Modular, compositional and sound verification of the input/output behavior of programs

Willem Penninckx
Bart Jacobs
Frank Piessens

Report CW 663, May 2014

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

Modular, compositional and sound verification of the input/output behavior of programs

Willem Penninckx
Bart Jacobs
Frank Piessens

Report CW 663, May 2014

Department of Computer Science, KU Leuven

Abstract

We present a sound verification approach for verifying input/output properties of programs. Our approach supports compositionality (you can build high-level I/O actions on top of low-level ones), modularity (you can define and implement input/output actions without taking into account which other actions exist) and other features.

Modular, compositional and sound verification of the input/output behavior of programs

Willem Penninckx, Bart Jacobs, Frank Piessens

Department of Computer Science, KU Leuven, Belgium

Report CW663, April 2014

Abstract

We present a sound verification approach for verifying input/output properties of programs. Our approach supports compositionality (you can build high-level I/O actions on top of low-level ones), modularity (you can define and implement input/output actions without taking into account which other actions exist) and other features.

1 Introduction

Many software verification approaches are based on Hoare logic. A Hoare triple consists of a precondition, a program, and a postcondition. If a Hoare triple is true, then every execution of the program starting from any state satisfying the precondition results in a state satisfying the postcondition. Hoare logic has been extended to support e.g. aliasing, concurrency, and so forth. But a certain limitation is often left untackled. Indeed, the pre- and postcondition of a Hoare triple typically constrain the state of a program by only looking at the initial and final state of memory. This makes it possible to prove e.g. that a quicksort implementation sorts properly, but it does not state that this result is e.g. printed on the screen. For the user of a program, the proofs about the state of memory of a program are useless if the user never sees the result on his screen. In the end, the program is supposed to correctly perform input/output, a problem typically left untouched. However, this is not the challenge itself. The interesting part of the challenge lays in the side constraints such as compositionality and modularity:

Modularity A programmer of a library typically does not consider all possible other libraries that might exist. Still, a programmer of an application can use multiple libraries in his program, even though these libraries do not know of each other's existence. Similarly, we want to write specifications of a library without keeping in mind existence of other libraries.

Compositionality In regular software development, a programmer typically does not call the low-level system calls. Instead, he calls high-level libraries, which might be implemented in terms of other libraries, implemented on top of other libraries, and so on. This is the concept of compositionality. The verification approach for I/O should support programs written in a compositional manner. Furthermore, it should be possible to write the formal I/O specifications themselves in a compositional manner, i.e. in terms of other libraries' I/O actions instead of in terms of the low-level system calls.

Other Besides compositionality and modularity, the I/O verification should also

- be sound (i.e. not searching for bugs but proving absence of bugs) and static (i.e. not detecting errors at runtime, but proving that such errors will never occur)
- blend in well with existing verification techniques that solve other problems like aliasing (or solve them itself)

- support non-deterministic behaviour (e.g. operations can fail, or return unspecified values, like reading user input)
- support underspecified specifications (e.g. the specifications describe two possibilities and the implementor can choose freely.
- support arguments for operations (e.g. when writing to a file, the content and the filename are arguments that should be part of the specifications)
- support concurrency
- support unspecified ordering of operations. If the order is unimportant, the specification should not fix them such that the implementor can choose freely.
- support specifying ordering of operations, also if the operations are specified and implemented by independent teams. e.g. it might be necessary that the put-shield-on operation happens before the start-explosion operation.
- support non-terminating programs: a non-terminating program can still only do the allowed I/O operations in the allowed order.
- support terminating programs: a program is only allowed to terminate if the I/O operations done are as specified and invoked.
- support operations that depend on the outcome of the previous operation, e.g. a specification like "read a number, and then print a number that is one higher than the read number".

This technical report provides an elegant way to perform sound modular compositional input/output verification based on separation logic and supports all the requirements explained above.

Section 2 describes the approach in a tutorial-style fashion, so it does not explain how it works but only how to use it. Section 3 formalizes the approach. Section 4 gives a proof outline of an example.

2 How to use the approach: a tutorial

We describe the input/output verification approach in a tutorial-style fashion. So this section does not explain how it works but only how to use it.

To get a better understanding, it might help to experiment. You can do so by using the VeriFast verifier, available for free from `http://people.cs.kuleuven.be/~bart.jacobs/verifast/`. The VeriFast distribution contains some examples of verifying I/O properties of programs. At the moment of writing, they are in the directory `examples/io`.

2.1 Warming up

When writing $\{P\}\, C\, \{Q\}$, we call P the precondition, Q the postcondition, and C a program (here, a function is also considered a program). A pre- and a postcondition both consist of an assertion. An assertion states some properties about the state of the program. Usually, the state of the program is the state of memory of the program. A precondition expresses some properties the program is allowed to assume to be true before execution; the postcondition are properties the program is supposed to make true after every possible execution starting from a state where the precondition is true. If this is the case, we say $\{P\}\, C\, \{Q\}$ is true.

Formal verification consists of proving that, starting from any state where the precondition is true, every execution of the program will result in a state for which the postcondition is true.

The pre- and postcondition together is what we call a contract. In this tutorial, we will learn only how to write contracts that specify the input-output behavior of programs. The formal semantics are out of scope of the tutorial and studied in section 3.

2.2 Time

We give a small input-output contract, namely one where no input-output is allowed.

```
{}
    // any implementation that doesn't crash or race and does no input-output
{}
```

Indeed, the empty pre- and postcondition do not allow input/output. Let's keep disallowing I/O while making the contract bigger:

```
{time(t1)}
  // any implementation that doesn't crash or race and does no input-output
{time(t1)}
```

The precondition `time(t_1)` can conceptually be considered as stating that the current time is t_1. For starters, you can think of a time as a point in time you are used to, e.g 8 AM or noon. Performing an input-output action would increase the time. The postcondition is the same as the precondition, so the program must make sure all constraints on the list of performed (and future) input-output actions still hold after execution. The only way to do this, is to not perform any input-output.

