Sound, modular and compositional verification of the input/output behavior of programs

Willem Penninckx, Bart Jacobs, Frank Piessens

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium

Abstract. We present a sound verification approach for verifying input/output properties of programs. Our approach supports defining high-level I/O actions on top of low-level ones (compositionality), defining input/output actions without taking into account which other actions exist (modularity), and other features. As the key ingredient, we developed a separation logic over Petri nets. We also show how with the same specification style we can elegantly modularly verify "I/O-like" code that uses the Template Pattern. We have implemented our approach in the VeriFast verifier and applied it to a number of challenging examples.

1 Introduction

Many software verification approaches are based on Hoare logic. A Hoare triple [6] 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 (if it terminates) in a state satisfying the postcondition. Hoare logic has been extended to support various features, e.g. aliasing and concurrency. But a certain limitation is often left untackled. Indeed, the pre- and postcondition of a Hoare triple typically constrain the behavior 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 prevent that e.g. an incorrect result is printed on the screen. For the user of a program, the proofs about the state of memory of a program are useless if the result visible on the screen is incorrect. In the end, the performed input/output must be correct, a problem typically left untouched.

There are some conceptual differences between verifying memory state and verifying I/O behavior. One difference is that, when verifying memory state, we only care about the final state. Indeed, if the function that sorts has an intermediate state that looks like garbage, but then cleans up and still gives a correctly sorted output, we are happy. In contrast, when verifying input/output we do care about the intermediary state. If e.g. the calculator displays the wrong output on the screen and then the right output, this is a bad calculator, even though the final image displayed on the screen is correct.

Another difference is that termination is usually a desired property of programs not performing I/O, but often undesired for programs performing I/O. For example, a quicksort implementation should always terminate, but a text editor should not.

However, just verifying I/O properties is not the interesting challenge itself. The interesting challenges are 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, themselves implemented on top of yet 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 static, i.e. detecting errors at compile time, not at run time.
- be sound, i.e. not searching for most bugs, but proving absence of bugs.
- blend in well with existing verification techniques that solve other problems like aliasing
- support non-deterministic behavior (e.g. operations can fail, or return unspecified values, like reading user input)
- support impricise specifications (e.g. the specifications describe two possibilities and the implementor can choose freely). The number of possibilities can be large, e.g. "print a number less than 0".
- 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 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. For example, it might be necessary that the put-shield-on operation happens before the start-explosion operation.
- support both non-terminating and terminating programs: a non-terminating program can still only do the allowed I/O operations in the allowed order, and a terminating program is only allowed to terminate after it has performed all desired I/O operations.
- 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 paper proposes an elegant way to perform input/output verification based on separation logic. It supports all the requirements explained above. We consider soundness, compositionality, and modularity the more interesting properties.

This paper does not include an approach to prove liveness properties. For both non-terminating and terminating programs, the approach presented in this paper allows to prove that all performed I/O is correct and in the correct order, but for non-terminating programs it does not provide a way to prove that any I/O happens. For terminating programs, the approach allows to prove that the intended I/O has happened upon termination.

The remainder of this paper is organized as follows. Section 2 defines a basic programming language supporting I/O. Section 3 uses this language to explain the verification approach. Section 4 proves soundness of this approach. Section 5 gives some examples. Section 6 provides a quick look at what is different and common in verifying I/O behavior and memory state. Section 8 concludes and points out future work.

2 The programming language

We define a simple programming language that supports performing I/O.

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

$$e ::= n \mid v \mid e + e \mid e - e \mid \text{head}(e) \mid \text{nil} \mid e :: e \mid e +\!\!+ e \mid \text{tail}(e) \mid \text{true} \mid \neg e \mid e = e \mid e \wedge e \mid e < e$$

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

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 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 use simple mathematical functions for lists with their expected meaning, e.g. tail and distinct.

The language is standard except that 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 that is output to the outside world, while the return-value can be considered as data that is input 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 = VarNames $\rightharpoonup (\mathbb{Z} \cup \mathbb{Z}^*)$, quantified over by s. Here, $\mathbb{Z}^*$ denotes the set of lists of integers. The partial function Stores maps the program variables to their current value.

We assume a set FuncDefs $\subset \{(f, \overline{v}, c) \mid f \in \text{FuncNames} \wedge \overline{v} \in \text{VarNames}^* \wedge c \in \text{Commands} \wedge \text{mod}(c) \cap \overline{v} = \emptyset \wedge \text{distinct}(\overline{v})\}$. Here, mod($c$) returns the set of variables that command c writes to. FuncDefs represents the functions of the

$$\textsc{Assign}\ \frac{}{s, v := e \Downarrow s[v := [\![e]\!]_s], \text{nil}, \textbf{done}} \qquad \textsc{IfThen}\ \frac{[\![e]\!]_s = \text{true} \qquad s, c_{then} \Downarrow s', \tau, \kappa}{s, \textbf{if } e \textbf{ then } c_{then} \textbf{ else } c_{else} \Downarrow s', \tau, \kappa}$$

$$\textsc{IfElse}\ \frac{[\![e]\!]_s \neq \text{true} \qquad s, c_{else} \Downarrow s', \tau, \kappa}{s, \textbf{if } e \textbf{ then } c_{then} \textbf{ else } c_{else} \Downarrow s', \tau, \kappa} \qquad \textsc{WhileIn}\ \frac{[\![e]\!]_s = \text{true} \qquad s, c\,; \textbf{while } e \textbf{ do } c \Downarrow s', \tau, \kappa}{s, \textbf{while } e \textbf{ do } c \Downarrow s', \tau, \kappa}$$

$$\textsc{WhileOut}\ \frac{[\![e]\!]_s \neq \text{true}}{s, \textbf{while } e \textbf{ do } c \Downarrow s, \text{nil}, \textbf{done}} \qquad \textsc{Skip}\ \frac{}{s, \textbf{skip} \Downarrow s, \text{nil}, \textbf{done}} \qquad \textsc{Seq}\ \frac{s_1, c_1 \Downarrow s_2, \tau_2, \textbf{partial}}{s_1, c_1; c_2 \Downarrow s_2, \tau_2, \textbf{partial}}$$

$$\textsc{Seq2}\ \frac{s_1, c_1 \Downarrow s_2, \tau_1, \textbf{done} \qquad s_2, c_2 \Downarrow s_3, \tau_2, \kappa}{s_1, c_1; c_2 \Downarrow s_3, \tau_1 +\!\!+ \tau_2, \kappa} \qquad \textsc{Empty}\ \frac{}{s, c \Downarrow s, \text{nil}, \textbf{partial}}$$

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

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

Fig. 1: Step semantics

program under consideration. Note that we disallow functions for which the body assigns to a parameter of the function. We also disallow overlap in parameter names. For simplicity, we only consider functions and programs without (mutual) recursion.

For better readability, we use abbreviations with the expected meaning, e.g. $e_1 \neq e_2$ means $\neg(e_1 = e_2)$ and $f(\overline{e})$ means $v := f(\overline{e})$ for a fresh v.

Evaluation of the expression e using store s is written as $[\![e]\!]_s$. We write $\overline{[\![e]\!]}$ as $[\![\overline{e}]\!]$. Evaluation of the expressions consisting of a variable is defined as $[\![v]\!]_s = s(v)$ (if v defined in s, otherwise unspecified). Evaluation of the other expressions is defined as $[\![op(\overline{e})]\!]_s = op([\![\overline{e}]\!]_s)$ where op is an operator with zero arguments (for constants, e.g. true, 2, or nil) or more. For example, $[\![\text{tail}(e)]\!]_s = \text{tail}([\![e]\!]_s)$, $[\![e_1 + e_2]\!]_s = [\![e_1]\!]_s + [\![e_2]\!]_s$ and $[\![2]\!]_s = 2$. Expressions that are not well-typed evaluate to an unspecified value, e.g. head(0) can evaluate to nil and to true.

Step semantics We define Traces as the set of lists over the set $\{bio(\overline{n}, r) \mid bio \in \text{BioNames} \wedge \overline{n} \in \mathbb{Z}^* \ \wedge \ r \in \mathbb{Z}\}$. 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 = $\{\mathbf{partial}, \mathbf{done}\}$, quantified over by κ.

