overline{z}, \overline{u}, r, r'.\\ \quad (r \vDash (I \wedge b'')[\overline{vwu}/\overline{XYU}]) \wedge\\ \quad (r' \vDash I[\overline{v}\,\overline{w'}/\overline{XY}]) \wedge\\ \quad (b \wedge b')[\overline{vwz}\,\overline{w'}/\overline{XYZY'}]\\ \quad \Rightarrow\ b''[\overline{v}\,\overline{w'}\,\overline{u}/\overline{XYU}]\end{array}}{A \vdash \mathsf{stable}\ P}$$

$$\textsc{Stable}\ \frac{\textbf{boxclass}\ B(\overline{X})\ \{\ \cdots\ \overline{\textbf{action}\ A_{1..n}\ \cdots}\ \overline{\textbf{handlePred}\ P_{1..m}\ \cdots}\ \}\qquad \forall i, j.\ A_i \vdash \mathsf{stable}\ P_j}{\mathsf{stable}\ B}$$

$$\textsc{CreateBox}\ \frac{\textbf{boxclass}\ B(\overline{X})\ \{\ \textbf{inv}\ \exists\overline{Y}.\ I\ \cdots\ \}\qquad \{[1]B(\_, \overline{v}) * R\}\ c\ \{Q\}}{\{I[\overline{vw}/\overline{XY}] * R\}\ c\ \{Q\}}$$

$$\textsc{Action}\ \frac{\begin{array}{l}\textbf{boxclass}\ B(\overline{X})\ \{\\ \quad \textbf{inv}\ \exists\overline{Y}.\ I\\ \quad \cdots\\ \quad \textbf{action}\ A(\overline{Z})\ \textbf{req}\ b\ \textbf{ens}\ b'\\ \quad \cdots\\ \quad \textbf{handlePred}\ P_i(\overline{U}_i)\ \{\ \textbf{inv}\ b_i\ \}\\ \}\end{array}\qquad \begin{array}{l}\{I[\overline{vw}/\overline{XY}] * \Pi_i b_i[\overline{vwu}_i/\overline{XYU}_i] * R\}\\ \forall \overline{w}.\ \ c\\ \quad \{\exists\overline{w'}.\ I[\overline{v}\,\overline{w'}/\overline{XY}] * \Pi_j(b''_j \vee b'_j[\overline{v}\,\overline{w'}\,\overline{u}'_j/\overline{XYU'}_j]) * (b \wedge b')[\overline{vw}\,\overline{w'}\overline{z}/\overline{XYY'Z}] * R'\}\end{array}}{\{[\pi]B(\beta, \overline{v}) * \Pi_i P_i(\_, \beta, \overline{u}_i) * R\}\ \langle c\rangle\ \{[\pi]B(\beta, \overline{v}) * \Pi_j(b''_j \vee P'_j(\_, \beta, \overline{u}'_j)) * R'\}}$$

Figure 2: Proof rules

Our proof system extends separation logic with extra rules for box class stability checking, box creation, and shared box mutation. The extra rules are shown in Figure 2. Note: there are restrictions on nested applications of the ACTION rule. For now, we assume that no such nested applications occur.

We assume that each declared box class B is stable: stable B. This means that each of the class' handle predicates is stable with respect to each of its actions. Stability of a handle predicate with respect to an action means that given arbitrary values $\overline{v}$ of the box class parameters, old values $\overline{w}$ and new values $\overline{w'}$ of the box invariant variables, values $\overline{z}$ of the action parameters, and values $\overline{u}$ of the handle predicate parameters, and arbitrary old and new resource bundles r and r' representing the old and new contents of the shared box, if the box invariant holds in the pre- and post-state, the action pre- and postcondition hold, and the handle predicate invariant holds in the pre-state, then the handle predicate invariant holds in the post-state.

Rule CREATEBOX allows a shared box instance to be created at any time provided its invariant holds for some part of the current locally held resources. Those resources are then consumed and a *box chunk* (a resource representing the existence of a shared box instance) is produced.

Rule ACTION allows the verification of an atomic command $\langle c \rangle$ that accesses the resources held by a shared box instance of class B, with identifier β, and with arguments $\overline{v}$. This requires that the thread own a fraction π of the box chunk. During verification of command c, the box invariant becomes available. It must be re-established before the atomic command is exited. Furthermore, local resources R may be passed into the atomic command and resources R' may be extracted and retained locally. Also, handle predicates may be consumed and produced. Any number of handle predicate chunks P_i for the box instance may be consumed on entry; their invariants b_i are assumed to hold in the pre-state. Any other number of handle predicates P'_j may be produced on exit, provided their invariants b'_j are established in the post-state. They may be produced conditionally under conditions $\neg b''_j$. It is checked that there is some action A and argument list $\overline{z}$ such that the action's precondition and postcondition are satisfied by the command.