2.3 Actions

```
{time(t1) * print_char_io(t1, 'h', t2)}
  print_char('h');
{time(t2)}
```

The above contract states the only input-output behavior the program is allowed to perform, is writing the character 'h', and that, after execution of the program, the program must have done this input-output behavior.

`print_char_io(t_1, 'h', t_2)` states it is possible to go from time t_1 to time t_2 by writing the character 'h'. You can consider `print_char_io(t_1, 'h', t_2)` as a permission[1] to print 'h' provided the current time is t_1, and a promise that the time will increase to t_2 when doing so.

The postcondition says the time must become t_2, and because the only "permission" to obtain t_2 is by performing the input-output action of writing 'h', the program must have written 'h' to satisfy the postcondition. It cannot print 'h' twice, because it has no permission to print 'h' starting from a time t_2.

2.4 Choice

In the previous contract, there was only one permission provided to obtain t_2. In the following contract, two permissions are provided. As a result, the implementation can choose freely which of the two permissions to use.

```
{time(t1) * print_char_io(t1, 'h', t2) * print_char_io(t1, 'w', t2)}
  print_char('w');
{time(t2)}
```

So, the implementation can either print 'h' or 'w'. It can even choose at runtime what to do (e.g. using a (non I/O performing) random generator). It, however, can not do both.

2.5 Sequence

The following contract states the program must print "hi" and should be self-explaining by now.

```
{time(t1) * print_char_io(t1, 'h', t2) * print_char_io(t2, 'i', t3)}
  print_char('h');
  print_char('i')
{time(t3)}
```

Note that you can combine sequencing and choice, e.g.

[1] technically, it is not always entirely correct to do so, but it most often works to reason about it as such.

```
{time(t1) * print_char_io(t1, 'h', t2) * print_char_io(t2, 'i', t4)
 * print_char_io(t1, 'l', t3) * print_char_io(t3, 'o', t4)
 * print_char_io(t4, '!', t5)}
  // ... (implementation here)
{time(t5)}
```

The above program prints either "hi!" or "lo!".

2.6 Defining actions

So far, we treated `print_char_io` as an action coming from nowhere. You can define your own action in terms of other actions:

```
predicate print_string_io(t1, str, t2) =
  if str = nil then
    t1 = t2
  else (
    print_char_io(t1, head(str), t_between)
    * print_string_io(t_between, tail(str), t2)
  )
```

`print_string_io(t1, ``hi!'', t2)` can conceptually be considered a "shorthand" to

```
print_char_io(t1, 'h', t_between0)
* print_char_io(t_between0, 'i', t_between1)
* print_char_io(t_between1, '!', t2)
```

Technically, it's not really a shorthand but a predicate in the sense of [1]. When using a proof checker, one can write predicates without understanding the underlying theory because errors will be spotted by the proof checker.

Now we can finally write a clean hello world:

```
function print_string(str) =
{time(t1) * print_string_io(t1, str, t2)}
  while str != nil
    print_char(head(str));
    str := tail(str)
{time(t2)}

function helloworld() =
{time(t1) * print_string_io(t1, ``hello world!'', t2)}
  print_string(``hello world!'')
{time(t2)}
```

Of course, the implementation (and contract) of `print_string` is usually not considered part of the hello-world program, but part of a (standard) library, just like C's `helloworld` typically does not contain the implementation of `printf`.

Note that we sneaked in compositionality: we defined an I/O action `print_string_io` in terms of lower-level I/O actions (here `print_char_io`). You can now also define higher-level actions in terms of `print_string_io`.

2.7 Interleaved actions

Quite often, the order in which things happen matters. For example, you might want to put on the goggles before turning on the laser beam. But sometimes order does not matter, and in these cases it would be annoying if the specifications restrict the order in which the implementor can write his program. Sometimes the specification should leave space for the implementor by not being too restrictive.

An example: consider Unix' `cat`, a small program that just writes what it reads. The following contract would be annoying (`read_string_io` is similar to `print_string_io` and reads until e.g. end of file).

```
{time(t1) * read_string_io(t1, str, t2) * print_string_io(t2, str, t3)}
  // ...
{time(t3)}
```

Indeed: the programmer's implementation would be forced to first read everything, and then write everything. This would be impractical, certainly in case of limited memory.

Let's try again:

```
predicate readwrite_io(t1, str, t2) =
  read_char_io(t1, c, t_between0)
  * if c < 0 then (
      t2 = t_between0
      * str = nil
    ) else (
      c == head(str)
      * print_char_io(t1, head(str), t_between1)
      * readwrite_io(t_between1, tail(str), t2)
    )
{time(t1) * readwrite_io(t1, str, t2)}
  // ...
{time(t2)}
```

This would certainly solve the memory problem. But it introduces another one. What if the implementor wants to keep a buffer? For example, he might want to read 10 bytes, and then write 10 bytes, then read 10, and so forth. The contract disallows this. If we want to disallow this, we're ready. But if we want to allow the programmer to use any buffer size he wants, we can use a feature called split-join.

This is a solution:

```
{time(t1)
 * split(t1, t2, t3)
 * read_string_io(t2, str, t4)
 * print_string_io(t3, str, t5)
 * join(t4, t5, t6)
}
  // ...
{time(t6)}
```

Here, the implementor can interleave reads and writes as much or as little as he wants. Technically, he is even allowed to write everything before reading everything, if he is able to prove that in every execution the written output will be the same as the read input (which he will not be able to do).

You should now be able to tell the difference with:

```
{time(t1) * read_string_io(t1, str, t2) * print_string_io(t1, str, t2)}
  // ...
{time(t2)}
```

(Spoiler: with split you must both read and print, but the order between reading and printing is unspecified. Without split, you must or read or print, but not both).

We're not ready yet. We saw that the precondition of `print_string` is

```
  time(t1) * print_string_io(t1, str, t2)
```

So, in order to call this function, one must have a "time" t_1 that is equal to the first argument of `print_string_io`. Which is, right after the precondition of `cat`, not the case.