Figure 1 shows the step semantics, relating a store and a command to a new store, a trace and a continuation. In Figure 1, $f[a := b]$ denotes the (partial) function obtained by updating the (partial) function f:

$$f[a := b] = \{\ (x, y) \mid (x \neq a \wedge x \in dom(f) \wedge y = f(x)) \\ \vee\ (x = a \wedge y = b)\ \}$$

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 always terminate, e.g. a server.

3 Verification approach

We present an approach to verify input/output-related properties of programs. The first subsection gives an informal intuition of how the approach works. Subsection 3.2 defines the approach formally.

3.1 Informal introduction

This subsection describes an intuitive understanding of the I/O verification approach using simple examples.

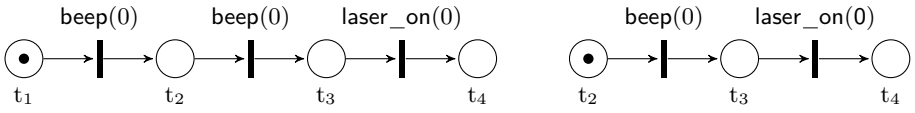

Fig. 2: Visual representation of a heap and one execution step thereof

Figure 2a shows a Petri net. The circles are called places, the black bars actions[1], and the dots in a circle are called tokens. This figure expresses that the program can beep twice, and afterwards (not earlier) turn on a laser beam. Instead of a graphical notation, we can describe the same as a multiset: $\{\!|\mathbf{token}(\mathrm{t}_1), \mathsf{beep}(\mathrm{t}_1, 0, \mathrm{t}_2), \mathsf{beep}(\mathrm{t}_2, 0, \mathrm{t}_3), \mathsf{laser\_on}(\mathrm{t}_3, 0, \mathrm{t}_4)|\!\}$. We call such a multiset a heap. The Petri nets in this subsection are graphical representation of heaps. You can ignore the zeroes for now.

In general, an assertion describes constraints. The assertion $x > 10$ constrains the program variable x to be greater than 10. Therefore, it constrains the store which maps program variables to values. Besides the store, an assertion also constrains the heap.

[1] In Petri net terminology actions are called transitions.

$\mathbf{token}(T_1) * \mathsf{beep}(T_1, \_, T_2) * \mathsf{beep}(T_2, \_, T_3) * \mathsf{laser\_on}(T_3, \_, T_4)$ is an assertion satisfied by the heap given earlier (together with a store and an interpretation mapping the logical variable T_1 to the place t_1, T_2 to t_2 and so on). You can ignore the underscore arguments for now.

Let's look at how we use an assertion. We write it in a Hoare triple as follows:

$\{\ \mathbf{token}(T_1) * \mathsf{beep}(T_1, \_, T_2) * \mathsf{beep}(T_2, \_, T_3) * \mathsf{laser\_on}(T_3, \_, T_4)\ \}$
beep();
beep();
laser_on()
$\{\ \mathbf{token}(T_4)\ \}$

One might be surprised that the intended I/O behavior is written in the precondition, while in the mindset of verifying memory state we have the habit of writing what the program does in the postcondition. Programs performing I/O are relatively often not intended to always terminate, and thus do not always reach the postcondition. You can consider the subassertions such as $\mathsf{beep}(T_2, \_, T_3)$ as permissions: a permission to execute an action under certain constraints.

You could associate with each point during execution of a program a heap, a store and an interpretation that describe the state of the program: e.g. if before the instruction $x := x + 1$, the store is $\{(x, 0)\}$, then after that instruction the store is $\{(x, 1)\}$. We do not use a heap during concrete execution of a program, but you can apply the same reasoning for the heap: after executing an I/O action, the permission to do so disappears from the heap[2], the token disappears, and a new token appears; see Figure 2b. After completely executing the example program starting from the example heap, the heap is thus $\{\!|\mathbf{token}(t_4)|\!\}$. Now look at the postcondition: it exactly constrains this to be the heap. In other words, the program is only allowed to terminate after having performed the actions.

Consider a program that reads one byte, and then outputs the same byte. Two of the many possible heaps that describe this behavior are $\{\!|\mathbf{token}(\mathrm{t}_1), \mathsf{read}(\mathrm{t}_1, \text{`x'}, \mathrm{t}_2), \mathsf{write}(\mathrm{t}_2, \text{`x'}, 0, \mathrm{t}_3)|\!\}$ and $\{\!|\mathbf{token}(\mathrm{t}_1), \mathsf{read}(\mathrm{t}_1, \text{`y'}, \mathrm{t}_2), \mathsf{write}(\mathrm{t}_2, \text{`y'}, 0, \mathrm{t}_3)|\!\}$. Since the world or environment can "return" different bytes when reading a byte, multiple heaps are required to describe all possible behaviors of the program.

The precondition $\mathbf{token}(T_1) * \mathsf{read}(T_1, \text{`x'}, T_2) * \mathsf{write}(T_2, \text{`x'}, \_, T_3)$ constrains the byte read to be 'x', and the byte written to be 'x'. Of the heaps given in the previous paragraph, only the first one models this precondition. Assertions can constrain the environment or the program, or both. $\mathsf{read}(T_1, \text{`x'}, T_2)$ constrains the value read from the world will be 'x'. This constrains the world or the environment. $\mathsf{write}(T_2, \text{`x'}, \_, T_3)$ states that the program must write value 'x'. This constrains the program. In both heaps and assertions, the last argument (that is not a place) of an action constrains the environment (input argument)

[2] This differs from the standard way of executing Petri nets where one usually does not remove parts of the Petri net during execution.

and the others constrain the program (output arguments). In the previous examples, the arguments with value zero in heaps and the arguments written as underscores in assertions were input arguments we were not interested in.

While constraining the environment can be useful, it is undesired in this example. A precondition that describes the intended behavior of the program that reads one byte and then prints that byte is: $\mathbf{token}(T_1) \ * \ \mathsf{read}(T_1, X, T_2) \ * \ \mathsf{write}(T_2, X, \_, T_3)$. In assertions, arguments of I/O actions can be any expression, and expressions in assertions can refer to program variables and logical variables such as X.

Quite often, the behavior of programs depends on the behavior of the environment. It is possible to write preconditions that take this into account, for example: $\mathbf{token}(T_1) * \mathsf{read}(T_1, X, T_2) * \mathbf{if}\ X > 10\ \mathbf{then}\ \mathsf{write}(T_2, X+1, \_, T_3)\ \mathbf{else}\ T_3 = T_2$. This assertion specifies the behavior of a program that reads a number and outputs one number higher if the number read is greater than ten. Note that writing this assertion is more convenient than writing or drawing all intended heaps.

For most nontrivial preconditions, a large (possibly infinite) number of heaps satisfy the precondition. This is necessary, because when thinking of the heaps as something executable, we will not know which of these heaps we will execute: it depends on the behavior of the environment, which we do not know in advance.

Let us try to write a contract for a program that reads a whole file. It needs read-permissions until end of file (EOF) has been read. Reading a negative number indicates reading EOF. We do not want the program to read past EOF, so we should not give more read-permissions than necessary. How many read-permissions we should write in the precondition thus depends on the behavior of the environment. We can write such a contract by defining a high-level I/O action as follows:

$$\begin{array}{l}\mathbf{predicate}\ \mathsf{reads}(T_1, \mathit{Text}, T_3) = \exists R. \exists T_2. \\ \quad \mathsf{read}(T_1, R, T_2) \ * \\ \quad \mathbf{if}\ R < 0\ \mathbf{then}\ \mathit{Text} = \mathrm{nil} \ * \ T_3 = T_2 \\ \quad \mathbf{else}\ (\mathsf{reads}(T_2, \mathit{Sub}, T_3) \ * \ \mathit{Text} = R :: \mathit{Sub})\end{array}$$

Note that this definition uses recursion. As we will see in the formal explanation, we allow infinite recursion (contrary to [10]), which is useful for e.g. a program that reads temperature sensor values forever.

Low-level actions that are not defined on top of other actions are called BIO actions. When verifying a program that uses an unverified library, the BIO actions can be actions provided by that library. If all libraries are verified but the kernel is not, BIO actions are system calls to the kernel.