In rule ACTION as presented in Figure 2, handle identifiers are ignored. Handle identifiers are important in advanced scenarios which will be discussed in a later section.

We show the example proof in the form of an annotated C program, as accepted and successfully verified by VeriFast, in Figure 3. This example is included in the VeriFast distribution in file `examples/shared_boxes/incrbox.c`.

2 Soundness Proof

In this section, we formalize the soundness property targeted by our proof system and then we prove it.

2.1 Operational Semantics

We first formalize a small-step operational semantics for our programming language.

```
#include <threading.h>
#include "atomics.h"
/*@ box_class incr_box(int *x) {
    invariant *x |-> ?value;
    action increase();
        requires true;
        ensures old_value <= value;
    handle_predicate observed(int v) {
        invariant v <= value;
        preserved_by increase() {}
    }
} @*/
//@ predicate_family_instance thread_run_data(inc)(int* x) = [_]incr_box(_, x);
void inc(int *x) /*@ : thread_run @*/
    /*@ requires thread_run_data(inc)(x); @*/ /*@ ensures true; @*/ {
    //@ open thread_run_data(inc)(x);
    while(true) /*@ invariant [_]incr_box(_, x); @*/ {
        ;
        /*@
        consuming_box_predicate incr_box(_, x)
        perform_action increase()
        { @*/ atomic_increment(x); /*@ };
        @*/
    }
}
void reader(int *x) /*@ requires [_]incr_box(_, x); @*/ /*@ ensures false; @*/ {
    for (;;) /*@ invariant [_]incr_box(_, x); @*/ {
        ;
        /*@
        consuming_box_predicate incr_box(_, x)
        perform_action increase()
        { @*/ int m0 = atomic_load_int(x); /*@ }
        producing_fresh_handle_predicate observed(m0);
        @*/
        /*@
        consuming_box_predicate incr_box(_, x)
        consuming_handle_predicate observed(_, m0)
        perform_action increase()
        { @*/ int m1 = atomic_load_int(x); /*@ };
        @*/
        assert(m0 <= m1);
    }
}
int main() /*@ requires true; @*/ /*@ ensures true; @*/ {
    int x;
    //@ create_box id = incr_box(&x);
    //@ leak incr_box(id, &x);
    //@ close thread_run_data(inc)(&x);
    thread_start(inc, &x);
    reader(&x);
}
```

Figure 3: The minimalistic example as an annotated C program accepted by VeriFast.

The set of machine configurations $\gamma \in \mathit{Configs}$ is defined as follows:

$$\begin{aligned} s \in \mathit{Stores} &= \mathcal{X} \to \mathbb{Z} \\ R \in \mathit{Heaps} &= \mathbb{Z} \rightharpoonup \mathbb{Z} \\ \kappa \in \mathit{Continuations} &::= \mathbf{done} \mid c; \kappa \\ \theta \in \mathit{ThreadConfigs} &= \mathit{Stores} \times \mathit{Continuations} \\ \gamma \in \mathit{Configs} &= \mathit{Heaps} \times (\mathit{ThreadConfigs} \to \mathbb{N}) \end{aligned}$$

A configuration consists of a heap and a multiset[1] of thread configurations. A thread configuration θ consists of a store and a continuation. A continuation is either **done**, indicating that the thread has finished, or a command followed by another continuation.

We use the notation $\{\!\{a, b, c\}\!\}$ to represent a multiset: $\{\!\{a_1, \ldots, a_n\}\!\} = \mathbf{0} + \{\!\{a_1\}\!\} + \cdots + \{\!\{a_n\}\!\}$ where $\mathbf{0} = \lambda x.\, 0$ represents the empty multiset and $M + \{\!\{a\}\!\} = M[a := M(a) + 1]$. We use the notations $+$ and $\uplus$ interchangeably for multiset addition.