One can obtain `time(t2) * time(t3)` out of `time(t1) * split(t1, t2, t3)`. Hence the name "split": it splits time. Similarly, one can obtain `time(t3)` out of `join(t1, t2, t3) * time(t1) * time(t2)`. Join joins times together.

Note that without split-join, the approach would be completely unusable in practice.

2.8 Time revisited

Earlier, we said you can consider a time like t_1 as a point in the real-world time, like 8 AM or noon. This might be a confusing way to reason since time is not always before or after another time. Indeed, when splitting time, we obtain two times which do not have much relative ordering.

2.9 Other properties

The above part of the tutorial might give a misleading impression on the expressiveness of the approach. Since the approach blends in with separation logic-based verification, we have all the expressiveness of separation logic-based verification.

For example, if we want to express "No single program shall ever write to a file without opening it", we can simply do this by putting a permission in the postcondition of the function to open a file that expresses that the file has been opened, and require this permission in the precondition of the function to read from files. This was already possible before and the approach presented in this technical report is compatible with it.

Also, the approach can be combined with concurrency. In case two threads can do actions where there is no locking required, one can simply use split-join to allow any interleaving of these actions. If locking is required, the permissions to perform the action can be included among the permissions protected by the lock (the invariant of the lock).

3 Formalisation

We present a simple programming language and a verification approach.

$v \in \text{VarNames}$, $v^l \in \text{ListVarNames}$, $n, r \in \mathbb{Z}$, $bio \in \text{BioNames}$, $l \in \text{Lists}$, $f \in \text{FuncNames}$

$l ::= \textbf{nil} \mid n :: l$

$e ::= n \mid v \mid e + e \mid e - e \mid \text{head}(e^l)$

$e^l ::= l \mid v^l \mid e^l +\!\!+ e^l \mid \text{tail}(e^l)$

$b ::= \text{true} \mid \neg b \mid e = e \mid b \wedge b \mid e < e \mid e^l = e^l$

$c ::= \textbf{skip} \mid v := e \mid c \,;\, c \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid \textbf{while } b\ c \mid v := bio(\overline{e}) \mid v := f((\overline{e}), (\overline{e^l}))$

The language is standard except it supports doing Basic Input Output actions (BIOs). A BIO can be thought of as a system call, but for readability we use names ($bio \in \text{BioNames}$) as their identifiers instead of numbers. The arguments of a BIO can be considered as data written to the outside world, while the return-value can be considered as data that is read from the outside world. This way, a BIO allows doing both input and output.

We define Commands as the set of commands creatable by the grammar symbol "c" and quantify over it with c. Stores $= \{s_v \cup s_l \mid s_v \in \text{VarNames} \rightharpoonup \mathbb{Z} \wedge s_l \in \text{ListVarNames} \rightharpoonup \text{Lists}\}$, quantified over by s.

We assume a set FuncDefs $\subset \{(f, (\overline{v}), (\overline{v^l}), c) \mid f \in \text{FuncNames} \wedge \overline{v} \in \overline{\text{VarNames}} \wedge \overline{v^l} \in \overline{\text{ListVarNames}} \wedge c \in \text{Commands} \wedge \text{mod}(c) \cap (\overline{v} \cup \overline{v^l}) = \emptyset\}$. Here, $\text{mod}(c)$ returns the set of variables that command c writes to. FuncDefs represents the functions of the program under consideration. Note that we disallow functions for which the body assigns to a parameter of the functions. We also disallow overlap in parameter names.

Evaluation of expressions

$[\![n]\!]_s$	$= n$
$[\![e_1 + e_2]\!]_s$	$= [\![e_1]\!]_s + [\![e_2]\!]_s$
$[\![e_1 - e_2]\!]_s$	$= [\![e_1]\!]_s - [\![e_2]\!]_s$
$[\![v]\!]_s$	$= s(v)$ if v defined in s, otherwise 0.
$[\![\text{head}(e^l)]\!]_s$	$= \text{head}([\![e^l]\!]^l_s)$.
$[\![l]\!]^l_s$	$= l$
$[\![v^l]\!]^l_s$	$= s(v^l)$ if v^l defined in s, otherwise nil.
$[\![e^l_1 +\!+ e^l_2]\!]^l_s$	$= [\![e^l_1]\!]^l_s +\!+ [\![e^l_1]\!]^l_s$.
$[\![\text{tail}(e^l)]\!]^l_s$	$= \text{tail}([\![e^l]\!]^l_s)$.
$[\![\text{true}]\!]^b_s$	$= \text{true}$
$[\![\neg b]\!]^b_s$	$= \neg[\![b]\!]^b_s$
$[\![b_1 \wedge b_2]\!]^b_s$	$= [\![b_1]\!]^b_s \wedge [\![b_2]\!]^b_s$
$[\![e_1 = e_2]\!]^b_s$	$= ([\![e_1]\!]_s = [\![e_2]\!]_s)$
$[\![e_1 < e_2]\!]^b_s$	$= [\![e_1]\!]_s < [\![e_2]\!]_s$
$[\![e^l_1 = e^l_2]\!]^b_s$	$= ([\![e^l_1]\!]_s = [\![e^l_2]\!]_s)$

Notation For a (partial) function f, $f[a := b]$ is the (partial) function $\{(x, y) | (x \neq a \wedge x \in dom(f) \wedge y = f(x)) \vee (x = a \wedge y = b)\}$.

For lists, we use the infix functions $+\!+$ for concatenation and :: for cons. We write the empty list as nil.

We frequently notate lists with overline, e.g. $\overline{e}$ denotes a (implicitly quantified) list of expressions. We leave the technical parts implicit, e.g. when two lists are expected to have the same length. Sometimes we use such a list as a set. We abbreviate $\overline{[\![e]\!]}$ as $[\![\overline{e}]\!]$. We overline sets to obtain the set of lists of elements of this set, e.g. $\overline{\text{VarNames}}$ denotes the set of lists of variable names.