In a contract both can be used, so

$$\{\, \mathbf{token}(T_1) * \mathsf{reads}(T_1, \mathit{Text}, T_2) \,\}\ c\ \{\, \mathbf{token}(T_2) \,\}$$

is a valid contract. In heaps only BIO actions are present. So $\{\!\{\mathbf{token}(\mathrm{t}_1), \mathsf{read}(\mathrm{t}_1, \text{`x'}, \mathrm{t}_2), \mathsf{read}(\mathrm{t}_2, -1, \mathrm{t}_3)\}\!\}$ is a valid heap for the precondition of this contract (together with an interpretation mapping $T_1$ to $\mathrm{t}_1$, $T_2$ to $\mathrm{t}_3$ and *Text* to the list $\text{`x'} :: -1 :: \mathrm{nil}$).

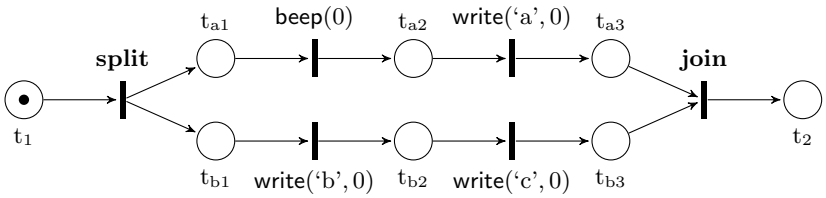

Fig. 3: Interleaving of actions

Figure 4 allows underspecification of actions: it allows to write 'a' and 'b' (but not both). A precondition of which this is a heap is $\mathbf{token}(T_1) * \mathsf{write}(T_1, \text{'a'}, \_, T_2) * \mathsf{write}(T_1, \text{'b'}, \_, T_2)$. It simply contains both permissions.

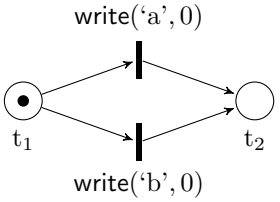

Fig. 4: Underspecification for actions

Instead of two write permissions, we can also write a contract with a write permission and a "dummy" I/O permission **no_op**, like this: $\mathbf{token}(T_1) * \mathbf{no\_op}(T_1, T_2) * \mathsf{write}(T_1, \text{'a'}, \_, T_2)$. This precondition expresses the program is allowed to write 'x' or terminate without performing any I/O.

Figure 3 allows arbitrary interleavings of beeping and writing 'a', and, writing 'b' and 'c'. The transition labeled **split** does not perform I/O. When executing this heap, the **split** transition splits the token of place t_1 into two tokens: one for t_{a1} and one for t_{b1}. Both tokens can then be used to execute a transition. This allows interleaving of actions[3]. The transition labeled **join** does the inverse of **split**: it merges two tokens into one. It only does this when both t_{a3} and t_{b3} have a token.

A Hoare triple with a precondition of which Figure 3 is a model is:

$$\left\{\begin{array}{l}\mathbf{token}(T_1) * \mathbf{split}(T_1, T_{a1}, T_{b1}) \\ * \ \mathsf{beep}(T_{a1}, \_, T_{a2}) * \mathsf{write}(T_{a2}, \text{'a'}, \_, T_{a3}) \\ * \ \mathsf{write}(T_{b1}, \text{'b'}, \_, T_{b2}) * \mathsf{write}(T_{b2}, \text{'c'}, \_, T_{b3}) \\ * \ \mathbf{join}(T_{a3}, T_{b3}, T_2)\end{array}\right\}$$
write('b');
beep();
write('c');
write('a')
$\{\ \mathbf{token}(T_2)\ \}$

The following is an incorrect program for the contract of the Hoare triple: write('b'); write('a'); write('c'); beep().

You might have noticed the precondition of the above contract can also be written without **split** and **join** by writing all possible interleavings by hand.

[3] It is also worth mentioning that **split** and **join** allow one to write contracts for multi-threaded programs.

However, **split** and **join** are very useful in combination with high-level I/O actions. Suppose we want to write a contract for Unix's cat, a program that reads a file and writes what it reads. We do not want to force cat to read the whole file before it starts writing, and we also do not want to enforce a fixed buffer size. If we define a high-level I/O action writes, similar to reads, we can write the precondition of cat as $\mathbf{token}(T_1) * \mathbf{split}(T_1, T_{r1}, T_{w1}) * \mathsf{reads}(T_{r1}, Xs, T_{r2}) * \mathsf{writes}(T_{w1}, Xs, T_{r2}) * \mathbf{join}(T_{r1}, T_{w1}, T_2)$.

3.2 The verification approach from a formal point of view

Heaps We use countably infinite multisets where an element can have a countably infinite number of occurrences. We define $\text{NatInf} = \mathbb{N} \cup \{\infty\}$. We represent a multiset of a set X as $X \to \text{NatInf}$. This allows infinitely many occurrences of items in the multiset. We define addition of any elements a and b in NatInf as follows ($+_{\mathbb{N}}$ denotes addition of natural numbers):

$$a + b = \begin{cases} a +_{\mathbb{N}} b & \text{if } a \neq \infty \wedge b \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

For multisets A and B, $A + B$ yields the multiset such that $(A + B)(x) = A(x) + B(x)$. Example: $\{\!\{1, 1, 2\}\!\} + \{\!\{1, 2, 3\}\!\} = \{\!\{1, 1, 1, 2, 2, 3\}\!\}$

Let X be a multiset of multisets. We define the union of X, written ΣX, as follows.

$$(\Sigma X)(y) = \begin{cases} \sum_{u \in X} u(y) & \text{if } |\{u \in X | u(y) > 0\}| \in \mathbb{N} \wedge \forall u \in X.u(y) \in \mathbb{N} \\ \infty & \text{otherwise} \end{cases}$$

We associate with each point during the execution of the program a multiset of permissions which we call the program's *heap* at that point. Such a permission is e.g. a permission to write to a file. For simplicity, the programming language used in this paper does not support dynamic memory allocation, so we do not use the heap for memory footprint or for the state of memory and the concrete execution does not use a heap. If desired, support for dynamic memory allocation with classic separation logic can easily be integrated.

We consider a set Places containing an infinite number of places.

We define Chunks as $\{\mathbf{token}(t) \mid t \in \text{Places}\} \cup \{\mathbf{join}(t_1, t_2, t_3) \mid t_1, t_2, t_3 \in \text{Places}\} \cup \{\mathbf{split}(t_1, t_2, t_3) \mid t_1, t_2, t_3 \in \text{Places}\} \cup \{bio(t_1, \overline{n}, r, t_2) \mid bio \in \text{BioNames} \wedge \overline{n} \in \mathbb{Z}^* \wedge r \in \mathbb{Z} \wedge t_1, t_2 \in \text{Places}\}$ and Heaps as Chunks $\to$ NatInf.

The intuitive meaning of a heap is given in Sec. 3.1.

Assertions Assertions, P, and assertion expressions, E, are written in the following grammar:

$V \in \text{LogicalVarNames}$, $p \in \text{PredNames}$.

$$E ::= n \mid v \mid V \mid E + E \mid E - E \mid \text{head}(E) \mid E +\!+ E \mid E :: E \mid \text{tail}(E) \mid \text{true} \mid \text{nil} \mid E = E \mid \neg E \mid E < E$$

$$P, Q, R ::= E \mid \mathbf{emp} \mid P * P \mid bio(E, \overline{E}, E, E) \mid \mathbf{split}(E, E, E) \mid \mathbf{join}(E, E, E) \mid \mathbf{no\_op}(E, E) \mid \mathbf{token}(E) \mid p(\overline{E}) \mid P \vee P \mid \exists V.\ P \mid \circledast_{V \in \mathbb{Z}} P \mid \circledast_{V \in \mathbb{Z}^*} P$$

Assertions can refer to both program variables and logical variables. Logical variables do not appear in programs and remain constant. An interpretation maps logical variables to their value, similar to a store for program variables, but a value can also be a place in Places.

Interpretations = LogicalVarNames $\rightharpoonup$ ($\mathbb{Z} \cup$ Places $\cup\, \mathbb{Z}^*$). We quantify over interpretations with i. We start logical variable names with a capital.

$[\![v]\!]_{s,i} = s(v)$ and $[\![V]\!]_{s,i} = i(V)$ (unspecified if s and i does not contain v and V respectively). Evaluation of the other assertion expressions is defined as $[\![op(\overline{e})]\!]_{s,i} = op([\![\overline{e}]\!]_{s,i})$ where op is an operator with zero or more arguments.

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

We use underscore, _, for assertion expressions we are not interested in; the meaning of $P[\_/V]$ is $\exists V.P$.