We define a big-step relation $\Downarrow \subseteq (\mathit{Heaps} \times \mathit{Stores} \times \mathit{Commands}) \times (\mathit{Heaps} \times \mathit{Stores} \cup \{\mathbf{abort}\})$ for commands that may appear inside atomic commands:

Assign
$$(R, s, x := e) \Downarrow (R, s[x := s(e)])$$

Lookup
$$\frac{s(e) \in \operatorname{dom} R}{(R, s, x := [e]) \Downarrow (R, s[x := R(s(e))])}$$

LookupAbort
$$\frac{s(e) \notin \operatorname{dom} R}{(R, s, x := [e]) \Downarrow \mathbf{abort}}$$

Mutate
$$\frac{s(e) \in \operatorname{dom} R}{(R, s, [e] := e') \Downarrow (R[s(e) := s(e')], s)}$$

MutateAbort
$$\frac{s(e) \notin \operatorname{dom} R}{(R, s, [e] := e') \Downarrow \mathbf{abort}}$$

SeqAtomic
$$\frac{(R, s, c) \Downarrow (R', s') \qquad (R', s', c') \Downarrow o}{(R, s, c; c') \Downarrow o}$$

SeqAbort
$$\frac{(R, s, c) \Downarrow \mathbf{abort}}{(R, s, c; c') \Downarrow \mathbf{abort}}$$

IfTrueAtomic
$$\frac{s(b) \qquad (R, s, c) \Downarrow o}{(R, s, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') \Downarrow o}$$

IfFalseAtomic
$$\frac{\neg s(b) \qquad (R, s, c') \Downarrow o}{(R, s, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') \Downarrow o}$$

AtomicAtomic
$$\frac{(R, s, c) \Downarrow o}{(R, s, \langle c \rangle) \Downarrow o}$$

[1] A multiset over elements of a set A is a function $M : A \to \mathbb{N}$ where $M(a)$ is the number of occurrences of a in M.

We define a small-step relation $\rightsquigarrow \subseteq \mathit{Configs} \times (\mathit{Configs} \cup \{\mathbf{abort}\})$ as follows:

Cons
$$\frac{0 < \ell \qquad \{\ell, \ldots, \ell + n - 1\} \cap \mathrm{dom}(R) = \emptyset \qquad R' = R[\ell := s(e_1), \ldots, \ell + n - 1 := s(e_n)]}{(R, \{\!\{(s, x := \mathbf{cons}(e_1, \ldots, e_n); \kappa)\}\!\} \uplus \Theta) \rightsquigarrow (R', \{\!\{(s[x := \ell], \kappa)\}\!\} \uplus \Theta)}$$

Seq
$$(R, \{\!\{(s, (c; c'); \kappa)\}\!\} \uplus \Theta) \rightsquigarrow (R, \{\!\{(s, c; (c'; \kappa))\}\!\} \uplus \Theta)$$

IfTrue
$$\frac{s(b)}{(R, \{\!\{(s, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'; \kappa)\}\!\} \uplus \Theta) \rightsquigarrow (R, \{\!\{(s, c; \kappa)\}\!\} \uplus \Theta)}$$

IfFalse
$$\frac{\neg s(b)}{(R, \{\!\{(s, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'; \kappa)\}\!\} \uplus \Theta) \rightsquigarrow (R, \{\!\{(s, c'; \kappa)\}\!\} \uplus \Theta)}$$

WhileTrue
$$\frac{s(b)}{(R, \{\!\{(s, \mathbf{while}\ b\ \mathbf{do}\ c; \kappa)\}\!\} \uplus \Theta) \rightsquigarrow (R, \{\!\{(s, c; \mathbf{while}\ b\ \mathbf{do}\ c; \kappa)\}\!\} \uplus \Theta)}$$

WhileFalse
$$\frac{\neg s(b)}{(R, \{\!\{(s, \mathbf{while}\ b\ \mathbf{do}\ c; \kappa)\}\!\} \uplus \Theta) \rightsquigarrow (R, \{\!\{(s, \kappa)\}\!\} \uplus \Theta)}$$

Fork
$$\frac{}{(R, \{\!\{(s, \mathbf{fork}\ c; \kappa)\}\!\} \uplus \Theta) \rightsquigarrow (R, \{\!\{(s, c; \mathbf{done}), (s, \kappa)\}\!\} \uplus \Theta)}$$

Atomic
$$\frac{(R, s, c) \Downarrow (R', s')}{(R, \{\!\{(s, \langle c \rangle; \kappa)\rangle\}\!\} \uplus \Theta) \rightsquigarrow (R', \{\!\{(s', \kappa)\}\!\} \uplus \Theta)}$$

AtomicAbort
$$\frac{(R, s, c) \Downarrow \mathbf{abort}}{(R, \{\!\{(s, \langle c \rangle; \kappa)\rangle\}\!\} \uplus \Theta) \rightsquigarrow \mathbf{abort}}$$