For an assertion P, $P[e/v]$ is the formula obtained by replacing all free occurrences of the variable v with e. We write multiple such replacements as $P[\overline{e}/\overline{v}]$. We also use this notation for replacing logical variables, logical expressions, etc.

For multisets A and B, $A - B$ denotes the multiset obtained by removing the occurrences of B in A, e.g. $\{\!\{1, 1, 1, 2, 2, 3\}\!\} - \{\!\{1, 2, 2\}\!\} = \{\!\{1, 1, 3\}\!\}$. $A + B$ yields the multiset such that for every x, the number of occurrences of x in $A + B$ (which can be zero) equals the number of occurrences of x in A plus the number of occurrences of x in B.

Step semantics We define Traces as the set of lists over the set $\{bio(\overline{n}) \mid bio \in \text{BioNames} \wedge \overline{n} \in \bigcup_{m>0} \mathbb{Z}^m\}$. An element of the list, $bio(\overline{n}, r)$, expresses the BIO bio has happened with arguments $\overline{n}$ and return value r. The order of the items in the list expresses the order in time in which they happened. We quantify over Traces with τ.

Continuations = Commands $\cup$ {**partial**, **done**}, quantified over by κ.

$$\textsc{Assign}\ \frac{}{s, v := e \Downarrow s[v := [\![e]\!]_s], \text{nil}, \textbf{done}} \qquad \textsc{AssignList}\ \frac{}{s, v^l := e^l \Downarrow s[v^l := [\![e^l]\!]^l_s], \text{nil}, \textbf{done}}$$

$$\textsc{IfThen}\ \frac{[\![b]\!]^b_s = \text{true} \qquad s, c_{then} \Downarrow s', \tau, \kappa}{s, \textbf{if}\ b\ \textbf{then}\ c_{then}\ \textbf{else}\ c_{else} \Downarrow s', \tau, \kappa} \qquad \textsc{IfElse}\ \frac{[\![b]\!]^b_s = \text{false} \qquad s, c_{else} \Downarrow s', \tau, \kappa}{s, \textbf{if}\ b\ \textbf{then}\ c_{then}\ \textbf{else}\ c_{else} \Downarrow s', \tau, \kappa}$$

$$\textsc{WhileIn}\ \frac{[\![b]\!]^b_s = \text{true} \qquad s, c\,; \textbf{while}\ b\ c \Downarrow s', \tau, \kappa}{s, \textbf{while}\ b\ c \Downarrow s', \tau, \kappa} \qquad \textsc{WhileOut}\ \frac{[\![b]\!]^b_s = \text{false}}{s, \textbf{while}\ b\ c \Downarrow s, \text{nil}, \textbf{done}} \qquad \textsc{Skip}\ \frac{}{s, \textbf{skip} \Downarrow s, \text{nil}, \textbf{done}}$$

$$\textsc{Seq2}\ \frac{s_1, c_1 \Downarrow s_2, \tau_2, \textbf{done} \qquad s_2, c_2 \Downarrow s_3, \tau_3, \kappa}{s_1, c_1; c_2 \Downarrow s_3, \tau_2 +\!\!+ \tau_3, \kappa} \qquad \textsc{Seq}\ \frac{s_1, c_1 \Downarrow s_2, \tau_2, \textbf{partial}}{s_1, c_1; c_2 \Downarrow s_2, \tau_2, \textbf{partial}} \qquad \textsc{Empty}\ \frac{}{s, c \Downarrow s, \text{nil}, \textbf{partial}}$$

$$\textsc{Bio}\ \frac{i \in \mathbb{Z}}{s, v := bio(\overline{e}) \Downarrow s[v := i], bio([\![\overline{e}]\!]_s, i) :: \text{nil}, \textbf{done}}$$

$$\textsc{FuncCall}\ \frac{\emptyset[\overline{v}, \overline{v^l} := [\![\overline{e}]\!]_s, [\![\overline{e^l}]\!]_s], c \Downarrow s_f, \tau, \kappa \qquad (f, (\overline{v}), (\overline{v^l}), c) \in \text{FuncDefs}}{s, v := f((\overline{e}), (\overline{e^l})) \Downarrow s[v := [\![\text{result}]\!]_{s_f}], \tau, \kappa}$$

Note that the step semantics do not only support terminating runs, but also partial runs. This allows us to verify the input/output behavior of programs that do not terminate.

Assertions We define Times as the set of times and quantify over it with t. Intuitively, a time t can be considered as a name for a timestamp with unknown value, or as a set of constraints on a timestamp.

We define Chunks as $\{\text{time}(t) \mid t \in \text{Times}\} \cup \{bio(t_1, \overline{n}, r, t_2) \mid bio \in \text{BioNames} \wedge \overline{n} \in \bigcup_{m \geq 0} \mathbb{Z}^m \wedge r \in \mathbb{Z} \wedge t_1, t_2 \in \text{Times}\}$ and Heaps as Chunks $\rightharpoonup \mathbb{N}$. We will use the heap for permissions (rather than memory footprint) and treat it like separation logic [2].

$V^n \in \text{IntegerLogicalVarNames}$, $V^t \in \text{TimeLogicalVarNames}$, $V^l \in \text{ListLogicalVarNames}$,
$p \in \text{PredNames}$, $V \in \text{IntegerLogicalVarNames} \cup \text{TimeLogicalVarNames} \cup \text{ListLogicalVarNames}$.

$$E^n ::= n \mid v \mid V^n \mid E^n + E^n \mid E^n - E^n \mid \text{head}(E^l)$$
$$E^t ::= t \mid V^t$$
$$E^l ::= l \mid v^l \mid V^l \mid E^l +\!\!+ E^l \mid \text{tail}(E^l)$$
$$B ::= \text{true} \mid E^n = E^n \mid E^t = E^t \mid E^l = E^l \mid \neg B \mid E^n < E^n$$
$$P, Q, R ::= B \mid \textbf{emp} \mid P \star P \mid bio(E^t, \overline{E^n}, E^n, E^t) \mid \textbf{split}(E^t, E^t, E^t) \mid \textbf{join}(E^t, E^t, E^t) \mid \textbf{time}(E^t) \mid p(\overline{E^n}, \overline{E^l}, \overline{E^t}) \mid P \vee P \mid \exists V.\ P$$

We assume all sets of variable names (IntegerLogicalVarNames, VarNames, ...) to be disjoint.