We write assertions of the form **if** E **then** P **else** $Q * R$; this is shorthand notation for $((E * P) \vee (\neg E * Q)) * R$. We abbreviate multiple existential quantifications, e.g. $\exists A, B$ is shorthand for $\exists A.\exists B$. We use standard abbreviations for boolean expressions with their expected meaning, e.g. $E_1 > E_2 \vee E_1 = E_2$ is abbreviated as $E_1 \geq E_2$, and $\neg(E_1 = E_2)$ as $E_1 \neq E_2$.

Assertions constrain the store, the heap and the interpretation. Figure 5 formally defines the semantics of assertions. It uses addition and union of multisets as defined in Sec. 3.2 (p. 9).

emp asserts that the heap is empty and $*$ is the separating conjunction [12].

The meaning of **token**, BIO, **no_op**, **split** and **join** assertions is explained in Sec. 3.1.

Predicates We use predicates based on and similar to [10]. A predicate can be considered as a named parameterized assertion, but the assertion can contain predicate names, including the name of the predicate itself. They are used for defining new input/output actions on top of BIO actions, other predicates, and the predicate itself. As example of a predicate definition that defines the action of reading repeatedly until an error or end of file is encountered is given in Section 3.1 (p. 7). Our definition of predicates is nonstandard, as we will see later, and allows infinite 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 $\times$ LogicalVarNames* $\times$ P.

We define J as PredNames $\times$ ($\mathbb{Z} \cup \mathbb{Z}^* \cup$ Places)* $\times$ Heaps. For predicates, we define a context $I_{\text{fix}} \subseteq J$ which expresses, for a given predicate name and argument values, the heap chunks a predicate assertion covers.

Let us define I_{fix}. Consider the function f:

$$
\begin{array}{rcl}
I,s,h,i \models E & \iff & [\![E]\!]_{s,i} = \text{true} \wedge h = \{\} \\
I,s,h,i \models bio(E_1,\overline{E},E_r,E_2) & \iff & h = \{bio([\![E_1,\overline{E},E_r,E_2]\!]_{s,i})\} \\
I,s,h,i \models \mathbf{join}(E_1,E_2,E_3) & \iff & h = \{\mathbf{join}([\![E_1,E_2,E_3]\!]_{s,i})\} \\
I,s,h,i \models \mathbf{split}(E_1,E_2,E_3) & \iff & h = \{\mathbf{split}([\![E_1,E_2,E_3]\!]_{s,i})\} \\
I,s,h,i \models \mathbf{no\_op}(E_1,E_2) & \iff & h = \{\mathbf{no\_op}([\![E_1,E_2]\!]_{s,i})\} \\
I,s,h,i \models \mathbf{token}(E) & \iff & h = \{\mathbf{token}([\![E]\!]_{s,i})\} \\
I,s,h,i \models \mathbf{emp} & \iff & h = \{\} \\
I,s,h,i \models P * Q & \iff & \exists h_1,h_2 \,.\; h_1 + h_2 = h \;\wedge\; I,s,h_1,i \models P \;\wedge \\
 & & I,s,h_2,i \models Q \\
I,s,h,i \models p(\overline{E}) & \iff & (p,([\![\overline{E}]\!]_{s,i}),h) \in I \\
I,s,h,i \models P \vee Q & \iff & (I,s,h,i \models P) \vee (I,s,h,i \models Q) \\
I,s,h,i \models \exists V.\; P & \iff & \exists x \in \mathbb{Z} \cup \mathbb{Z}^* \cup \text{Places}.\; I,s,h,i \models P[x/V] \\
I,s,h,i \models \circledast_{V\in\mathbb{Z}} P & \iff & \exists f : \mathbb{Z} \to \text{Heaps} \,.\; h = \Sigma_{n\in\mathbb{Z}} f(n) \wedge \forall n \in \\
 & & \mathbb{Z} \,.\; I,s,f(n),i \models P[n/V] \\
I,s,h,i \models \circledast_{V\in\mathbb{Z}^*} P & \iff & \exists f : \mathbb{Z}^* \to \text{Heaps} \,.\; h = \Sigma_{l\in\mathbb{Z}^*} f(l) \wedge \forall l \in \\
 & & \mathbb{Z}^* \,.\; I,s,f(l),i \models P[l/V]
\end{array}
$$

Fig. 5: Satisfaction relation of assertions

$$
f : \mathcal{P}(J) \to \mathcal{P}(J) : j \mapsto \{\; (p,(\overline{x}),h) \in J \mid\; \exists \overline{V}, P.\; (p,(\overline{V}),P) \in \text{PredDefs} \wedge j,\emptyset,h,\emptyset \models P[\overline{x}/\overline{V}] \;\}
$$

We will prove that f has a greatest fixpoint and define I_{fix} as the greatest fixpoint of f. The reason we take the greatest fixpoint instead of the least, is such that we can specify I/O behavior of programs that do not have a condition (such as: user clicks exit button) to terminate. Consider for example the following predicate definition: **predicate** $\mathsf{inf\_print}(T_1,X) = \exists T_2 \,.\; \mathsf{print}(T_1,X,\_,T_2) * \mathsf{inf\_print}(T_2,X+1)$. In case we use the greatest fixpoint, this expresses the action of printing a sequence of numbers (e.g. 1, 2, 3, ...). If we would have chosen to take the least fixpoint, the predicate $\mathsf{inf\_print}$ would be equivalent to false, and we would not be able to specify the I/O behavior of this never ending program.

To prove that the greatest fixpoint of f exist, we will apply Knaster-Tarski's theorem which states that any monotone function on a complete lattice has a greatest and a least fixpoint.

We consider the partial order relation $\subseteq$. Note that $J, \subseteq$ is a complete lattice.

We have to show that f is monotone, i.e. that for any $j_1, j_2 \in J$ such that $j_1 \subseteq j_2$, then $f(j_1) \subseteq f(j_2)$. Take such a j_1 and j_2. Let $y \in f(j_1)$ (in case this is impossible, i.e. $f(j_1) = \emptyset$, it immediately follows that $f(j_1) \subseteq f(j_2)$). Because $y \in f(j_1)$, $y \in \{(p,(\overline{x}),h) \mid \exists \overline{V}, P.\; (p,(\overline{V}),P) \in \text{PredDefs} \wedge j_1,\emptyset,h,\emptyset \models P[\overline{x}/\overline{V}]\}$. It

suffices to show that $\forall j_1, j_2, s, h, i, P.\ j_1 \subseteq j_2 \Rightarrow j_1, s, h, i \models P \Rightarrow j_2, s, h, i \models P$. This can easily be proven by induction on P. Note that negation of assertions is syntactically disallowed.

Big star The big star operator, $\circledast$, allows to easily express an infinite number of permissions. If one would accept '...' in formulae, we could write $\circledast_{V \in \mathbb{Z}} P$ as $P[0/V] * P[1/V] * P[-1/V] * P[2/V] * P[-2/V] * \ldots$. The following example expresses the permission to print any number greater than 20.

$$\mathbf{token}(T_1) * \circledast_{V \in \mathbb{Z}} (\mathbf{if}\ V > 20\ \mathbf{then}\ \mathsf{print}(T_1, V, \_, T_2)\ \mathbf{else}\ \mathbf{emp})$$

Besides the big star operator, predicates also allow us to express an infinite number of permissions. The big star operator therefore does not increase expressiveness, but it increases convenience.

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. We give a simple example of a Hoare triple:

$\{\ \mathbf{token}(T_1) * \mathsf{print}(T_1, 1, \_, T_2) * \mathsf{print}(T_2, 2, \_, T_3)\ \}$
print(1); print(2)
$\{\ \mathbf{token}(T_3)\ \}$

The contract of this program states that the program can write the numbers 1 and 2 in this order. If the program terminates, it has performed these actions.

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. For more interesting examples, see Section 5.

We define a relation traces $\subset$ (Heaps $\times$ Traces $\times$ (Heaps $\cup \{\bot\}$)) in Figure 6. $\{h\}\, \tau\, \{g\}$ denotes $(h, \tau, g) \in$ traces.

A heap expresses a permission to execute a (potentially infinite) sequence of BIO actions (with certain arguments). Multiple sequences of actions can be allowed by a heap. (h, τ, g), where $g \neq \bot$, expresses that h allows to perform the sequence of actions τ followed by a sequence of actions allowed by the heap g.

