Specifying I/O using Abstract Nested Hoare Triples in Separation Logic

Willem Penninckx
imec-DistriNet, Department of Computer Science
KU Leuven
Leuven, Belgium
willem@willemp.be

Amin Timany
imec-DistriNet, Department of Computer Science
KU Leuven
Leuven, Belgium
amin.timany@cs.kuleuven.be

Bart Jacobs
imec-DistriNet, Department of Computer Science
KU Leuven
Leuven, Belgium
bart.jacobs@cs.kuleuven.be

Abstract

We propose a separation logic-based approach for modular specification and verification of the I/O behavior of a program. The approach uses higher-order separation logic predicates to express *abstract nested Hoare triples* that abstractly associate a precondition and a postcondition with an I/O action. The approach supports verifying higher-level I/O actions built on top of lower-level ones (e.g. the I/O abstractions offered by the programming language's standard library, implemented on top of system calls), as well as virtual I/O actions that in fact only manipulate memory, against specifications that are indistinguishable from those of the "primitive I/O actions".

CCS Concepts • **Theory of computation → Program specifications**; *Pre- and post-conditions*; *Program verification*;

Keywords input/output, modular program verification, module specifications, separation logic

ACM Reference Format:
Willem Penninckx, Amin Timany, and Bart Jacobs. 2019. Specifying I/O using Abstract Nested Hoare Triples in Separation Logic. In *Formal Techniques for Java-like Programs (FTfJP'19), July 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3340672.3341118

1 The Problem of Specifying I/O Behavior

We introduce the problem addressed by this paper, as well as our proposed approach to address it, in the context of our VeriFast [3, 10] approach and tool for modular formal verification of C programs.[1] VeriFast takes as input a C compilation unit, annotated with function specifications, as well as other specification constructs such as *abstract predicates* that describe the layout of data structures and proof hints such as loop invariants, written in a variant of separation logic [6] inside specially marked comments. It performs symbolic execution of each function, starting from a symbolic state that represents an arbitrary state that satisfies the precondition, and checking at every return point that the symbolic state satisfies the postcondition. Symbolic execution of function calls uses the callee's specification, not its implementation. If the tool reports "0 errors found" for all compilation units of a program, then, barring bugs in the tool, and assuming that the specifications provided for the system APIs used by the program are sound, all possible executions of the program are free of undefined behavior and comply with the stated specifications.

In this paper, we consider the problem of modular verification of the interactive (I/O) behavior of programs such as the following:

```
#include <stdio.h>

int main() {
   putchar('h');
   putchar('i');
   return 0;
}
```

This program uses the `putchar` function from the C standard library, declared in header file `stdio.h`, to output the message "hi" to the standard output stream. However, the default version of `stdio.h` currently shipping with VeriFast does not allow us to specify this: its specification for `putchar` is simply

```
void putchar(char c);
//@ requires true;
//@ ensures true;
```

allowing us to verify the specification

```
int main();
//@ requires true;
//@ ensures true;
```

FTfJP'19, July 15, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6864-3/19/07...$15.00
https://doi.org/10.1145/3340672.3341118

[1]VeriFast supports Java as well.

for our program, the same specification satisfied also by the program that outputs "bye", or nothing at all.

2 Abstract Nested Hoare Triples

2.1 Basic Idea

Our proposed approach for specifying and verifying the I/O behavior of programs in separation logic is to associate a *higher-order separation logic predicate* with each I/O action, which we call its *transition predicate*. We use an I/O action's transition predicate to abstractly associate precondition-postcondition pairs with the I/O action. In this paper, we adopt the convention of naming a transition predicate after its associated I/O action, with an underscore appended at the end. For example, the assertion `putchar_(P, c, Q)` uses the transition predicate `putchar_` to associate precondition `P` and postcondition `Q` with the action of outputting the character `c`. Here, `P` and `Q` are values of type `predicate()`, i.e. they are themselves separation logic predicates. Notice that such assertions are in fact abstract versions of *nested Hoare triples* [9]; therefore, we call them *abstract nested Hoare triples*. We can use them to specify function `putchar` very abstractly as follows:

```
/*@
predicate putchar_(predicate() P, char c, predicate() Q);
@*/

void putchar(char c);
//@ requires putchar_(?P, c, ?Q) &*& P();
//@ ensures Q();
```