We use P, Q and R to quantify over the assertions formed by grammar symbol P.

We define Interpretations $= \{i_n \cup i_t \cup i_l \mid i_n \in \text{IntegerLogicalVarNames} \rightharpoonup \mathbb{Z} \wedge i_t \in \text{TimeLogicalVarNames} \rightharpoonup \text{Times} \wedge i_l \in \text{ListLogicalVarNames} \rightharpoonup \text{Lists}\}$ and quantify over it with i. An interpretation maps logical variables to values.

emp denotes the heap is empty and $\star$ is the separating conjunction [2]. The existential quantors for integers, times, and lists are necessary such that they can be used in definitions of inductive predicates.

We use predicates based on and similar to [1]. A predicate can be considered as a named assertion, but the assertion can contain the predicate name to allow recursion. A predicate definition consists of a predicate name, a number of integer argument names, a number of list argument names, a number of time argument names (all argument names distinct), and an assertion. We disallow mutual recursion. We write PredDefs for the set of predicate definitions for the program under consideration. This is

the set of definitions for (the contracts of) a particular program, not the set of all possible definitions. PredDefs $\subset$ PredNames $\times$ $(\overline{V^n})$ $\times$ $(\overline{V^l})$ $\times$ $(\overline{V^t})$ $\times$ P.

An assertion $bio(t_1, \overline{e}, e_r, t_2)$ expresses the permission to perform the BIO bio with arguments $\overline{e}$ at timestamp t_1 and includes the prediction that performing that BIO at the given time will return e_r and finish at time t_2.

The **split** and **join** assertions allow interleaved actions and choosing freely which actions happen first. We refer to the tutorial for better insight in the expressiveness of these assertions.

In the tutorial, we wrote assertions of the form **if** b **then** P **else** Q. This is shorthand notation for $(b \star P) \vee (\neg b \star Q)$.

Evaluation of assertion expressions

$$
\begin{array}{ll}
[\![n]\!]_{s,i} & = n \\
[\![v]\!]_{s,i} & = s(v) \text{ if } v \text{ defined in } s, \text{ otherwise 0.} \\
[\![V^n]\!]_{s,i} & = i(V^n) \text{ if } V^n \text{ defined in } i, \text{ otherwise 0.} \\
[\![V^t]\!]_{s,i} & = i(V^t) \text{ if } V^t \text{ defined in } i, \text{ otherwise undefined.} \\
[\![E_1^n + E_2^n]\!]_{s,i} & = [\![E_1^n]\!]_{s,i} + [\![E_2^n]\!]_{s,i} \\
[\![E_1^n - E_2^n]\!]_{s,i} & = [\![E_1^n]\!]_{s,i} - [\![E_2^n]\!]_{s,i} \\
[\![l]\!]_{s,i} & = l. \\
[\![v^l]\!]_{s,i} & = s(v^l) \text{ if } v^l \text{ defined in } s, \text{ otherwise nil.} \\
[\![V^l]\!]_{s,i} & = i(V^l) \text{ if } V^l \text{ defined in } i, \text{ otherwise nil.} \\
[\![E_1^l +\!\!+ E_2^l]\!]_{s,i} & = [\![E_1^l]\!]_{s,i} +\!\!+ [\![E_2^l]\!]_{s,i}. \\
[\![\text{head}(E^l)]\!]_{s,i} & = \text{head}([\![E^l]\!]_{s,i}) \\
[\![\text{tail}(E^l)]\!]_{s,i} & = \text{tail}([\![E^l]\!]_{s,i}) \\
[\![E_1^n = E_2^n]\!]_{s,i} & = ([\![E_1^n]\!]_{s,i} = [\![E_2^n]\!]_{s,i}) \\
[\![E_1^t = E_2^t]\!]_{s,i} & = ([\![E_1^t]\!]_{s,i} = [\![E_2^t]\!]_{s,i}) \\
[\![E_1^l = E_2^l]\!]_{s,i} & = ([\![E_1^l]\!]_{s,i} = [\![E_2^l]\!]_{s,i}) \\
[\![\neg B]\!]_{s,i} & = \neg [\![B]\!]_{s,i} \\
[\![E_1^n < E_2^n]\!]_{s,i} & = [\![E_1^n]\!]_{s,i} < [\![E_2^n]\!]_{s,i} \\
[\![\text{true}]\!]_{s,i} & = \text{true}
\end{array}
$$

Satisfaction relation of formulae For predicates, we assume a context I which we will define later. $I \subseteq$ PredNames $\times$ $\overline{\mathbb{Z}}$ $\times$ $\overline{\text{Lists}}$ $\times$ $\overline{\text{Times}}$ $\times$ Heaps. I expresses, for a given predicate name and argument values, the heap chunks a predicate assertion covers.