2.2 Meaning of Assertions

The assertions of our proof system denote *resource bundles* $r \in \mathcal{R}$:

$$\begin{array}{rl} & \ell, v, \beta, h, u \in \mathbb{Z} \\ \alpha \in \mathit{Chunks} ::= & \ell \mapsto v \mid B(\beta, \overline{v}) \mid P(h, \beta, \overline{u}) \\ r \in \mathcal{R} = & \mathit{Chunks} \to [0, 1] \end{array}$$

A *chunk* is either a points-to chunk $\ell \mapsto v$, a box chunk $B(\beta, \overline{v})$, or a handle predicate chunk $P(h, \beta, \overline{u})$. A resource bundle is a function from chunks to real numbers between 0 and 1, inclusive. Note that the heaps $R \in \mathit{Heaps}$ can be identified with a subset of the resource bundles: we will identify heap R with the resource bundle $\{(\ell \mapsto v) \mapsto 1 \mid (\ell \mapsto v) \in R\}$.

We define *consistency* of a resource bundle as follows:

$$\frac{\begin{array}{c} \forall \ell, v_1, v_2.\ r(\ell \mapsto v_1) > 0 \wedge r(\ell \mapsto v_2) > 0 \Rightarrow v_1 = v_2 \\ \forall B, \beta, \overline{v}_1, \overline{v}_2.\ r(B(\beta, \overline{v}_1)) > 0 \wedge r(B(\beta, \overline{v}_2)) > 0 \Rightarrow \overline{v}_1 = \overline{v}_2 \\ \forall P, h, \beta_1, \overline{u}_1, \beta_2, \overline{u}_2.\ r(P(h, \beta_1, \overline{u}_1)) > 0 \wedge r(P(h, \beta_2, \overline{u}_2)) > 0 \Rightarrow \beta_1 = \beta_2 \wedge \overline{u}_1 = \overline{u}_2 \end{array}}{\mathsf{consistent}\ r}$$

We define satisfaction $r \vDash a$ of a closed assertion in the obvious way. We say a implies a' iff $\forall r.\ r \vDash a \Rightarrow r \vDash a'$.

2.3 Soundness Property

The soundness property that we target with our proof system is that if $\{\mathbf{true}\}\ c\ \{\mathbf{true}\}$ then $(\emptyset, \{\!|(\mathbf{0}, c; \mathbf{done})|\!\}) \not\rightsquigarrow^* \mathbf{abort}$.

2.4 Soundness with respect to the Big-Step Semantics

We extend the big-step semantics to operate on resource bundles as follows:

$$\frac{r + r_{\mathrm{F}} = R + r_{\mathrm{R}} \qquad (R, s, c) \Downarrow (R', s') \qquad R' + r_{\mathrm{R}} = r' + r_{\mathrm{F}}}{(r, s, c) \Downarrow (r', s')}$$

$$\frac{r + r_{\mathrm{F}} = R + r_{\mathrm{R}} \qquad (R, s, c) \Downarrow \mathbf{abort}}{(r, s, c) \Downarrow \mathbf{abort}}$$

Separation logic is sound with respect to the big-step semantics:

Lemma 1. *If $\{a\}\ c\ \{a'\}$ was derived without using the* CreateBox *or* Action *rules, and $r, s \vDash a$ and $(r, s, c) \Downarrow o$, then $\exists r', s'.\ o = (r', s') \wedge r', s' \vDash a'$.*

Proof. By induction on the derivation of the Hoare triple. □

2.5 Safety Relation

We define the *semantic assertions* $SemAsns = 2^{\mathcal{R} \times Stores}$.

We define a safety relation $\mathsf{safe} \subseteq Commands \times SemAsns \times \mathcal{R} \times Stores$ in Figure 4.

We prove a correspondence between correctness and safety of a command. (We prove a generalized property, involving a frame r and a weakened postcondition Q; this makes the induction hypothesis strong enough for the frame rule and the rule of consequence.)

