Modular Verification of Liveness Properties of the I/O Behavior of Imperative Programs

Bart Jacobs[0000−0002−3605−249X]

KU Leuven, Department of Computer Science, imec-DistriNet Research Group
bart.jacobs@cs.kuleuven.be

Abstract. One way of verifying systems whose components interact by exchanging messages, such as distributed systems or certain types of concurrent systems, is by defining a protocol that governs the communication between the components and then verifying that each component's input and output (I/O) actions comply with its role in the protocol.
In this paper, we propose a separation logic-based approach for specifying and verifying liveness properties of the I/O behavior of such components implemented as imperative programs, such as the property that a server eventually responds to each request. Our approach builds on earlier work for specifying safety properties of the I/O behavior of programs in separation logic by means of abstract nested Hoare triples, and encodes a liveness property verification problem into a termination verification problem by specifying that some appropriately chosen I/O operation (for example, the response to the N'th request, for some unknown but fixed N) will cause the program to terminate.

1 Introduction

Consider the following program:

$$\textbf{loop}\ (\textbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \textbf{in}\ \mathsf{send}(\mathsf{msg}))$$

This program implements a simple echo server. It sits in an infinite loop. In each iteration, it receives a message and echoes it back out. The problem we address in this paper is: how to specify and verify the safety and liveness properties of the I/O behavior of programs such as this one in Hoare logic [4]? We target Hoare logic so that we obtain a modular verification approach. For the example program specifically, we want to be able to specify the safety property that it only sends messages that it has received, and that it sends a message at most once[1]; furthermore, we want to be able to specify the liveness property that it sends each message it has received at least once.

For specifying the safety properties of the I/O behavior, we apply our earlier work [9, 8, 7] on *abstract nested Hoare triples* in *separation logic* [11]. For specifying liveness properties, we build on earlier work [5] on verification of *basic*

[1] For simplicity, in this paper we assume a program never receives the same message more than once.

liveness. We combine and extend this work to obtain an approach that supports specifying a wide range of liveness properties. Adopting the terminology of Chang *et al.* [2], we show how our approach supports not just *guarantee properties* (of the form $\Diamond p$, *eventually* p in temporal logic) (example: the program eventually terminates) and *simple response properties* (of the form $\Box\Diamond p$, *always eventually* p) (example: the program always eventually responds to each request), but also *general response properties* (of the form $\bigwedge_i \Box\Diamond p_i$) (example: the program always eventually responds to each request and always eventually emits a heartbeat signal), *simple reactivity properties* (of the form $\Box\Diamond p \Rightarrow \Box\Diamond q$) (example: if the program always eventually receives a request, it always eventually responds to each request), *general reactivity properties* (of the form $\bigwedge_i(\Box\Diamond p_i \Rightarrow \Box\Diamond q_i)$) (example: the program always eventually emits a heartbeat signal, and if it always eventually receives a request, it always eventually responds to each request) and *persistence properties* (of the form $\Diamond\Box p$) (example: the program eventually always sends messages using the new message format).

The structure of this paper is as follows. In §2, we present our approach informally. In §3, we formalize the syntax and semantics of a simple programming language with I/O. In §4, we formalize our logic for liveness. We discuss related work in §5 and offer a conclusion in §6.

2 Our Approach, Informally

In this section, we present our approach informally. In subsequent sections, we formalize the programming language and the logic.

2.1 Safety

For specifying the safety properties of the I/O behavior, we apply our earlier work [9, 8, 7]: we use *abstract nested Hoare triples* in *separation logic* [11]. For example, a specification for the program `send("Hello");send("world!")` could look as follows:

$$\{P_1 \wedge \mathsf{send\_}(P_1, \texttt{"Hello"}, P_2) \wedge \mathsf{send\_}(P_2, \texttt{"world!"}, P_3)\}$$
$$\mathsf{send}(\texttt{"Hello"});\mathsf{send}(\texttt{"world!"})$$
$$\{P_3\}$$

where the specification for send is

$$\{P \wedge \mathsf{send\_}(P, m, Q)\}\ \mathsf{send}(m)\ \{Q\}$$

and $\mathsf{send\_}(P, m, Q)$ is a *higher-order predicate* that states that any resource that satisfies separation logic predicate P is sufficient to send message m; doing so consumes such a resource and produces a resource that satisfies separation logic predicate Q. We refer to higher-order predicates such as $\mathsf{send\_}$ as *abstract nested Hoare triples* because their meaning is similar to that of the Hoare triple $\{P\}\ \mathsf{send}(m)\ \{Q\}$. The difference between abstract nested Hoare triples and actual nested Hoare triples is that the abstract ones are in fact just user-defined

predicates, so their meaning is entirely defined by the user, rather than having a fixed meaning assigned by the logic. The meaning of predicate $\mathsf{send\_}$, for example, is defined by the author of the module that implements function send.