$$
\begin{array}{lcl}
I, s, h, i \models B & \iff & [\![B]\!]_{s,i} = true \wedge h = \emptyset \\
I, s, h, i \models bio(t_1, \overline{E^n}, E_r^n, t_2) & \iff & h = \{\!\{bio(t_1, [\![\overline{E^n}, E_r^n]\!]_{s,i}, t_2)\}\!\} \\
I, s, h, i \models \mathbf{join}(t_1, t_2, t_3) & \iff & h = \{\!\{\mathbf{join}(t_1, t_2, t_3)\}\!\} \\
I, s, h, i \models \mathbf{split}(t_1, t_2, t_3) & \iff & h = \{\!\{\mathbf{split}(t_1, t_2, t_3)\}\!\} \\
I, s, h, i \models \mathbf{time}(t) & \iff & h = \{\!\{\mathbf{time}(t)\}\!\} \\
I, s, h, i \models \mathbf{emp} & \iff & h = \{\!\{\}\!\} \\
I, s, h, i \models P \star Q & \iff & \exists h_1, h_2 \,.\; h_1 + h_2 = h \;\wedge\; I, s, h_1 \models P \;\wedge\; I, s, h_2 \models Q \\
I, s, h, i \models p(\overline{E^n}, \overline{E^l}, \overline{t}) & \iff & (p, ([\![\overline{E^n}]\!]_{s,i}), ([\![\overline{E^l}]\!]), (\overline{t}), h) \in I \\
I, s, h, i \models P \vee Q & \iff & (I, s, h, i \models P) \vee (I, s, h, i \models Q) \\
I, s, h, i \models \exists V^n.\; P & \iff & \exists n \in \mathbb{Z}.\; I, s, h, i \models P[V^n := n] \\
I, s, h, i \models \exists V^l.\; P & \iff & \exists l.\; I, s, h, i \models P[V^l := l] \\
I, s, h, i \models \exists V^t.\; P & \iff & \exists t.\; I, s, h, i \models P[V^t := t]
\end{array}
$$

Note that the satisfaction relation is undefined for formulae with unbound variables of V^n, V^l and V^l.

We assumed I so far but did not define it yet.

$$
\begin{aligned}
&I_0 = \emptyset \\
&I_{n+1} = \{\ (p, (\overline{n}), (\overline{l}), (\overline{t}), h) \mid \exists \overline{V^n}, \overline{V^l}, \overline{V^t}, P.\ (p, (\overline{V^n}), (\overline{V^l}), (\overline{V^t}), P) \in \text{PredDefs} \wedge \\
&\qquad I_n, \emptyset, h, \emptyset \models P[\overline{n}, \overline{l}, \overline{t}/\overline{V^n}, \overline{V^l}, \overline{V^t}]\ \}
\end{aligned}
$$

We define I as $\bigcup_{n \in \mathbb{N}} I_n$.

Validity of Hoare triples Intuitively, the Hoare triple $\{P\}\, c\, \{Q\}$ expresses that the program c satisfies the contract with precondition P and postcondition Q.

For examples of Hoare triples and their meaning, we refer to the tutorial (section 2).

Note that a program that satisfies the contract cannot perform any other I/O operations, cannot do them in another order, cannot do them more than once, etc.

We define a relation traces $\subset$ (Heaps $\times$ Traces $\times$ Heaps). $\{h_1\}\, \tau\, \{h_2\}$ denotes $(h_1, \tau, h_2) \in$ traces. It expresses that τ is allowed by h_1. An implementation is thus allowed to produce the trace τ. Thus, a heap is mapped to a set of allowed traces. Remember that an element in the heap can make a prediction about the world, e.g. $bio(t_1, \overline{n}, r, t_2)$ predicts performing the BIO bio with arguments $\overline{n}$ (at a certain time) will have return-value r. Thus, a heap can contradict itself, e.g. $\{\!|\mathbf{time}(\mathrm{t}_1), \mathrm{some\_bio}(\mathrm{t}_1, 1, 2, \mathrm{t}_2), \mathrm{some\_bio}(\mathrm{t}_1, 1, 3, \mathrm{t}_2)|\!\}$ contradicts itself because it says performing the BIO some_bio (at time t_1 with argument 1) will return 2 and will return 3. After a BIO that violates a prediction, all further BIOs are allowed.

$$
\textsc{TraceBio}\ \frac{r = r' \Rightarrow \{h + \{\!|\mathbf{time}(t_2)|\!\}\}\, \tau\, \{h'\}}{\{h + \{\!|\mathbf{time}(t_1), bio(t_1, \overline{n}, r, t_2)|\!\}\}\, bio(\overline{n}, r') :: \tau\, \{h'\}} \qquad \textsc{TraceNil}\ \frac{}{\{h\}\, \mathrm{nil}\, \{h\}}
$$

$$
\textsc{TraceSplit}\ \frac{\{h + \{\!|\mathbf{time}(t_2), \mathbf{time}(t_3)|\!\}\}\, \tau\, \{h'\}}{\{h + \{\!|\mathbf{time}(t_1), \mathbf{split}(t_1, t_2, t_3)|\!\}\}\, \tau\, \{h'\}} \qquad \textsc{TraceJoin}\ \frac{\{h + \{\!|\mathbf{time}(t_3)|\!\}\}\, \tau\, \{h'\}}{\{h + \{\!|\mathbf{time}(t_1), \mathbf{time}(t_2), \mathbf{join}(t_1, t_2, t_3)|\!\}\}\, \tau\, \{h'\}}
$$

We define validity of a Hoare triple. Intuitively, it expresses that any execution starting from a state (a store, a heap and an interpretation) that satisfies the precondition, results in a trace that is allowed by the heap. In case the execution is a finished one (i.e. the program terminated), the state at termination must satisfy the postcondition.

$$
\begin{aligned}
&\forall P, c, Q.\ \models \{P\}\, c\, \{Q\} \iff \\
&\quad \forall s, h, i.\ I, s, h, i \models P \Rightarrow \\
&\qquad \forall s', \tau', \kappa'.\ s, c \Downarrow s', \tau', \kappa' \Rightarrow \\
&\qquad\quad \exists h'.\ \{h\}\, \tau'\, \{h'\} \wedge \\
&\qquad\qquad (\kappa' = \mathbf{done} \Rightarrow I, s', h', i \models Q)
\end{aligned}
$$

In case you expected a universal quantifier for h', note that the concrete execution does not use a heap. If it would use a heap, h' would be introduced in the universal quantification together with s', τ' and κ'.