Lemma 2. *If $\{a\}\ c\ \{a'\}$ and $a' * r \Rightarrow Q$, then $a * r \Rightarrow \mathsf{safe}(c, Q)$.*

Proof. By induction on the derivation. □

We define safety of a continuation:

$$\mathsf{safe}(\mathbf{done}, r, s) \qquad \frac{\mathsf{safe}(c, \mathsf{safe}(\kappa), r, s)}{\mathsf{safe}(c; \kappa, r, s)}$$

We define safety of a machine configuration:

$$\frac{\begin{array}{c}\iota_{\mathrm{B}} : \mathbb{Z} \rightharpoonup_{\mathrm{fin}} \mathcal{B} \times \mathbb{Z}^* \\ \iota_{\mathrm{P}} : \mathbb{Z} \rightharpoonup_{\mathrm{fin}} \mathcal{P} \times \mathbb{Z} \times \mathbb{Z}^* \qquad \rho : \mathrm{dom}\ \iota_{\mathrm{B}} \to \mathcal{R} \qquad r = R \uplus \iota_{\mathrm{B}} \uplus \iota_{\mathrm{P}} \\ r = \Sigma_\beta \rho(\beta) + \Sigma_i r_i \qquad \forall (\beta \mapsto (B, \overline{v}_\beta)) \in \iota_{\mathrm{B}}.\ \rho(\beta) \vDash I_B[\overline{v}_\beta \overline{w}_\beta / \overline{X}_B \overline{Y}_B] \\ \forall (h \mapsto (P, \beta, \overline{u})) \in \iota_{\mathrm{P}}.\ b_P[\overline{v}_\beta \overline{w}_\beta \overline{u} / \overline{X}_\beta \overline{Y}_\beta \overline{U}_P] \qquad \forall i.\ r_i, s_i \vDash \mathsf{safe}(\kappa_i, \mathbf{true})\end{array}}{\mathsf{safe}\,(R, \Sigma_i \{\!|(s_i, \kappa_i)|\!\})}$$

$$\frac{Q(r, s[x := s(e)])}{\mathsf{safe}(x := e, Q, r, s)}$$

LOOKUP
$$\frac{r \vDash s(e) \overset{\pi}{\mapsto} v \qquad Q(r, s[x := v])}{\mathsf{safe}(x := [e], Q, r, s)}$$

MUTATE
$$\frac{r, s \vDash e \mapsto v * (e \mapsto e' \mathrel{-\!\!*} Q)}{\mathsf{safe}([e] := e', Q, r, s)}$$

ATOMICNOBOX
$$\frac{\forall o.\ (r, s, c) \Downarrow o \Rightarrow \exists r', s'.\ o = (r', s') \wedge Q(r', s')}{\mathsf{safe}(c, Q, r, s)}$$

ATOMICBOX
$$\frac{\begin{array}{l}\textbf{boxclass } B(\overline{X}) \ \{ \\ \quad \textbf{inv } \exists \overline{Y}.\ I \\ \quad \cdots \\ \quad \textbf{action } A(\overline{Z}) \textbf{ req } b \textbf{ ens } b' \\ \quad \cdots \\ \quad \textbf{handlePred } P_i(\overline{U}_i) \ \{ \textbf{ inv } b_i \ \} \\ \} \\ r, s \vDash [\pi] B(\beta, \overline{v}) * \Pi_i P_i(\_, \beta, \overline{u}_i) * \\ \quad (\forall \overline{w}. \\ \quad\quad I[\overline{vw}/\overline{XY}] * \Pi_i b_i[\overline{vwu}_i/\overline{XYU}_i] \ \mathrel{-\!\!*} \\ \quad\quad \mathsf{safe}(c, \exists \overline{w'}. \\ \quad\quad\quad I[\overline{v}\,\overline{w'}/\overline{XY}] * \Pi_j (b''_j \vee b'_j[\overline{v}\,\overline{w'}\,\overline{u}'_j/\overline{XYU}'_j]) \\ \quad\quad\quad * (b \wedge b')[\overline{vw}\,\overline{w'}\,\overline{z}/\overline{XYY}'\,\overline{Z}] \\ \quad\quad\quad * ([\pi] B(\beta, \overline{v}) * \Pi_j (b''_j \vee P'_j(\_, \beta, \overline{u}'_j)) \mathrel{-\!\!*} Q)))\end{array}}{\mathsf{safe}(\langle c \rangle, Q, r, s)}$$

CREATEBOX
$$\frac{r, s \vDash I[\overline{vw}/\overline{XY}] * ([1] B(\_, \overline{v}) \mathrel{-\!\!*} Q)}{\mathsf{safe}(c, Q, r, s)}$$

Figure 4: Safety of a command

In words: a machine configuration is safe if there exists a set of box instances (with a box identifier, a box class, values for the box parameters, and values for the box invariant variables) and a set of handle predicate instances (with a handle predicate name, a handle identifier, a box identifier, and a set of handle predicate arguments) such that there exists a partitioning of the available resources (i.e. one points-to chunk for each heap cell plus one box chunk for each box instance plus one handle chunk for each handle) into one bundle for each box and one bundle for each thread, such that each box's bundle satisfies the box invariant and all handle predicate invariants pertaining to it, and each thread's bundle ensures the safety of that thread's continuation.