Here, `?P` is VeriFast syntax for introducing a logical variable `P`, and `&*&` is the separating conjunction. The scope of a logical variable introduced in a function specification's precondition is the entire specification, i.e. the precondition and the postcondition. Logical variables introduced in the precondition are universally quantified at the level of the specification. That is, the meaning of the above specification is that for any predicates `P` and `Q`, if `putchar_(P, c, Q)` and (separately) `P` hold in a given state, then `putchar(c)` may be called, and after such call, `putchar_(P, c, Q)` and (some resources described by) `P()` will have been consumed and (some resources described by) `Q()` will have been produced.

We can now specify our example program as follows:

```
int main();
/*@ requires putchar_(?P1, 'h', ?P2) &*&
     putchar_(P2, 'i', ?P3) &*& P1(); @*/
//@ ensures P3();
```

According to this specification, function `main` starts out with (some resources described by) `P1()`. Per transition predicate `putchar_(P1, 'h', P2)`, this allows it to output `'h'`. (It does not allow it to output `'i'` at this point, since to do so it would need `P2()`, which it does not yet have.) By outputting `'h'`, it loses `P1()` but gains `P2()`. Then, by outputting `'i'`, it loses `P2()` and gains `P3()`, which allow it to satisfy its postcondition and terminate.

Notice that this specification expresses the desired property: it states that `main` shall only ever output a prefix of "hi", and furthermore, that if it returns, it shall have outputted exactly "hi".

Note: we will sometimes refer to the predicates used as the precondition or postcondition argument of transition predicates, such as the predicates `P1`, `P2`, and `P3` in the specification of `main` above, as *places*, and to assertions that assert such a place, such as `P1()` and `P3()` in the specification of `main`, as *tokens*; indeed, the precondition of `main` can be read as specifying a *Petri net* with a transition from place `P1` to `P2` labelled by action `putchar('h')` and a transition from `P2` to `P3` labelled by action `putchar('i')`, and with a single *token* at place `P1`. *Firing* the two transitions causes the token to move from place `P1` to place `P3`. However, the reader can ignore this analogy if they do not find it useful.

2.2 Building Higher-Level Actions

This approach straightforwardly supports defining higher-level I/O actions on top of lower-level ones. For example, consider the function `putc_beep`, defined in terms of `putchar` and some imagined I/O action `beep`:

```
//@ #define pr predicate()

/*@ predicate putc_beep_(pr P, char c, pr Q) =
     putchar_(P, c, ?R) &*& beep_(R, Q); @*/

void putc_beep(char c)
//@ requires putc_beep_(?P, c, ?Q) &*& P();
//@ ensures Q();
{ putchar(c); beep(); }
```

Notice that, even though function `putc_beep` is not a primitive I/O action, clients cannot tell this from its specification; its form is exactly analogous to that of `putchar`. (Note: the syntax `?R` in a predicate definition introduces an existentially quantified logical variable. The definition means: `putc_beep_(P, c, Q)` holds if and only if there exists a predicate `R` such that `putchar_(P, c, R)` holds and, separately, `beep_(R, Q)` holds.)

2.3 Underspecification

Besides elegantly supporting transparently building higher-level I/O operations on top of lower-level ones, this approach has the additional feature of elegantly supporting underspecification, in two forms: in the form of alternative paths of transition predicates between two places (predicates) `P` and `Q`, and in the form of enabling multiple *concurrent* paths of transition predicates by *splitting* and *joining* places. In the following example, we use both forms to express that the program may print either "hi!" or "hey!", and that furthermore, the program must beep at some point after printing the "h" and before printing the "!":

```
int main();
/*@ requires putchar_(?P1, 'h', sep(P2, P3)) &*&
    &*& putchar_(P2, 'i', ?P4)
    &*& putchar_(P2, 'e', ?P5) &*&
    putchar_(P5, 'y', P4) &*& beep_(P3, ?P6) &*&
    putchar_(sep(P4, P6), '!', ?P7) &*&
    P1(); @*/
//@ ensures P7();
```