We can read the specification for the `"Hello, world!"` example as follows: for any P_1, P_2, and P_3, if a resource (satisfying) P_1 allows me to send `"Hello"` and obtain a resource P_2, and P_2 allows me to send `"world!"` and obtain P_3, and I have P_1, then I can run safely and if I terminate, I will end up with P_3.

In a similar fashion we can specify programs that perform both input and output. For example, the body of the echo loop can be specified as follows:

$$\{P_1 \wedge \mathsf{recv\_}(P_1, m, P_2) \wedge \mathsf{send\_}(P_2, m, P_3)\}$$
$$\textbf{let } \mathsf{msg} = \mathsf{recv}() \textbf{ in } \mathsf{send}(\mathsf{msg})$$
$$\{P_3\}$$

where the specification for recv is

$$\{P \wedge \mathsf{recv\_}(P, m, Q)\}\ \mathsf{recv}()\ \{Q \wedge \mathsf{res} = m\}$$

In postconditions, we use res to denote the result of a command. The predicate $\mathsf{recv\_}(P, m, Q)$ means that if you have P, you are allowed to receive. This will consume P and produce Q. Furthermore, the message you will receive is m. It follows that the specification for the single echo iteration states that it is allowed to receive a message, and then send the message it received, and that if it terminates, it shall indeed have performed these actions.

Using these ingredients, we can specify the full echo server, by first coinductively defining two predicates for expressing permission to receive and send an infinite sequence of messages, respectively:

$$\mathsf{recv\_stream}(P, m \cdot \mu) = \exists Q.\ \mathsf{recv\_}(P, m, Q) \wedge \mathsf{recv\_stream}(Q, \mu)$$
$$\mathsf{send\_stream}(P, m \cdot \mu) = \exists Q.\ \mathsf{send\_}(P, m, Q) \wedge \mathsf{send\_stream}(P, \mu)$$

We use μ to range over infinite sequences of messages[2], and $m \cdot \mu$ to denote the sequence with head m and tail μ. One possible specification for the safety of the echo server is then:

$$\{P_\mathsf{r} * P_\mathsf{s} \wedge \mathsf{recv\_stream}(P_\mathsf{r}, \mu) \wedge \mathsf{send\_stream}(P_\mathsf{s}, \mu)\}$$
$$\textbf{loop } (\textbf{let } \mathsf{msg} = \mathsf{recv}() \textbf{ in } \mathsf{send}(\mathsf{msg}))$$
$$\{\mathsf{False}\}$$

This is where separation logic's separating conjunction $-\!*\!-$ comes in; it allows us to express that I have permission P_r to receive, and *separately* I have permission P_s to send. We can apply separation logic's *frame rule*

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

[2] For simplicity, we assume the same message does not appear twice in such a sequence. As a result, "echoing out the message at position i in μ" is equivalent to "responding to the i'th request".

to the triple

$$\{P_\mathsf{r} \wedge \mathsf{recv\_stream}(P_\mathsf{r}, m \cdot \mu)\}\ \mathsf{recv}()\ \{\exists P'_\mathsf{r}.\ P'_\mathsf{r} \wedge \mathsf{recv\_stream}(P'_\mathsf{r}, \mu) \wedge \mathsf{res} = m\}$$

taking *frame* $R = (P_\mathsf{s} \wedge \mathsf{send\_stream}(P_\mathsf{s}, m \cdot \mu))$ to obtain

$$\begin{array}{l}\{P_\mathsf{r} * P_\mathsf{s} \wedge \mathsf{recv\_stream}(P_\mathsf{r}, m \cdot \mu) \wedge \mathsf{send\_stream}(P_\mathsf{s}, m \cdot \mu)\}\\ \mathsf{recv}()\\ \{\exists P'_\mathsf{r}.\ P'_\mathsf{r} * P_\mathsf{s} \wedge \mathsf{recv\_stream}(P'_\mathsf{r}, \mu) \wedge \mathsf{send\_stream}(P_\mathsf{s}, \mathsf{res} \cdot \mu)\}\end{array}$$

If we only had $P_\mathsf{r} \wedge P_\mathsf{s}$ instead of $P_\mathsf{r} * P_\mathsf{s}$, we would not be able to conclude that $\mathsf{recv}()$ preserves P_s.