Lemma 3. *Safety of a machine configuration is preserved by machine steps:*

$$\mathsf{safe}\,\gamma \wedge \gamma \rightsquigarrow o \Rightarrow \exists \gamma'.\ o = \gamma' \wedge \mathsf{safe}\,\gamma'$$

Proof. By induction on the derivation of $\mathsf{safe}(\kappa_i, \mathbf{true}, r_i, s_i)$ for the thread i that performs the step. We elaborate an illustrative case.

- **Case** CreateBox. The thread's bundle r_i can be split into a part r_I that satisfies the resource invariant of some box class B, and a residue r'. We pick a new box identifier β and extend ι_{B} with the new box instance. We define the new bundle for thread i as $r_i' = r' + \{\!\!\{B(\beta, \overline{v})\}\!\!\}$. Since no handles have β as their box identifier, all constraints are satisfied. We finish by applying the induction hypothesis.

□

Theorem 1 (Soundness)**.** *If* $\{\mathbf{true}\}\ c\ \{\mathbf{true}\}$ *then* $(\emptyset, \{\!\!\{(\mathbf{0}, \kappa)\}\!\!\}) \not\rightsquigarrow^* \mathbf{abort}$.

Proof. The initial configuration is safe. We can derive by induction on the number of steps that any reachable outcome is a safe configuration. □

3 Verifying the Michael-Scott Queue

We show an encoding of the Michael-Scott queue concurrent data structure (for a garbage-collected language) into our formal syntax in Figure 5.

We wish to verify this implementation against the following specification:

$$\{I(\epsilon)\}\ q := \mathsf{create}()\ \{\mathsf{queue}(1, q, I)\}$$

$$\frac{\forall \alpha.\ \{I(\alpha) * P\}\ \rho\ \{I(\alpha \cdot v) * Q\}}{\{\mathsf{queue}(\pi, q, I) * P\}\ \mathsf{enqueue}(q, v, \rho)\ \{\mathsf{queue}(\pi, q, I) * Q\}}$$

$$\frac{\{I(\epsilon) * P\}\ \rho\ \{I(\epsilon) * Q(0)\} \qquad \forall v, \alpha.\ \{I(v \cdot \alpha) * P\}\ \rho'\ \{I(\alpha) * Q(v)\}}{\{\mathsf{queue}(\pi, q, I) * P\}\ x := \mathsf{dequeue}(q, \rho, \rho')\ \{\mathsf{queue}(\pi, q, I) * Q(x)\}}$$

These specifications are similar to the specification style of [6], but with some differences. When a queue is created, a *queue invariant* I, an assertion parameterized by a sequence of values, is associated with it. Upon creation of the queue, the invariant, instantiated with the empty sequence, is consumed. The client may include fractional ownership of ghost cells in this invariant to track

```
procedure create() returns (result){
  n := cons(next := 0, value := 0);
  q := cons(head := n, tail := n);
  result := q
}
procedure enqueue(q, x) {
  new := cons(next := 0, value := x);
  done := 0;
  while done = 0 do (
    ⟨t := [q.tail]⟩;
    ⟨n := [t.next]; if n = 0 then [t.next] := new⟩;
    if n = 0 then
      done := 1
    else
      ⟨t′ := [q.tail]; if t′ = t then [q.tail] := n⟩
  )
}
procedure dequeue(q) returns (result){
  done := 0;
  while done = 0 do (
    ⟨h := [q.head]⟩;
    ⟨n := [h.next]⟩;
    if n = 0 then (
      result := 0; done := 1
    ) else (
      ⟨t := [q.tail]; if t = h then [q.tail] := n⟩;
      ⟨h′ := [q.head]; if h′ = h then [q.head] := n⟩;
      if h′ = h then (
        ⟨result := [n.value]⟩;
        done := true
      )
    )
  )
}
```

Figure 5: The Michael-Scott queue

information about the contents of the queue. Therefore, when the queue is updated, these ghost cells may also need to be updated. This is made possible by allowing the proof author to associate ghost commands ρ and ρ' which update these ghost cells with calls of enqueue and dequeue. the Hoare triples for enqueue and dequeue have premises specifying the behavior of these ghost commands.