Proof rules

$$\textsc{Assignment}\ \frac{}{\{P[e/v]\}\, v := e\, \{P\}} \qquad \textsc{AssignmentList}\ \frac{}{\{P[e^l/v^l]\}\, v^l := e^l\, \{P\}} \qquad \textsc{Composition}\ \frac{\{P_1\}\, c_1\, \{P_2\} \qquad \{P_2\}\, c_2\, \{P_3\}}{\{P_1\}\, c_1; c_2\, \{P_3\}}$$

$$\textsc{Consequence}\ \frac{P_1 \Rightarrow P_2 \qquad \{P_2\}\, c\, \{P_3\} \qquad P_3 \Rightarrow P_4}{\{P_1\}\, c\, \{P_4\}} \qquad \textsc{While}\ \frac{\{b \star P\}\, c\, \{P\}}{\{P\}\, \textbf{while}\ b\ c\, \{\neg b \star P\}} \qquad \textsc{Skip}\ \frac{}{\{P\}\, \textbf{skip}\, \{P\}}$$

$$\textsc{If}\ \frac{\{P \star b\}\, c_{\text{then}}\, \{Q\} \qquad \{P \star \neg b\}\, c_{\text{else}}\, \{Q\}}{\{P\}\, \textbf{if}\ b\ \textbf{then}\ c_{\text{then}}\ \textbf{else}\ c_{\text{else}}\, \{Q\}} \qquad \textsc{Disjunction}\ \frac{\{P_1\} c \{Q\} \qquad \{P_2\} c \{Q\}}{\{P_1 \vee P_2\}\, c\, \{Q\}}$$

$$\textsc{Bio}\ \frac{}{\{P[e_r/v] \star bio(t_1, \overline{e}, e_r, t_2) \star \textbf{time}(t_1)\}\, v := bio(\overline{e})\, \{P \star \textbf{time}(t_2)\}}$$

$$\textsc{Split}\ \frac{\{\textbf{time}(t_2) \star \textbf{time}(t_3)\}\, c\, \{\textbf{time}(t_4)\}}{\{\textbf{time}(t_1) \star \textbf{split}(t_1, t_2, t_3)\}\, c\, \{\textbf{time}(t_4)\}} \qquad \textsc{Join}\ \frac{\{\textbf{time}(t_3)\}\, c\, \{\textbf{time}(t_4)\}}{\{\textbf{time}(t_1) \star \textbf{time}(t_2) \star \textbf{join}(t_1, t_2, t_3)\}\, c\, \{\textbf{time}(t_4)\}}$$

$$\textsc{Frame}\ \frac{\{P\}\, c\, \{Q\} \qquad \text{fv}(R) \cap \text{mod}(c) = \emptyset}{\{P \star R\}\, c\, \{Q \star R\}}$$

$$\textsc{FuncCall}\ \frac{\begin{array}{c}\{P\}\, c\, \{Q\} \qquad \text{fv}(P) \subseteq (\overline{v} \cup \overline{v^l}) \\ \text{fv}(Q) \subseteq (\overline{v} \cup \overline{v^l} \cup \{\text{result}\}) \qquad \{\text{result}, v\} \cap \text{fv}(\overline{e} \cup \overline{e^l}) = \emptyset \qquad (f, (\overline{v}), (\overline{v^l}), c) \in \text{FuncDefs}\end{array}}{\{P[\overline{e}, \overline{e^l}/\overline{v}, \overline{v^l}]\}\, v := f((\overline{e}), (\overline{e^l}))\, \{Q[\overline{e}, \overline{e^l}, v/\overline{v}, \overline{v^l}, \text{result}]\}}$$

$$\textsc{Exists}\ \frac{\{P\}\, c\, \{Q\}}{\{\exists V.\ P\}\, c\, \{\exists V.\ Q\}} \qquad \textsc{Substitution}\ \frac{\{P\}\, c\, \{Q\} \qquad \text{fv}(\overline{E^n}, \overline{E^l}) \cap \text{mod}(c) = \emptyset}{\{P[\overline{E^n}, \overline{E^l}, \overline{E^t}/\overline{V^n}, \overline{V^l}, \overline{V^t}]\}\, c\, \{Q[\overline{E^n}, \overline{E^l}, \overline{E^t}/\overline{V^n}, \overline{V^l}, \overline{V^t}]\}}$$

Here, fv of an expression or formula returns the set of free variables of the expression or formula. The frame rule is studied in [2].

We say a Hoare triple $\{P\}\, c\, \{Q\}$ is derivable, written $\vdash \{P\}\, c\, \{Q\}$, if it can be derived using the above proof rules.

Theorem 3.1 (Soundness). $\forall P, c, Q.\ \vdash \{P\}\, c\, \{Q\} \Rightarrow \models \{P\}\, c\, \{Q\}$.

A proof is future work.

4 Example

Consider the example below. The contract specifies it reads from standard input (until end-of-file), and writes what it reads to both standard output and standard error. It is written in a compositional manner: an action `tee_out` represents the action of writing to both standard output and standard error. The action that represents the whole program is built upon the `tee_out` action. The specifications allow a read-buffer of any size. The implementation chooses a read-buffer of size 2. This example is also shipped with VeriFast. At the moment of writing you can find it in `examples/io/tee/tee_buffered.c` in the VeriFast release ZIP or tar.

```
predicate tee_out(t1, c, t2) =
  split(t1, t_stdout1, t_stderr1)
  * print_char(t_stdout1, c, t_stdout2)
  * print_char_stderr_io(t_stderr1, c, t_stderr2)
```

```
  * join(t_stdout2, t_stderr2, t2)

function tee_out(c) =
{
  time(t1) * tee_out(t1, c, t2)
}
  tmp := write_char(c);
  tmp := write_char_stderr(c)
{
  time(t2)
}

predicate reads(t1, contents, t2) =
  read(t1, c, t_read)
  * if c >= 0 then
      length(contents) > 0 * c == head(contents)
      * reads(t_read, tail(contents), t2)
    else
      t2 == t_read * contents == nil

predicate tee_out_string(t1, contents, t2) =
  if contents == nil then
    t2 == t1
  else
    tee_out(t1, head(contents), t_out)
    * tee_out_string(t_out, tail(contents), t2)

predicate tee(t1, text, t2) =
  split(t1, t_read1, t_write1)
  * reads(t_read1, text, t_read2)
  * tee_out_string(t_write1, text, t_write2)
  * join(t_read2, t_write2, t2)

function main() =
{
  time(t1) * tee(t1, text, t2)
}
  c2 := 0;
  while (c2 >= 0) (
    c1 := read_char();
    if (c1 >= 0) (
      c2 := read_char();
      tmp := tee_out(c1);
      if (c2 >= 0)(
        tmp := tee_out(c2)
      )
    ) else (
      c2 := -1
    )
  )
{
  time(t2)
}
```