where `sep(P, Q)` denotes the separating conjunction of predicates `P` and `Q`, as a value of type `predicate()`. (VeriFast does not support directly writing `P() &*& Q()` in a value position.) (To continue the Petri net analogy, a transition predicate whose precondition/postcondition is of the form `sep(P1, P2)` can be thought of as a transition with multiple incoming/outgoing arrows, consuming/producing a token at each incoming/outgoing place.)

2.4 Input

Another feature of the approach is elegant support for specifying *input actions*, and specifying how a program's behavior should depend on the values retrieved from the environment. Furthermore, it supports expressing assumptions about such values, as illustrated by the following example:

```
void toUpper()
/*@ requires getchar_(?P1, ?c, ?P2) &*&
    'a' <= c &*& c <= 'z' &*&
    putchar_(P2, c - 'a' + 'A', ?P3) &*& P1(); @*/
//@ ensures P3();
{ char c = getchar(); putchar(c - 'a' + 'A'); }
```

where `getchar`'s specification is as follows:[2]

```
char getchar();
//@ requires getchar_(?P, ?c, ?Q) &*& P();
//@ ensures Q() &*& result == c;
```

Notice that in this approach, a function's precondition may refer to the values yielded by input actions that the program has yet to perform; this can be thought of as a form of *prophecy variables*. For example, in the precondition of `getchar`, logical variable `c` refers to the character that the program will receive from the environment when it calls `getchar`.

The approach we propose here has evolved from earlier work [7]. That earlier work supports the features shown so far, but it does not support the features we show in the remainder of this paper.

2.5 I/O and Memory

Our approach supports transparently implementing higher-level I/O operations in terms of a combination of lower-level I/O operations and memory manipulation. For example, consider the following implementation of the `flush` and `putchar` C standard library functions[3] in terms of the `write` system call:

```
char buffer[1000];
int count;
/*@
predicate buffer_token(list<char> cs, pr Q) =
    count |-> ?n &*&
    buffer[..n] |-> cs &*& buffer[n..1000] |-> _ &*&
    write_(?P, cs, Q) &*& P();
@*/
void flush()
//@ requires buffer_token(_, ?Q);
//@ ensures buffer_token({}, Q);
{ write(buffer, count); count = 0; }
void putchar(char c)
//@ requires buffer_token(_, ?P) &*& write_(P, {c}, ?Q);
//@ ensures buffer_token(_, Q);
{
    if (count == 1000) flush();
    buffer[count++] = c;
}
```

(`{}` and `{c}` are VeriFast syntax for the empty list and the singleton list containing character `c`, respectively.) We can verify these functions against the stated specifications, if we assume the following (simplified) declaration of `write`:

```
/*@
predicate write_char_(pr P, char c, pr Q);
predicate write_(pr P, list<char> cs, pr Q) =
    switch (cs) {
    case nil: return Q == P;
    case cons(c, cs0): return
        write_char_(P, c, ?R) &*& write_(R, cs0, Q);
    };
@*/
void write(char *buffer, int count);
/*@ requires buffer[..count] |-> ?cs &*&
    write_(?P, cs, ?Q) &*& P(); @*/
//@ ensures Q();
```

The predicate `buffer_token(cs, Q)` asserts ownership of the global variables `count` and `buffer`, as well as a token `P()` that allows the program to obtain a token at `Q` (i.e. to obtain `Q()`) by writing the contents `cs` of the buffer. The specification for `putchar` states that, given a token that allows the program to reach `Q` after flushing the pre-state buffer and then writing `c`, it produces a token that allows the program to reach `Q` after flushing the post-state buffer.