To verify the data structure, we declare the box class msqueue_box, as follows:

```
boxclass msqueue_box(q, l) {
  inv ∃i, nodes, vs, h, t.
    lseg(i, 0, nodes, vs) ∗ q.head ↦ nodes_h ∗ q.tail ↦ nodes_t
    ∗ h ≤ t ∗ |nodes| − 1 ≤ t ∗ l(vs_{h+1..|vs|})
  action enqueue(n, v)
    ens nodes′ = nodes · n ∧ vs′ = vs · v
  action dequeue()
    ens h′ = h + 1
  action move_tail()
    ens t′ = t + 1
  handlePred was_head(hd) { inv ∃j ≤ h. hd = nodes_j }
  handlePred was_head_with_succ(hd, nn) {
    inv ∃j ≤ h. hd = nodes_j ∧ nn = nodes_{j+1}
  }
  handlePred was_head_with_succ_not_tail(hd, nn) {
    inv ∃j ≤ h. hd = nodes_j ∧ nn = nodes_{j+1} ∧ j < t
  }
  handlePred node_has_value(n, v) { inv ∃j. n = nodes_j ∧ v = vs_j }
  handlePred was_tail(tn) { inv ∃j ≤ t. tn = nodes_j }
  handlePred was_tail_with_succ(tn, nn) {
    inv ∃j ≤ t. tn = nodes_j ∧ nn = nodes_{j+1}
  }
}
```

Here, we adopt three notational conventions (which have not yet been implemented in VeriFast): firstly, for each box invariant variable Y whose primed version is not mentioned in an action postcondition, that postcondition gets an additional conjunct saying $Y' = Y$; secondly, each action postcondition implicitly gets an additional disjunct saying that nothing has changed; thirdly, an action precondition that is not declared explicitly defaults to **true**.

A proof outline for the queue is shown in Figures 6 and 7.

4 Additional Features

In this section we briefly describe additional features of VeriFast's shared boxes.

4.1 Handle Identifiers

The examples we used in the preceding sections had the property that at no point in time did any thread perform a distinguished role in the protocol: all threads were subject to the same restrictions, or, in other words still, the rely condition did not mention thread identities.

```
predicate queue(f, q, I) = [f]msqueue_box(_, q, I)
procedure create() returns (result){
  {I(ϵ)}
  n := cons(next := 0, value := 0);
  q := cons(head := n, tail := n);
  result := q
  {lseg(n, 0, n, 0) * q.head ↦ n * q.tail ↦ n * I(ϵ)}
  {[1]msqueue_box(_, q, I)}   CreateBox
}
procedure enqueue(q, x) {
  {queue(f, q, I) * P}
  new := cons(next := 0, value := x);
  done := 0;
  {queue(f, q, I) * (done = 0 * new.next ↦ 0 * new.value ↦ x * P ∨ done = 1 * Q)}
  while done = 0 do (
    {[f]msqueue_box(β, q, I) * new.next ↦ 0 * new.value ↦ x * P}
    ⟨t := [q.tail]⟩;
    {[f]msqueue_box(β, q, I) * new.next ↦ 0 * new.value ↦ x * P * was_tail(_, β, t)}
    ⟨n := [t.next]; if n = 0 then ([t.next] := new; ρ)⟩;
    { [f]msqueue_box(β, q, I) * (n = 0 * Q ∨
      n ≠ 0 * new.next ↦ 0 * new.value ↦ x * P * was_tail_with_succ(_, β, t, n)) }
    if n = 0 then
      done := 1
    else
      ⟨t' := [q.tail]; if t' = t then [q.tail] := n⟩
  )
  {queue(f, q, I) * Q}
}
```

Figure 6: Proof of the Michael-Scott queue, part 1 of 2

```
procedure dequeue(q) returns (result){
  {queue(f, q, I) * P}
  done := 0;
  {queue(f, q, I) * (done = 0 * P ∨ done = 1 * Q(result))}
  while done = 0 do (
    {[f]msqueue_box(_, q, I) * P}
    ⟨h := [q.head]⟩;
    {[f]msqueue_box(β, q, I) * P * was_head(_, β, h)}
    ⟨n := [h.next]; if n = 0 then ρ⟩;
    {[f]msqueue_box(β, q, I) * (n = 0 * Q ∨ n ≠ 0 * P * was_head_with_succ(_, β, h, n))}
    if n = 0 then (
      result := 0; done := 1
    ) else (
      ⟨t := [q.tail]; if t = h then [q.tail] := n⟩;
      {[f]msqueue_box(β, q, I) * P * was_head_with_succ_not_tail(_, β, h, n)}
      ⟨h' := [q.head]; if h' = h then ([q.head] := n; ρ')⟩;
      { [f]msqueue_box(β, q, I) *
        (h = h' * (∃v. Q(v) * is_good_node(_, β, n, v)) ∨ h ≠ h' * P) }
      if h' = h then (
        ⟨result := [n.value]⟩;
        done := true
      )
    )
  )
  {queue(f, q, I) * Q(result)}
}
```