We give a proof outline. We start with the function `tee_out`.

```
{time(t1) * tee_out(t1, c, t2)}
```

```
{time(t_stdout1) * time(t_stderr1))
  * print_char_io(t_stdout1, c, t_stdout2)
  * print_char_stderr_io(t_stderr1, c, t_stderr2)
  * join(t_stdout2, t_stderr2, t2)
}
write_char(c);
{time(t_stdout2) * time(t_stderr1))
  * print_char_stderr_io(t_stderr1, c, t_stderr2)
  * join(t_stdout2, t_stderr2, t2)
}
write_char_stderr(c)
{time(t_stdout2) * time(t_stderr2)) * join(t_stdout2, t_stderr2, t2)}
{time(t2)}
```

Before looking at the main function, we define a helper predicate:

```
predicate invariant(c2, t2) =
 if c2 >= 0 then
   time(t_r1) * reads(t_r1, text, t_r2)
   * time(t_w1) * tee_out_string(t_w1, text, t_w2)
   * join(t_r2, t_w2, t2)
 else
   time(t2)
```

Next, we give a proof outline for the main function:

```
{ time(t1) * tee(t1, text, t2) }
{ 0 = 0 * time(t1) * tee(t1, text, t2)}
c2 := 0;
{ invariant(c2, t2) }
while (c2 >= 0) (
  {
    c1' = c1' * time(t_r1) * read(t_r1, c1', t_rb1) *
    if c1' >= 0 then ( length(text') > 0 * c1' = head(text') * reads(t_rb1, tail(text'), t_r2) )
        else ( t_r2 = t_rb1 * text' = nil )
    * time(t_w1) * tee_out_string(t_w1, text', t_w2)
    * join(t_r2, t_w2, t2)
  }
  c1 := read_char();
  {
    time(t_rb1) *
    if c1 >= 0 then ( length(text') > 0 * c1 = head(text') * reads(t_rb1, tail(text'), t_r2) )
      else ( t_r2 = t_rb1 * text' = nil )
    * time(t_w1) * tee_out_string(t_w1, text', t_w2)
    * join(t_r2, t_w2, t2)
  }
  if (c1 >= 0) (
    {
      c2' = c2' * time(t_rb1) * read(t_rb1, c2', t_rb2)
      * length(text') > 0 * c1 = head(text')
      * if c2' >= 0 then ( length(tail(text')) > 0 * c2' = head(tail(text'))
          * reads(t_rb2, tail(tail(text')), t_r2) )
        else ( t_r2 = t_rb2 * tail(text') = nil )
      * time(t_w1) * tee_out_string(t_w1, text', t_w2)
      * join(t_r2, t_w2, t2)
    }
    c2 := read_char();
    {
```

```
    time(t_rb2) * length(text') > 0 * c1 = head(text')
    * if c2 >= 0 then ( length(tail(text')) > 0 * c2 = head(tail(text'))
       * reads(t_rb2, tail(tail(text')), t_r2) )
      else t_r2 = t_rb2 * tail(text') = nil
    * time(t_w1) * tee_out(t_w1, head(text'), t_wb1)
    * tee_out_string(t_wb1, tail(text'), tw2)
    * join(t_r2, t_w2, t2)
  }
  tmp := tee_out(c1);
  {
    time(t_rb2) * length(text') > 0 * c1 = head(text')
    * if c2 >= 0 then ( length(tail(text')) > 0 * c2 = head(tail(text'))
       * reads(t_rb2, tail(tail(text')), t_r2) )
      else ( t_r2 = t_rb2 * tail(text') = nil )
    * time(t_wb1) * tee_out_string(t_wb1, tail(text'), tw2)
    * join(t_r2, t_w2, t2)
  }
  if (c2 >= 0)(
    {
      * time(t_rb2) * reads(t_rb2, tail(tail(text')), t_r2)
      * length(text') > 1 * c1 = head(text') * c2 = head(tail(text'))
      * time(t_wb1) * tee_out(t_wb1, head(tail(text')), t_wb2)
      * tee_out_string(t_wb2, tail(tail(text')), t_w2)
      * join(t_r2, t_w2, t2)
    }
    tmp := tee_out(c2);
    {
      * time(t_rb2) * reads(t_rb2, tail(tail(text')), t_r2)
      * length(text') > 1 * c1 = head(text') * c2 = head(tail(text'))
      * time(t_wb2) * tee_out_string(t_wb2, tail(tail(text')), t_w2)
      * join(t_r2, t_w2, t2)
    }
  ) else (
    { c2 < 0 * time(t2) }
  )
  { invariant(c2, t2) }
) else (
  { -1 = -1 * time(t2) }
  c2 := -1;
  { invariant(c2, t2) }
)
{ invariant(c2, t2) }
)
{ time(t2) }
```


Acknowledgements

We would like to thank Amin Timany for many useful discussions. This work was partially funded by EU FP7 FET-Open project ADVENT under grant number 308830.

References

[1] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.

[2] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02*, pages 55–74, Washington, 2002. IEEE.