This specification is fine, except that it is different from the specification for `putchar` we saw earlier. The specification we really want is the following:

```
void putchar(char c);
//@ requires putchar_(?P, c, ?Q) &*& P();
//@ ensures Q();
```

[2]The real `getchar` function returns an **int** and may report failure; for simplicity, we ignore these complications here.

[3]This is a simplification; the real function is called `fflush` and takes the stream to be flushed as an argument.

Fortunately, however, we can match the two specifications by defining `putchar_` appropriately:

```
predicate_ctor buffer_token1(pr Q)() =
    buffer_token(_, Q);
predicate putchar_(pr P, char c, pr Q) =
    write_(?P0, {c}, ?Q0) &*&
    P == buffer_token1(P0) &*&
    Q == buffer_token1(Q0);
```

(The *predicate constructor* `buffer_token1` allows the assertion `buffer_token(_, Q)`, for some particular `Q`, to be used as a value of type **`predicate`**`()`.) In fact, we can make *any* function specification

```
void foo(int arg);
//@ requires P(arg, ?x);
//@ ensures Q(x);
```

match an "I/O-style" specification

```
void foo(int arg)
//@ requires foo_(?P1, arg, ?Q1) &*& P1();
//@ ensures Q1();
```

by defining the transition predicate `foo_` as

```
predicate_ctor P0(int arg, X x)() = P(arg, x);
predicate_ctor Q0(X x)() = Q(x);
predicate foo_(pr P1, int arg, pr Q1) =
    exists(?x) &*& P1 == P0(arg, x) &*& Q1 == Q0(x);
```

Similarly, we can prove an I/O-style specification for `flush`:

```
predicate_ctor buffer_token0(pr Q)() =
    buffer_token({}, Q);
predicate flush_(pr P, pr Q) =
    exists<pr>(?Q0) &*&
    P == buffer_token1(Q0) &*&
    Q == buffer_token0(Q0);
void flush();
//@ requires flush_(?P, ?Q) &*& P();
//@ ensures Q();
```

We can now verify a client program of `putchar` and `flush` in a way that is completely agnostic to the way these functions are implemented:

```
int main()
/*@ requires putchar_(?P1, 'h', ?P2)
    &*& flush_(P2, ?P3) &*& P1(); @*/
//@ ensures P3();
{ putchar('h'); flush(); return 0; }
```

and then use this proof to verify a lower-level specification of the program in terms of system calls:

```
int start()
/*@ requires write_(?Q1, {'h'}, ?Q2)
    &*& Q1() &*& module(stdio, true); @*/
//@ ensures Q2() &*& module(stdio, false);
{ return main(); }
```

(In some C implementations on Linux, a program's `start` function is its actual low-level entry point. It is usually auto-generated by the C compiler. It first initializes the standard library and then calls the `main` function.) (The assertion `module(M, init)` asserts ownership of the global variables of module `M`, either initialized to zero if `init` is true, or in an arbitrary state otherwise.)

2.6 Mixing Abstraction Levels

Consider now a variant of the above program, where `main` uses both the high-level `putchar` operation and the low-level `beep` operation:

```
int main()
/*@ requires beep_(?P1, ?P2) &*& putchar_(P2, 'h', ?P3)
    &*& flush_(P3, ?P4) &*& P1(); @*/
//@ ensures P4();
{ beep(); putchar('h'); flush(); return 0; }
int start()
/*@ requires beep_(?Q1, ?Q2) &*& write_(Q2, {'h'}, ?Q3)
    &*& Q1() &*& module(stdio, true); @*/
//@ ensures Q3() &*& module(stdio, false);
{ return main(); }
```

Notice that we cannot verify the call of `main` in `start`: to prove `beep_(P1, P2)` we only have `beep_(Q1, Q2)` available; therefore, we must instantiate `P1` with `Q1` and `P2` with `Q2`. However, the unknown predicate `Q2` is not (necessarily) of the form `buffer_token1(_)`, as required by `putchar_`. In fact, we know `Q1` (and therefore `Q2`) does not include ownership of the global variables `count` and `buffer`, since we have `Q1()` and separately `module(stdio, true)`. However, `putchar` needs access to these variables.