Figure 7: Proof of the Michael-Scott queue, part 2 of 2

In many other concurrent algorithms, thread identities do play a role in a rely condition. For example, in the case of a spin lock, only the thread that acquired the lock may release it (except if the thread explicitly yielded ownership of the lock to some other thread). As another example, in an algorithm that uses hazard pointers for memory reclamation, such as the Treiber stack [7], only the thread that removed a node from the data structure may deallocate it.

VeriFast supports these scenarios by associating a *handle identifier* with each handle predicate. Action specifications may specify which participants may perform the action by constraining the special variable actionHandles, which denotes the list of the handle identifiers of the handles consumed by the action. As a result, only those threads which own particular handle predicate chunks can perform certain actions.

Similarly, a handle predicate invariant may mention the handle predicate's identifier using the special variable predicateHandle.

To support stable and unique identities, for each handle predicate that is produced by an action, the proof author must specify the handle identifier (which must be the identity of one of the handles that was consumed) or else that the handle identifier should be a fresh one.

The following examples that ship with VeriFast in the `examples/shared_boxes` directory use handle identifiers:

Example	Description
`spinlock.c`	Spinlock
`ticket_lock.c`	Ticketed lock
`concurrentstack.c`	Treiber stack with hazard pointers
`cowl.c`	Copy-on-write list

4.2 Nested actions

In order to build fine-grained concurrent data structures on top of other fine-grained concurrent data structures, it is useful to be able to nest actions. Note, however, that care must be taken to deal correctly with box re-entry, i.e. performing multiple nested actions on the same box. Obviously, it would be unsound to produce the box invariant multiple times.

VeriFast supports nested actions. Box re-entry is ruled out by assigning a unique *box level* to each box (whose relationship to existing box levels may be specified by the proof author), and checking that an inner action is on a higher-level box than its outer action.

The following examples use nested actions:

Example	Description
`gotsmanlock.c`	Gotsman lock [5]
`atomic_integer.c`	Atomic integer
`spinlock_with_atomic_integer`	Spinlock on top of atomic integer
`ticketlock_with_atomic_integer`	Ticketed lock on top of atomic integer

It is important to note, however, that composing fine-grained concurrent data structures each verified using shared boxes does not always require nested actions. For example, the examples `cell_refcounted.c`, `cowl.c`, and `lcl_set.c` (a set implementation using a lock-coupling list) are built on top of `gotsmanlock.c` without the need for nested actions.

4.3 Action permissions

An alternative way to deal with algorithms where participants have distinct roles, is using *action permissions*, first introduced in CAP [3]. VeriFast supports action permissions: an action may be declared as `permbased`. As in CAP, an action permission chunk is produced when the box is created. Performing a `permbased` action requires (a fraction of) the action permission chunk.

The examples `ticketlock_cap.c` and `ticketlock_with_atomic_integer.c` use action permissions.

Both of these examples use `permbased` actions that have parameters. In this case, upon creation of the box, conceptually a distinct chunk is produced for each value of the parameter. To represent this finitely in VeriFast's symbolic heap, a *dispenser* chunk is produced which represents the action permission chunks for all parameter values except for a given list of values for which a separate action permission has been split off.

4.4 Spatial handle predicate invariants

In CAP [3], stability of a shared region assertion may depend on chunks locally held by the thread. VeriFast supports this as well by allowing spatial handle predicate invariants. One example that illustrates this is `ticketlock_cap.c`.

5 Conclusion

Through the mechanism of shared boxes, VeriFast integrates rely-guarantee reasoning into its separation logic-based program logic. We introduced the mechanism through the motivating examples of a monotonic counter and the Michael-Scott queue, formalized the proof system and sketched a soundness proof, and briefly discussed additional features and additional examples available in the VeriFast distribution. Perhaps most notably, we achieved a reasonably clean proof of a Treiber stack with hazard pointers.

Acknowledgements

This work was supported by the European Commission under EU FP7 FET-Open project ADVENT (grant number 308830).

References

[1] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

[2] Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE*, 2009.

[3] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[4] Xinyu Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

[5] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.

[6] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.

[7] M. M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6), 2004.

[8] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *CACM*, 19(5):279–285, May 1976.

[9] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.