An element in the heap can make a prediction about the environment, e.g. $bio(t_1, \overline{n}, r, t_2)$ predicts performing the BIO bio with arguments $\overline{n}$ (starting at place t_1) will have return-value r. If a programs performs a BIO where the environment violates a prediction, the program can then perform any sequence of BIOs. In that case, we write $(h, \tau, \bot)$. This expresses the sequence of actions τ is allowed by h: τ will consists of a (potentially empty) list of allowed actions where the environment did not break a prediction, followed by an action where the environment broke a prediction, followed by any (finite, infinite or empty) sequence of actions.

Also note that a heap can contradict itself, e.g. $\{\mathbf{token}(\mathrm{t}_1), \mathsf{somebio}(\mathrm{t}_1, 2, \mathrm{t}_2), \mathsf{somebio}(\mathrm{t}_1, 3, \mathrm{t}_2)\}$. It contradicts itself because it says performing the BIO somebio (starting at place t_1) will return 2 and will return 3.

$$\textsc{TraceBio}\quad \frac{}{\{\{\mathbf{token}(t_1), bio(t_1, \overline{n}, r, t_2)\}\}\ bio(\overline{n}, r) :: \text{nil}\ \{\{\mathbf{token}(t_2)\}\}} \qquad \textsc{TraceNil}\quad \frac{}{\{\{\}\}\ \text{nil}\ \{\{\}\}}$$

$$\textsc{TraceFrame}\quad \frac{\{h\}\,\tau_1\,\{h'\}}{\{h + h_r\}\,\tau_1\,\{h' + h_r\}} \qquad \textsc{TraceComposition}\quad \frac{\{h\}\,\tau_1\,\{h_0\} \qquad \{h_0\}\,\tau_2\,\{h'\}}{\{h\}\,\tau_1 +\!+ \tau_2\,\{h'\}} \qquad \textsc{TraceLeak}\quad \frac{\{h\}\,\tau\,\{h' + h_r\}}{\{h\}\,\tau\,\{h'\}}$$

$$\textsc{TraceSplit}\quad \frac{}{\{\{\mathbf{token}(t_1), \mathbf{split}(t_1, t_2, t_3)\}\}\ \text{nil}\ \{\{\mathbf{token}(t_2), \mathbf{token}(t_3)\}\}}$$

$$\textsc{TraceJoin}\quad \frac{}{\{\{\mathbf{token}(t_1), \mathbf{token}(t_2), \mathbf{join}(t_1, t_2, t_3)\}\}\ \text{nil}\ \{\{\mathbf{token}(t_3)\}\}}$$

$$\textsc{TraceContradict}\quad \frac{r \neq r' \qquad \{h_1\}\,\tau_1 +\!+ bio(\overline{n}, r) :: \tau_2\,\{h_2\}}{\{h_1\}\,\tau_1 +\!+ bio(\overline{n}, r') :: \tau_3\,\{\bot\}}$$

Fig. 6: Definition of the traces relation. h quantifies over Heaps, not Heaps $\cup\, \bot$.

We define validity of a Hoare triple. Intuitively, it expresses that any execution starting from a state (a store s, a heap h and an interpretation i) that satisfies the precondition, results in a trace that is allowed by the heap h. In case the execution is a finished one (i.e. the program terminated) and the environment did not violate a prediction expressed in h, the state at termination must satisfy the postcondition.

$$\begin{array}{l}
\forall P, c, Q.\ \models \{P\}\,c\,\{Q\} \iff \\
\quad \forall s, h, i.\ I_{\text{fix}}, s, h, i \models P \Rightarrow \\
\quad\quad \forall s', \tau, \kappa.\ s, c \Downarrow s', \tau, \kappa \Rightarrow \\
\quad\quad\quad \exists g \in \text{Heaps} \cup \{\bot\}.\ \{h\}\,\tau\,\{g\} \wedge \\
\quad\quad\quad\quad (\kappa = \mathbf{done} \wedge g \neq \bot \Rightarrow I_{\text{fix}}, s', g, i \models Q)
\end{array}$$

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

Proof rules The proof rules are listed in Figure 7. Here, $\text{fpv}(E)$ and $\text{fpv}(P)$ returns the set of free program variables of the expression E and formula P respectively. The Frame rule is copied from separation logic [12].

Note that the programming language does not support recursive function calls. The structure of the proof tree is similar to the structure of the call graph.

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

$$\textsc{Assignment}\ \frac{}{\{P[e/v]\}\, v := e\, \{P\}} \qquad \textsc{While}\ \frac{\{e * P\}\, c\, \{P\}}{\{P\}\, \textbf{while}\ e\ \textbf{do}\ c\, \{\neg e * P\}} \qquad \textsc{Composition}\ \frac{\{P_1\}\, c_1\, \{P_2\} \qquad \{P_2\}\, c_2\, \{P_3\}}{\{P_1\}\, c_1; c_2\, \{P_3\}}$$

$$\textsc{Consequence}\ \frac{P_1 \Rightarrow P_2 \qquad \{P_2\}\, c\, \{P_3\} \qquad P_3 \Rightarrow P_4}{\{P_1\}\, c\, \{P_4\}} \qquad \textsc{If}\ \frac{\{P * e\}\, c_{\text{then}}\, \{Q\} \qquad \{P * \neg e\}\, c_{\text{else}}\, \{Q\}}{\{P\}\, \textbf{if}\ e\ \textbf{then}\ c_{\text{then}}\ \textbf{else}\ c_{\text{else}}\, \{Q\}}$$

$$\textsc{Skip}\ \frac{}{\{P\}\, \textbf{skip}\, \{P\}} \qquad \textsc{Disjunction}\ \frac{\{P_1\}\, c\, \{Q\} \qquad \{P_2\}\, c\, \{Q\}}{\{P_1 \vee P_2\}\, c\, \{Q\}} \qquad \textsc{Substitution}\ \frac{\{P\}\, c\, \{Q\} \qquad \text{fpv}(\overline{E}) \cap \text{mod}(c) = \emptyset}{\{P[\overline{E}/\overline{V}]\}\, c\, \{Q[\overline{E}/\overline{V}]\}}$$

$$\textsc{NoOp}\ \frac{\{P * \textbf{token}(V_{t1}) * \textbf{no\_op}(V_{t1}, V_{t2})\}\, c\, \{Q\}}{\{P * \textbf{token}(V_{t2})\}\, c\, \{Q\}} \qquad \textsc{Leak}\ \frac{\{P\}\, c\, \{Q * R\}}{\{P\}\, c\, \{Q\}}$$

$$\textsc{Split}\ \frac{\{P * \textbf{token}(V_{t2}) * \textbf{token}(V_{t3})\}\, c\, \{Q\}}{\{P * \textbf{token}(V_{t1}) * \textbf{split}(V_{t1}, V_{t2}, V_{t3})\}\, c\, \{Q\}}$$

$$\textsc{Join}\ \frac{\{P * \textbf{token}(V_{t3})\}\, c\, \{Q\}}{\{P * \textbf{token}(V_{t1}) * \textbf{token}(V_{t2}) * \textbf{join}(V_{t1}, V_{t2}, V_{t3})\}\, c\, \{Q\}}$$

$$\textsc{Bio}\ \frac{v \notin \text{fpv}(E_r)}{\{bio(V_{t1}, \overline{e}, E_r, V_{t2}) * \textbf{token}(V_{t1})\}\, v := bio(\overline{e})\, \{v = E_r * \textbf{token}(V_{t2})\}}$$

$$\textsc{Frame}\ \frac{\{P\}\, c\, \{Q\} \qquad \text{fpv}(R) \cap \text{mod}(c) = \emptyset}{\{P * R\}\, c\, \{Q * R\}} \qquad \textsc{Exists}\ \frac{\{P\}\, c\, \{Q\}}{\{\exists V.\ P\}\, c\, \{\exists V.\ Q\}}$$

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

Fig. 7: Proof rules

For the Consequence rule, we define implication of assertions as

$$\forall P, Q\ .\ P \Rightarrow Q \iff \forall s, h, i\ .\ I_{\text{fix}}, s, h, i \models P \Rightarrow I_{\text{fix}}, s, h, i \models Q$$

Note that $\circledast_{V \in \mathbb{Z}} P \Rightarrow P[E/V] * \circledast_{V \in \mathbb{Z}} (\textbf{if}\ V = E\ \textbf{then emp else}\ P)$ if $V \notin \text{fv}(E)$, where $\text{fv}(E)$ returns the set of free variables of E.