Note, however, that this specification requires that the server reply to requests in the order in which it receives them. This is overly strict, and forbids concurrent implementations. To allow the server to reply to requests in any order, we replace predicate $\mathsf{send\_stream}$ by $\mathsf{send\_all}$, defined as follows:

$$\mathsf{send\_all}(P, m \cdot \mu) = \exists P_1, P_2.\ (P \Rrightarrow P_1 * P_2) \wedge \mathsf{send\_}(P_1, m, \mathsf{True}) \wedge \mathsf{send\_all}(P_2, \mu)$$

where $P \Rrightarrow Q$, called a *view shift* [6], essentially simply means that P implies Q.[3]

2.2 Basic liveness

By itself, the logic we proposed in earlier work for verification of safety properties of the I/O behavior of programs guarantees that the program does not perform any I/O that is not permitted by the specification, and that *if* the program terminates, it will have performed the I/O prescribed by the specification, but it does not guarantee that the program will indeed perform the prescribed I/O. It might instead go into a silent infinite loop, and not perform any I/O from some point on. Or it might respond to some requests, but allow others to starve.

Of course, we could combine the I/O logic with any logic for verifying termination, such as the one we proposed in earlier work [5]. Interpreted in such a combined logic, the specifications shown above for the `"Hello, world!"` example and for the single echo loop iteration express full total correctness. However, this approach is not appropriate for the complete echo server: it is not supposed to terminate. (Indeed, the specification for the echo server shown above, interpreted in a total logic, is unimplementable.)

In earlier work [5], we proposed an approach for verifying *basic liveness* of non-terminating programs. Basic liveness means that the program always eventually performs I/O, i.e. that it never completely stops responding. Our approach was to reduce the basic liveness verification problem to a termination verification problem by imagining that the N'th I/O operation performed by the program,

[3] More precisely, $P \Rrightarrow Q$ means that a state satisfying P can be transformed into a state satisfying Q by optionally performing an update of the *ghost state*. However, the reader can ignore the concept of ghost state for now.

for some fixed but unknown N, causes abrupt termination of the program. For example, in this approach, we can prove basic liveness of **loop** (beep()) by assuming the following specification for beep():

$$\{\mathsf{IO}(n)\}\ \mathsf{beep}()\ \{0 < n \wedge \mathsf{IO}(n-1)\}$$

where $\mathsf{IO}(n)$ means the $(n+1)$'th next I/O operation will terminate the program (so, in particular, $\mathsf{IO}(0)$ means that the very next I/O operation will terminate the program), and then proving

$$\{\mathsf{IO}(n)\}\ \mathbf{loop}\ (\mathsf{beep}())\ \{\mathsf{False}\}$$

in a total logic. In words, this specification states that, assuming that the $(n+1)$'th I/O operation performed by the program terminates the program, the program terminates, and furthermore, the program does not terminate normally (because the postcondition is False, so **skip** and beep(); beep() do not satisfy this specification[4]). Here, the number of I/O operations left before the program is terminated can be used as a loop variant.

2.3 Simple responsiveness

Notice that for the echo server, basic liveness is insufficient as a specification: an implementation could satisfy it simply by receiving requests, but not responding to any of them.

In this paper, we improve upon our earlier work by proposing an approach for specifying and verifying not just basic liveness, but richer, application-specific liveness properties of a program's I/O behavior as well, such as the property that our echo server eventually responds to each request. We again reduce the liveness property verification problem to a termination verification problem by imagining that a particular well-chosen I/O operation causes abrupt termination of the program. For example, to prove that the echo server eventually responds to each request, it is sufficient to prove that it terminates, under the assumption that the response to the k'th request, for some fixed but unknown k, terminates the program. Since the choice of which I/O operation terminates the program is application-dependent, we do not encode it directly into the I/O primitives' specification, as we did for basic liveness verification. Instead, we integrate it with the program's I/O safety specification, by stating in the program's precondition that the postcondition of the I/O operation that terminates the program is False. For example, if we apply this approach to the echo server, we obtain the following specification:

$$\{P_\mathsf{r} * P_\mathsf{s} \wedge \mathsf{recv\_stream}(P_\mathsf{r}, \mu) \wedge \mathsf{send\_all}'(P_\mathsf{s}, k, \mu) \wedge 0 \leq k\}$$
$$\mathbf{loop}\ (\mathbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \mathbf{in}\ \mathsf{send}(\mathsf{msg}))$$
$$\{\mathsf{False}\}$$

[4] Actually, beep(); beep() does satisfy the specification for $n < 2$, but the point is that it does not satisfy the specification $\forall n.\ \{\mathsf{IO}(n)\}\ -\ \{\mathsf{False}\}$.

where $\mathsf{send\_all}'(P, k, \mu)$ is defined in exactly the same way that $\mathsf{send\_all}(P, \mu)$ was defined above (so it means that resource P gives permission to send all of the messages in μ, in any order), except that the postcondition of sending the message at index k in μ is False (so sending the message at index k in μ terminates the program) (if $0 \leq k$)[5]:

$$\begin{array}{l}\mathsf{send\_all}'(P, k, m \cdot \mu) = \\ \quad \exists P_1, P_2.\ (P \Rrightarrow P_1 * P_2) \wedge \mathsf{send\_}(P_1, m, k \neq 0) \wedge \mathsf{send\_all}'(P_2, k-1, \mu))\end{array}$$

where $k \in \mathbb{Z}$. Again, k can be used as a loop variant to prove termination of the loop.