Notice, however, that we can solve this problem if we can assume a *frame axiom* for transition predicate `beep_`:

$$\texttt{beep\_(Q1, Q2)} \Rightarrow \texttt{beep\_(sep(Q1, ebuf), sep(Q2, ebuf))}$$

similar to the Frame Rule of separation logic, as applied to function call `beep()`:

$$\textsc{Frame}\quad \frac{\{\texttt{Q1}\}\ \texttt{beep()}\ \{\texttt{Q2}\}}{\{\texttt{Q1} * \texttt{ebuf()}\}\ \texttt{beep()}\ \{\texttt{Q2} * \texttt{ebuf()}\}}$$

where `ebuf()` asserts ownership of the empty buffer:

```
predicate ebuf() = count |-> 0 &*& buffer[..1000] |-> _;
```

To enable this type of scenario, we therefore introduce a convention that each transition predicate shall be accompanied by such a frame axiom. In VeriFast, this takes the form of a *lemma function*, a type of ghost function:

```
/*@
lemma void beep__frame(pr R);
    requires beep_(?P, ?Q);
    ensures beep_(sep(P, R), sep(P, Q));
@*/
```

This lemma *almost* allows us to verify function `start` above. The only problem is that while `sep(Q2, ebuf)` *implies* `buffer_token1(Q2)`, it is not *equivalent* and hence not *equal* to it.[4] To complete the proof, we also need a *weakening* property:

$$\frac{\texttt{P0} \Rightarrow \texttt{P1} \qquad \texttt{Q0} \Rightarrow \texttt{Q1}}{\texttt{beep\_(P1, Q0)} \Rightarrow \texttt{beep\_(P0, Q1)}}$$

analogous to Hoare logic's Rule of Consequence applied to a `beep()` call:

$$\textsc{Conseq}\quad \frac{\texttt{P0} \Rightarrow \texttt{P1} \qquad \{\texttt{P1}\}\ \texttt{beep()}\ \{\texttt{Q0}\} \qquad \texttt{Q0} \Rightarrow \texttt{Q1}}{\{\texttt{P0}\}\ \texttt{beep()}\ \{\texttt{Q1}\}}$$

So, by also introducing the convention that each transition predicate shall be accompanied by a weakening axiom, we enable support for verifying programs where some function specifications mix I/O actions of different abstraction levels.

Note: we can easily make our definitions of `putchar_` and `flush_` above satisfy the frame and weakening properties as well by adapting them as follows:

```
predicate putchar_(pr P, char c, pr Q) =
    write_(?P0, {c}, ?Q0) &*&
    exists<pr>(?R) &*&
    implies(P, sep(buffer_token1(P0), R)) &*&
    implies(sep(buffer_token1(Q0), R), Q);
predicate flush_(pr P, pr Q) =
    exists<pr>(?R) &*& exists<pr>(?Q0) &*&
    implies(P, sep(buffer_token1(Q0), R)) &*&
    implies(sep(buffer_token0(Q0), R), Q);
```

This allows these I/O actions to be mixed in function specifications with even higher-level ones.

2.7 Virtual Input

Clearly, our approach also supports I/O-style specifications for purely "virtual I/O actions", such as those of Java's `ByteArrayOutputStream`, which simply appends any "outputted" bytes to an in-memory buffer. A more interesting instance of this is in the context of the verification of multithreaded programs, where threads communicate through shared queues. One might want to verify each thread in such a way that their proof can be reused unchanged if one later decides to run the threads in separate processes or on separate machines, communicating via sockets. Our approach enables this, by enabling I/O-style specifications for the shared queue operations; these can then later be replaced transparently by actual I/O operations, such as socket operations.