4 Soundness

Theorem 1 (Soundness). $\forall P, c, Q.\ \vdash \{P\}\, c\, \{Q\} \Rightarrow \models \{P\}\, c\, \{Q\}$.

Proof. Due to space limits, we only outline some cases of the induction on $\vdash \{P\}\, c\, \{Q\}$ which is nested in an induction on $s, c \Downarrow s', \tau, \kappa$. In these cases we know $I_{\text{fix}}, s, h, i \models P$.

- **Bio:** Because of the Bio proof rule, we know there is some E_r, v, bio, V_{t1}, $\overline{e}$, E_r, V_{t2} such that $P = bio(V_{t1}, \overline{e}, E_r, V_{t2}) * \mathbf{token}(V_{t1})$, $c = v := bio(\overline{e})$, and $Q = (v = E_r * \mathbf{token}(V_{t2}))$.
  We consider the case where the step rule that applies is Bio (the other case, where the step rule is Empty, is trivial). We know $s, c \Downarrow s', \tau, \kappa$. Thus there is some n such that $\tau = bio([\![\overline{e}]\!]_s, n) :: \text{nil}$.
  If $n \neq [\![E_r]\!]_{s,i}$ we need to prove that there is some g such that $\{h\}\, \tau\, \{g\}$. Note that $\kappa = \mathbf{partial}$. Let $g = \bot$. By applying the TraceBio and the TraceContradict rule, we obtain $\{h\}\, \tau\, \{g\}$.
  If $n = [\![E_r]\!]_{s,i}$, we know $h = \{\!\{bio([\![V_{t1}]\!]_{s,i}, [\![\overline{e}]\!]_s, n, [\![V_{t2}]\!]_{s,i}), \mathbf{token}([\![V_{t1}]\!]_{s,i})\}\!\}$. Let $g = \{\!\{\mathbf{token}([\![V_{t2}]\!]_{s,i})\}\!\}$. Because of the TraceBio rule, we obtain $\{h\}\, \tau\, \{g\}$, which we wanted to prove. Because of the Bio step rule we know $\kappa = \mathbf{done}$. $I_{\text{fix}}, s', g, i \models Q$ follows from the equalities of Q and g given above, and from $s' = s[v := n]$, which we know because of the Bio step rule.
- **Frame:** Because of the Frame proof rule, there is some P_0, Q_0, R such that $P = P_0 * R$, $Q = Q_0 * R$ and $\vdash \{P_0\}\, c\, \{Q_0\}$. Because $P = P_0 * R$ and $I_{\text{fix}}, s, h, i \models P$, there is some h_0, h_r such that $h_0 + h_r = h \wedge I_{\text{fix}}, s, h_0, i \models P_0 \wedge I_{\text{fix}}, s, h_r, i \models R$.
  Because of the induction hypothesis, we know $\models \{P_0\}\, c\, \{Q_0\}$. Thus for some h_0', it holds that $\{h_0\}\, \tau\, \{h_0'\}$ and $\kappa = \mathbf{done} \Rightarrow I_{\text{fix}}, s', h_0', i \models Q_0$.
  Let $g = h_0' + h_r$. By applying the TraceFrame rule, we obtain $\{h\}\, \tau\, \{g\}$, which is what we wanted to prove.
  If $\kappa = \mathbf{done}$, we need to prove that $I_{\text{fix}}, s', g, i \models Q$.
  Because $\text{fpv}(R) \cap \text{mod}(c) = \emptyset$ and $s, c \Downarrow s', \tau, \kappa$ and $I_{\text{fix}}, s, h_r, i \models R$, we obtain $I_{\text{fix}}, s', h_r, i \models R$.
  Combined with $I_{\text{fix}}, s', h_0', i \models Q_0$ we know $I_{\text{fix}}, s', g, i \models Q$ by definition of $*$.
- **Split:** Because of the Split proof rule, there is some $P_1, V_{t1}, V_{t2}, V_{t3}$ such that $P = P_1 * \mathbf{token}(V_{t1}) * \mathbf{split}(V_{t1}, V_{t2}, V_{t3})$. Let $P_0 = P_1 * \mathbf{token}(V_{t2}) * \mathbf{token}(V_{t3})$.
  We know $h = h_{p1} + \{\!\{\mathbf{token}([\![V_{t1}]\!]_{s,i}), \mathbf{split}([\![V_{t1}, V_{t2}, V_{t3}]\!]_{s,i})\}\!\}$ for some h_{p1}.
  Let $h_0 = h_{p1} + \{\!\{\mathbf{token}([\![V_{t2}]\!]_{s,i}), \mathbf{token}([\![V_{t3}]\!]_{s,i})\}\!\}$.
  Because of the induction hypothesis, $\models \{P_0\}\, c\, \{Q\}$. Combined with $I_{\text{fix}}, s, h_0, i \models P_0$, we obtain there is some g such that $\{h_0\}\, \tau\, \{g\}$.
  By using the TraceSplit and TraceFrame rule, we know $\{h\}\, \text{nil}\, \{h_0\}$. Now we can apply the TraceComposition rule to obtain $\{h\}\, \tau\, \{g\}$, which we wanted to prove.
  Because $\models \{P_0\}\, c\, \{Q\}$ and $I_{\text{fix}}, s, h_0, i \models P_0$, we know $\kappa = \mathbf{done} \Rightarrow I_{\text{fix}}, s', g, i \models Q$.

5 Examples

We give some examples of contracts.

5.1 Tee

```
predicate tee_out(T1, C, T2) = ∃Tp1, Tp2, Tr1, Tr2.
  split(T1, Tp1, Tr1)
  * stdout(Tp1, C, _, Tp2)
  * stderr(Tr1, C, _, Tr2)
  * join(Tp2, Tr2, T2)

function tee_out(c) =
  { token(T1) * tee_out(T1, c, T2) }
    stdout(c);
    stderr(c)
  { token(T2) }

predicate tee_outs(T1, Text, T2) = ∃Tout.
  if Text = nil then T2 = T1 else (
    tee_out(T1, head(Text), Tout)
    * tee_outs(Tout, tail(Text), T2) )

predicate tee(T1, Text, T2) = ∃Tr1, Tw1, Tr2, Tw2.
  split(T1, Tr1, Tw1)
  * reads(Tr1, Text, Tr2)
  * tee_outs(Tw1, Text, Tw2)
  * join(Tr2, Tw2, T2)

function main() =
  { token(T1) * tee(T1, Text, T2) }
    c2 := 0;
    while c2 ≥ 0 do (
      c1 := read();
      if c1 ≥ 0 then
        c2 := read();
        tee_out(c1);
        if c2 ≥ 0 then
          tee_out(c2)
        else skip
      else
        c2 := −1 )
  { token(T2) }
```

Fig. 8: Specification and implementation of the Tee program

Figure 8 lists the implementation and specification of a program that reads from standard input (until end-of-file), and writes what it reads to both standard output and standard error. The contract is written in a compositional manner: an action tee_out represents the action of writing to both standard output (stdout) and standard error (stderr). The action that represents the whole program, tee, 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. The reads predicate defined in Section 3.1 (p. 7) is used.

Figure 9 gives a proof outline for the tee_out function and the main function.

5.2 Read files mentioned in a file

The specification of Figure 10 allows the program to read a file, "f", which contains filenames (of length one character). The files mentioned in this file are read. The program prints to standard output the contents of these files in order. To make the example more interesting, the specifications allow the program to

Fig. 9: Proof outline of the tee_out and main function of the tee program

```
predicate freads(T1, F, Text, Tend) =
∃C, Sub, T2.
  fread(T1, F, C, T2)
  * if C ≥ 0 then
      freads(T2, F, Sub, Tend)
      * Text = C :: Sub
  else (
      Tend = T2
      * Text = nil)

predicate get_file(T1, Name, Text,
Tend) =
∃Handle, T2, T3.
  fopen(T1, Name, Handle, T2)
  * freads(T2, Handle, Text, T3)
  * fclose(T3, Handle, _, Tend)

predicate prints(T1, Text, Tend) =
∃T2.
  if Text = nil then
    Tend = T1
  else (
    print(T1, head(Text), _, T2)
    * prints(T2, tail(Text), Tend))

predicate get_files(T1, FNames,
Text, Tend) =
∃Text1, Text2, T2, Fname, SubNames.
  if FNames = nil then
    Tend = T1
    * Text = nil
  else (
    get_file(T1, Fname :: nil, Text1, T2)
    * get_files(T2, SubNames, Text2, Tend)
    * Fnames = Fname :: SubNames
    * Text = Text1 ++ Text2)
```

```
predicate read_fname_list(T1, Handle,
FNames, Tend) =
∃C, T2, Sub.
  fread(T1, Handle, C, T2)
  * if C ≥ 0 then
      read_fname_list(T2, Handle, Sub, Tend)
      * if C > 0 ∧ C ≤ 127 then
          FNames = C :: Sub
      else
          FNames = Sub
  else (
    Tend = T2
    * FNames = nil)

predicate main(T1, Fname, Tend) =
∃T2, Tmeta, Trw, FNames, Tmeta2, Tr, Tw,
Tmeta3, Tr2, Tw2, Trw2, Handle .
  fopen(T1, Fname, Handle, T2)
  * split(T2, Tmeta, Trw)
    * read_fname_list(Tmeta, Handle,
      FNames, Tmeta2)
    * fclose(Tmeta2, Handle, Tmeta3)
    * split(Trw, Tr, Tw)
      * get_files(Tr, FNames, Text, Tr2)
      * prints(Tw, Text, Tw2)
    * join(Tr2, Tw2, Trw2)
  * join(Tmeta3, Trw2, Tend)

function main() =
  { token(T1) * main(T1, 'f' :: nil, T2) }
    ...
  { token(T2) }
```

Fig. 10: Specification of a program that prints the contents of all files whose filenames are in a given list. This list is not static, it is read from a file "f".

choose whether to output a read character as soon as possible or postpone it, and whether to read a file as soon as possible or postpone it after or while reading "f". Filenames that consist of the zero character or non 7bit-ASCII are ignored.

The predicates representing actions are written on top of standard library actions like fopen for opening a file. These could be BIOs, but they can also be predicates built on top of lower-level actions, e.g. system calls.

The implementation is left out. As mentioned earlier, a verified implementation is shipped as an example with VeriFast.

5.3 Print any string of the grammar of matching brackets

```
predicate brackets(T1, T2) =
∃T_open, T_center, T_close.
  no_op(T1, T2)
  ∗ print(T1, '(', _, T_open)
  ∗ brackets(T_open, T_center)
  ∗ print(T_center, ')', _, T_close)
  ∗ brackets(T_close, T2)
```

```
function main() =
  { token(T1) ∗ brackets(T1, T2) }
    print('(');
    print(')')
  { token(T2) }
```

Fig. 11: Specification of a program that is allowed to output any string of the matching brackets grammar.

The specification of Figure 11 states that the program is only allowed to output a string of the grammar of matching brackets. Note that the specifications do not specify which string: any string of the grammar is allowed. The grammar under consideration is clearly visible in the specification, and is as follows:

$B ::= (B)B \mid \epsilon$

Here, ϵ denotes the empty string. In the specification, '(' and ')' are shorthand for 40 and 41 respectively (the ASCII number of these characters).

5.4 Turing machine

This example's only purpose is to illustrate the expressiveness of the contracts, i.e. it is not an example of a typical contract. It is possible to define a predicate

predicate tm(T_1, *EncodingOfTM*, *InitialState*, *TapeLeft*, *TapeRight*, T_2)

that expresses that the allowed I/O actions are the actions a Turing machine, given as second argument, performs. In other words, the program under consideration is allowed to perform the I/O actions that the Turing machine (TM) performs. The TM is encoded as a list of integers by serializing the table representation of the TM's transition function. The states and symbols are represented using integers.

Normally, the transition function of a TM maps the state and the symbol read from the tape, to the symbol to write on the tape, a new state, and an action: whether the TM's tape should move left, right, or not move. Typically, a TM accepts an input if the machine ends in an accepting state when launched from the full input on its tape. To make it more interactive, we add one input and one output action. The output action prints (i.e. writes to the world) the symbol on the tape without moving the tape. The tape does not move. The input action reads a symbol from the world and puts it on the tape at the current position.

A benefit of this approach is that it supports nonterminating programs naturally. The TM does not have to terminate. If the program terminates, it must have performed all the I/O actions the TM does (otherwise it will not obtain its postcondition $\mathbf{token}(T_2)$), and the TM must have terminated as well.

Besides interactive and nonterminating I/O we also want underspecified input/output. The specifications should thus allow multiple behaviors. This is easy to deal with by making the TM non deterministic.

The code of this example is left out to save space, but an annotated C version is shipped with VeriFast (see next section).

5.5 Mechanical verification of the examples

C versions of all examples in this paper have been mechanically verified using the VeriFast [7] tool. This increases confidence that the approach is usable in practice and not only on paper. The input of VeriFast is the C source code of the program, the contracts of the functions, and extra annotations. With this input, VeriFast outputs whether it is "convinced" the C implementation conforms to the contracts. In case VeriFast says yes, we are sure the implementation is free of bugs violating the contract (and basic properties like memory safety). Note that this is not just detecting bugs: it proves absence of bugs.

The examples in this paper are included in the directory `examples/io` in the VeriFast distribution. VeriFast is available from `http://distrinet.cs.kuleuven.be/software/VeriFast/`.

VeriFast only performs very limited automated proof; non-trivial proof steps must be explicitly specified in annotations. In particular, VeriFast does not encode predicates as SMT solver axioms; the user must explicitly fold and unfold predicates through 'open' and 'close' annotations. Therefore, VeriFast typically has a high annotation overhead and a short execution time. This is also the case with the presented examples:

Example	LoC	Lines of annotations	Time (ms)
Tee	24	54	192
Read files mentioned in a file	20	90	101
Matching brackets	4	20	96
Turing machine	7	58	103
Template method (Sec. 6)	39	77	60

The reported timings are using the Redux SMT solver on a Intel Core i5 CPU (max value of 10 runs).

6 Verifying memory state using I/O style verification

One might wonder whether there really is a difference between verifying I/O behavior and verifying memory-state. For example, we could consider writing to a file both as I/O and as performing memory operations. If the filesystem is in memory, and we consider not only the process that writes the file but also the kernel, then we are only manipulating memory. Therefore, verifying I/O and verifying memory state can be considered as another point of view, and not necessarily as technically different.

The regular approach for verification using the memory-state point of view can be insufficient for verifying applications for which the I/O behavior point of view is "natural". For example, the memory-state point of view usually only cares about the state when the program has reached the postcondition. This is insufficient for verifying I/O behavior. First, for I/O applications, nontermination is common and often not undesired, hence it is normal that the post-condition is never reached. Second, by looking only at (memory) state when the program has reached the postcondition, we ignore intermediate state. For I/O applications intermediate state is important: for a movie player, not only the last frame but all frames of the movie should be displayed correctly.

```
abstract class A {
  void template() {
    m1();
    m2();
  }
  abstract void m1();
  abstract void m2();
}
```

Fig. 12: Minimalistic example of the Template Method design pattern (in Java).

While the memory-state point of view can be insufficient from an I/O style point of view, one might wonder what happens when we try the other way around: what if we want to verify an application for which the memory-based point of view is expected at first sight, from an I/O point of view.

This section will take a quick look at this question, by looking at one example or use case: the Template Method design pattern.

Template Method [5] is an object-oriented design pattern in which an abstract class has a method implementing an algorithm of which a number of steps are delegated to subclasses. This delegation happens by calling abstract methods, which subclasses can implement.

How can we write the contract of this template method? The method must perform what the subclass's hook methods (m1 and m2 in Figure 12) will do, but we do not know what that will be. Furthermore, we do not want to change the contract or perform verification again of the template method when new subclasses are added. In this section, we will write an easy contract for the template method using the approach of this paper.