Notice, however, that in our proposed specification style for input operations, as illustrated by the specification of `getchar` above, the input operation's transition predicate takes the operation's *result* as an *argument*. In general, to construct a definition for such a transition predicate for a virtual input operation, *prophecy variables* are needed. Consider for example a (contrived) implementation of `getchar` in terms of a random number generator:[5]

```
char getchar() {
    for (;;) {
        int x = rand();
        if ('a' <= x && x <= 'z')
            return x;
    }
}
```

where the specification of `rand()` is:

```
int rand();
//@ requires true;
//@ ensures true;
```

This implementation of `getchar` returns some arbitrary lowercase letter. We now want to verify the following program:

```
char buffer;
void putchar(char c) { buffer = c; }
int main()
//@ requires true;
//@ ensures 'A' <= result && result <= 'Z';
{ toUpper(); return c; }
```

Here, we use function `toUpper` specified and verified above.

In order to be able to reuse the existing proof of `toUpper` in the proof of this program, we need to verify our implementation of `getchar` against the specification shown above. We can do so by using an encoding of *prophecy variables* into VeriFast as follows:

```
typedef long long pvar;
/*@
#define set fixpoint(int, bool)
predicate pvar(pvar id, set S, int v);
predicate pvar_params(set S, int w) = S(w) == true;
@*/
pvar create_pvar();
//@ requires pvar_params(?S, _);
//@ ensures pvar(result, S, ?v) &*& S(v) == true;
void pvar_assign(pvar x, int v);
//@ requires pvar(x, ?S, ?v0) &*& S(v) == true;
//@ ensures v == v0;
```

This encoding defines a function for creating a prophecy variable and a function for assigning a value to a prophecy variable created earlier. When creating a prophecy variable, a set of integers `S` and a witness `w` showing that the set is nonempty have to be supplied. The operation produces a predicate `pvar(x, S, v)` asserting that `x` is the identifier of a prophecy variable constrained by set `S` and with *prophecized value* `v`, which is in set `S`. Assigning the prophecy variable consumes the `pvar` predicate and requires that the supplied value `v` is in the set `S`, and produces the fact that the supplied value equals the prophecized value.

[4]In fact, in VeriFast predicate extensionality does not hold, so the predicate values would not be equal even if they were equivalent.

[5]A more realistic example, of a multithreaded chat server where prophecy variables are needed to deal with nondeterminism caused by the interleaving of threads, is offered in the accompanying technical report of this paper [8].

Note that the functions `create_pvar` and `pvar_assign` are declared in this VeriFast encoding as real functions, not as ghost functions, even though they are in fact no-ops at run time. This is necessary for soundness: if `pvar_assign` was a ghost operation, we could `v + 1` to the prophecy variable, where `v` is the prophecized value, leading to the contradiction `v + 1 == v`. To eliminate this hack, we plan to equip VeriFast with support for multiple levels of ghost code, to ensure that the prophecized value is "more ghost" than the prophecy variable assignment operation.

We can now prove the example program as follows, using the frame properties for `getchar_` and `putchar_` to frame off ownership of global variables `p` and `buffer`, respectively:

```
pvar p;
char buffer;
/*@
fixpoint bool lc(int x) { return 'a' <= x && x <= 'z'; }
predicate_ctor getchar_pre(char c)() =
    p |-> ?x &*& pvar(x, lc, c);
predicate getchar_post() = p |-> _;
predicate getchar_(pr P, char c, pr Q) =
    exists<pr>(?R) &*&
    implies(P, sep(getchar_pre(c), R) &*&
    implies(sep(getchar_post, R), Q);
predicate putchar_pre() = buffer |-> _;
predicate_ctor putchar_post(char c)() =
    buffer |-> c &*& 'A' <= c && c <= c <= 'Z';
predicate putchar_(pr P, char c, pr Q) =
    exists<pr>(?R) &*&
    implies(P, sep(putchar_pre, R)) &*&
    implies(sep(putchar_post(c), R), Q);
@*/
char getchar()
//@ requires getchar_(?P, ?c, ?Q) &*& P();
//@ ensures Q() &*& result == c;
{
    for (;;) {
        int x = rand();
        if ('a' <= x && x <= 'z') {
            pvar_assign(p, x);
            return x;
        }
    }
}
void putchar(char c)
//@ requires putchar_(?P, c, ?Q) &*& P();
//@ ensures Q();
{ buffer = c; }
int main()
//@ requires module(main, true);
//@ ensures module(main, false);
{
    //@ close pvar_params(lc, 'a');
    p = create_pvar();
    toUpper();
    return c;
}
```

3 Related Work

Our earlier work [7] proposed a specification style for I/O based on notions of Petri nets and tokens built into the logic. In that logic, I/O-style specifications cannot be interpreted as predicates about memory, preventing the use of I/O-style specifications for semi-virtual (e.g. buffered) or fully virtual I/O operations.