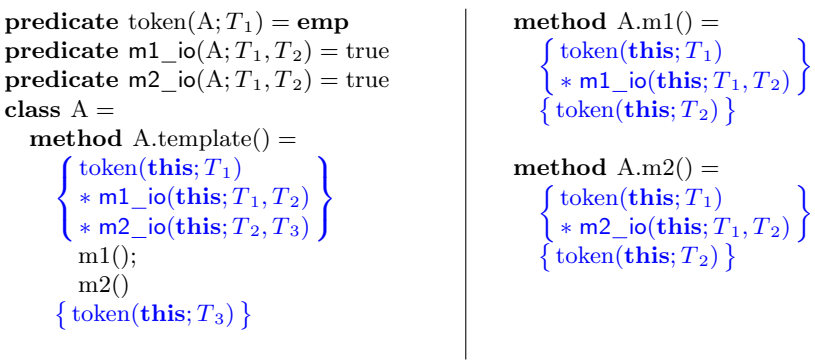

Fig. 13: A template method and its hook methods, with I/O style contracts. m1_io, m2_io and token are predicate families. For a subclass with implementation of m1 and m2, see Figure 14.

The language described so far does not support object-oriented programming, so we extend the language in a standard way to support object field access, method calls, casts, writing to object fields, object allocation and object deallocation. We assume a set Classes quantified over with C, and set MethodNames quantified over with m.

$e ::= \ldots \mid \textbf{this}$
$c ::= \ldots \mid v := e.v \mid v := v.m(\overline{e}) \mid v := (C)v \mid e.v := e \mid v := \textbf{new}(C) \mid \textbf{dispose}(v)$

We assume a set MethodDefs $\subset C \times m \times \overline{v} \times c$ and AbstractMethodDecls $\subset C \times m \times \overline{v}$ that describe the methods and abstract methods of the program under consideration. We only consider valid sets: there is no overlap in arguments ($\overline{v}$), an (abstract) method cannot appear twice in the same class, and a method cannot write to its arguments. We assume a partial function from Classes to Classes expressing inheritance. We only support non-circular single inheritance.

The step semantics is extended as expected. Note that it will need a concrete heap, to keep track of (1) the values of object fields to support memory (de)allocation and field access, and (2) the dynamic type of objects to support dynamic binding of method calls.

The assertion language needs to be extended to support assertions describing the fields of objects. We also need predicate families, i.e. predicates indexed by class. This allows multiple versions of a predicate. For the semantics of predicate families and proof rules, we refer to [10]. Furthermore, the assertions can describe that an object is an instance of a class C (not a subclass of C). We drop the keyword "**token**". We drop support for **split** and **join**.

$E ::= \ldots \mid \textbf{this}$
$P ::= \ldots \mid E.v \mapsto E \mid p(E; \overline{E}) \mid E : C$

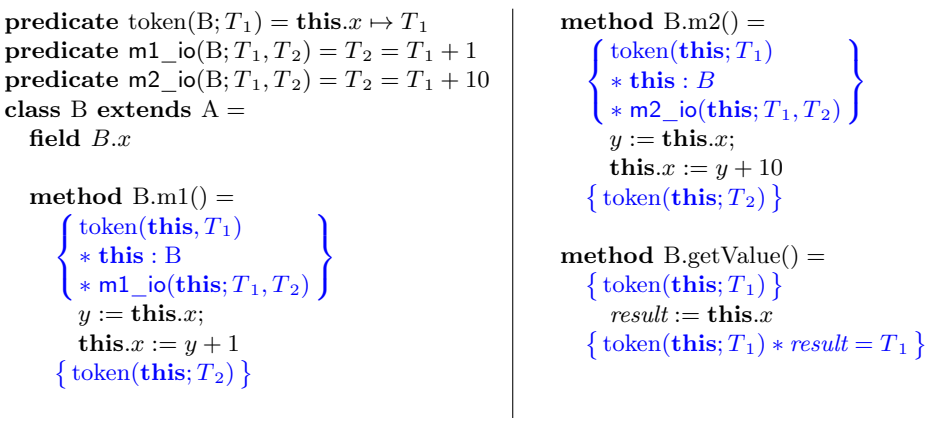

Fig. 14: A subclass implementing hook methods m1 and m2.

Figure 13 shows how the contract of the template method can then be written. Figure 14 shows an example of a subclass.

This section presented an approach for verifying memory, while Sec. 3 presented an approach for verifying I/O properties. Both are instantiations of the same specification style. Note that the approach of Sec. 3 supports some more features: nondeterminism/underspecification and interleaving.

An annotated Java version of the example of this section is shipped with VeriFast.

7 Related work

Approaches for verifying input/output behavior and case studies doing so have been developed and performed before.

The Verisoft email client [1] has a verified fullscreen text-based user interface. The approach identifies points in the main loop of the program, and restricts I/O to the screen to only these points. This approach is elegant but does not scale. A program of reasonable size typically uses libraries that also perform I/O. A contribution of our approach is compositional I/O verification. This allows verified libraries to perform I/O.

The heaps (not the assertions) of the approach described by this paper can be represented using Petri nets. The assertions are more expressive by using features such as using actions composed out of other actions. These features are also present in coloured Petri nets [8], which is a generalization of Petri nets where tokens are represented as data values. By modeling the contracts as a coloured Petri net, one could analyse the contracts using techniques to analyze coloured Petri nets. Note that the goal of our contribution is not just to model input/output behavior, but to verify input/output behavior of programs.

Nakata and Uustalu [9] define a Hoare logic for a programming language with tracing semantics. The assertion language is inspired by interval temporal logic. The programming language is defined using big step semantics, but relates a program and a state to a trace, instead of to a state as is usually done. An assertion can express properties of this (potentially infinite) trace. The paper rather provides a framework to build upon than proposing a final assertion language. In every example of the paper, the assertion language is extended to support the example. Using such an extension it is possible to prove liveness properties, which our approach does not support. The paper uses Coq and not a verification tool specialized for software verification. It is unclear how well the approach blends in with solutions for other problems, e.g. aliasing.

Linear Time Calculus (LTC) provides a methodology for modeling dynamic systems in general in an extension of first order logic. In LTC, an action has an argument which represents a point in (linear) time when the action happens. Such a point in time is a natural number and is clearly before or after another point in time. This differs from our approach where an action has two arguments, each representing a place. A place is not always clearly before or after another place. LTC is a generic approach, while our approach focuses on software verification. For an explanation of LTC we refer to [2], which also shows many tool-supported practical applications.

Model checking [3,11,4] allows checking whether properties written in a temporal language such as LTL and CTL hold for a model. Such models can be created automatically from the software subject to verification. Expressing temporal properties in temporal languages is natural, making it a good candidate for expressing input/output-related properties. Furthermore, liveness properties can be expressed. The approach suffers from the state explosion problem, a problem that remains after major improvements made in the last three decades [4].

Wisnesky, Malecha, and Morrisett [13] verify I/O properties by constraining the list of performed actions in the postcondition, but this approach does not seem to prevent nonterminating programs to perform undesired I/O.

8 Conclusions and future work

We identified several requirements for approaches that verify the input/output behavior of computer programs, including modularity, compositionality, soundness and non-determinism of the environment. We created a verification approach that meets these requirements.

Because the approach is designed to work compositionally and modularly, we hope the approach works well for bigger applications, but to confirm this, a real-world case study of considerable size should be carried out. Such a case study is future work.

Acknowledgements

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

References

1. Gerd Beuster, Niklas Henrich, and Markus Wagner. Real world verification – Experiences from the Verisoft email client. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning (ESCoR 2006)*, volume 192 of *CEUR Workshop Proceedings*, pages 112–125. CEUR-WS.org, August 2006.
2. Bart Bogaerts, Joachim Jansen, Maurice Bruynooghe, Broes De Cat, Joost Vennekens, and Marc Denecker. Simulating dynamic systems using linear time calculus theories. *Theory and Practice of Logic Programming*, 14:477–492, 7 2014.
3. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1981. Springer-Verlag.
4. Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin Heidelberg, 2012.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
6. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, 1969.
7. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frederic Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods 2011*, volume 6617 of *LNCS*, pages 41–55, Heidelberg, 2011. Springer.
8. Lars M. Kristensen, Søren Christensen, and Kurt Jensen. The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.
9. Keiko Nakata and Tarmo Uustalu. A Hoare logic for the coinductive trace-based big-step semantics of While. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 488–506, Berlin, Heidelberg, 2010. Springer-Verlag.
10. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 32nd Symposium on Principles of Programming Languages*, pages 247–258. ACM, 2005.
11. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
12. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Symposium on Logic in Computer Science*, pages 55–74, Washington, 2002. IEEE.
13. Ryan Wisnesky, Gregory Malecha, and Greg Morrisett. Certified web services in Ynot. In *5th International Workshop on Automated Specification and Verification of Web Systems*, July 2009.