Férée *et al.* [1] performed a separation-logic based machine-checked verification of a number of classical UNIX command-line tools largely implemented as shallowly embedded functions in HOL4. Their specifications for the I/O operations are in terms of a separation logic predicate `STDIO(fs)` where `fs` reflects the state of the file system.

Ho *et al.* [2] propose an approach for proof-producing synthesis of imperative programs with I/O from monadic functions of higher-order logic. For reasoning about I/O, they use the same approach as Férée *et al.*.

Koh *et al.* [4] used the Verified Software Toolchain to produce a machine-checked separation logic-based proof of the functional correctness of a simple networked server written in C. In their approach, specifications of I/O operations are in terms of a separation logic predicate `ITree(t)` where `t` is an *interaction tree*, a coinductive (i.e. lazy) tree of I/O operations and (angelic) choice points. Branching occurs on choice points and on the result values (i.e. inputs) yielded by I/O operations. Their trees do not seem to be able express assumptions about the environment.

4 Conclusion

In this short paper, we outlined an approach for modular reasoning about the I/O behavior of programs in a VeriFast-like setting. It supports building higher-level I/O operations on top of lower-level ones, mixing I/O operations of varying abstraction levels in function specifications, and treating semi-virtual or fully virtual I/O operations as if they were primitive ones.

The technical report accompanying this paper [8] offers a formal definition of the syntax and semantics of a programming language with I/O, a formal definition and soundness proof of the proposed logic, and formalized versions of a buffered I/O example and a virtual I/O example involving a multithreaded chat server where threads communicate through a shared queue. The examples shown in the technical report have been formalized both in VeriFast and in Coq; for the latter, we used the Iris [5] library for separation logic reasoning about concurrent higher-order programs.

Note: the technical report proposes a formalization of prophecy variables. We explicitly do not claim that part as a contribution of this paper; we are preparing a paper on that topic for submission elsewhere.

Acknowledgments

This material is based upon work supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

References

[1] Hugo Férée, Johannes Åman Pohjola, Ramana Kumar, Scott Owens, Magnus O. Myreen, and Son Ho. 2018. Program Verification in the Presence of I/O - Semantics, Verified Library Routines, and Verified Applications. In *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers (Lecture Notes in Computer Science)*, Ruzica Piskac and Philipp Rümmer (Eds.), Vol. 11294. Springer, 88–111. https://doi.org/10.1007/978-3-030-03592-1_6

[2] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. 2018. Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions. In *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings (Lecture Notes in Computer Science)*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.), Vol. 10900. Springer, 646–662. https://doi.org/10.1007/978-3-319-94205-6_42

[3] Bart Jacobs (Ed.). 2018. *VeriFast 18.02*. Zenodo. https://doi.org/10.5281/zenodo.1182724

[4] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 234–248. https://doi.org/10.1145/3293880.3294106

[5] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP*.

[6] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science)*, Laurent Fribourg (Ed.), Vol. 2142. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

[7] Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *ESOP*.

[8] Willem Penninckx, Amin Timany, and Bart Jacobs. 2019. Abstract I/O Specification. *CoRR* abs/1901.10541 (2019). arXiv:1901.10541 http://arxiv.org/abs/1901.10541

[9] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. 2011. Nested Hoare triples and frame rules for higher-order store. *Logical Methods in Computer Science* 7, 3 (2011).

[10] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2015. Featherweight VeriFast. *Logical Methods in Computer Science* 11, 3 (2015). https://doi.org/10.2168/LMCS-11(3:19)2015