This approach also allows us to verify other echo server implementations, such as one that performs buffering and reordering of requests, against the same specification:

$$\begin{array}{l}\{P_{\mathsf{r}} * P_{\mathsf{s}} \wedge \mathsf{recv\_stream}(P_{\mathsf{r}}, \mu) \wedge \mathsf{send\_all}'(P_{\mathsf{s}}, k, \mu)\} \\ \textbf{loop}\ (\\ \quad \textbf{let}\ \mathsf{msg1} = \mathsf{recv}()\ \textbf{in} \\ \quad \textbf{let}\ \mathsf{msg2} = \mathsf{recv}()\ \textbf{in} \\ \quad \textbf{let}\ \mathsf{msg3} = \mathsf{recv}()\ \textbf{in} \\ \quad \mathsf{send}(\mathsf{msg3}); \\ \quad \mathsf{send}(\mathsf{msg1}); \\ \quad \mathsf{send}(\mathsf{msg2}) \\ ) \\ \{\mathsf{False}\}\end{array}$$

Other simple responsiveness properties (i.e. properties of the form $\Box\Diamond p$) can be encoded similarly.

2.4 General responsiveness

Suppose the echo server should eventually respond to each request, and also eventually log each request. We can reduce this property to termination by imagining that *either* the response to the k'th request, for some k, *or* logging the j'th request, for some j, terminates the program:

$$\begin{array}{l}\left\{\begin{array}{l} P_{\mathsf{r}} * P_{\mathsf{s}} * P_{\mathsf{l}} \wedge \mathsf{recv\_stream}(P_{\mathsf{r}}, \mu) \wedge \\ \mathsf{send\_all}'(P_{\mathsf{s}}, k, \mu) \wedge \mathsf{log\_all}'(P_{\mathsf{l}}, j, \mu) \wedge (0 \leq k \vee 0 \leq j) \end{array}\right\} \\ \textbf{loop}\ (\textbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \textbf{in}\ \mathsf{send}(\mathsf{msg}); \mathsf{log}(\mathsf{msg})) \\ \{\mathsf{False}\}\end{array}$$

Since the program only knows that *either* $0 \leq k$ *or* $0 \leq j$, it must *both* log all requests *and* respond to all of them to be sure of termination.

Other general responsiveness properties (i.e. properties of the form $\bigwedge_i \Box\Diamond p_i$) can be encoded similarly.

[5] Notice that if $k < 0$, $\mathsf{send\_all}'(P, k, \mu)$ is equivalent to $\mathsf{send\_all}(P, \mu)$.

2.5 Reactivity

Suppose receiving can suffer transient failures. To prove that the echo server always eventually sends, we need to assume that receiving always eventually succeeds. This can be expressed as follows:

$$\{P_\mathsf{r} * P_\mathsf{s} \wedge \mathsf{recv\_stream}'(P_\mathsf{r}, \nu, \mu) \wedge \mathsf{send\_all}'(P_\mathsf{s}, k, \mu)\}$$
$$\textbf{loop}\ (\textbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \textbf{in}\ \textbf{if}\ \mathsf{msg} \neq \bot\ \textbf{then}\ \mathsf{send}(\mathsf{msg}))$$
$$\{\mathsf{False}\}$$

where $\mathsf{recv\_stream}'(P, \nu, \mu)$ means that receiving message μ_i will fail ν_i times before succeeding, for all $i \in \mathbb{N}$:

$$\mathsf{recv\_stream}'(P, 0 \cdot \nu, m \cdot \mu) = \exists Q.\ \mathsf{recv\_}(P, m, Q) \wedge \mathsf{recv\_stream}'(Q, \nu, \mu)$$
$$\mathsf{recv\_stream}'(P, (n+1) \cdot \nu, \mu) = \exists Q.\ \mathsf{recv\_}(P, \bot, Q) \wedge \mathsf{recv\_stream}'(Q, n \cdot \nu, \mu)$$

and $\nu \in \mathbb{N}^\omega$ ranges over infinite sequences of natural numbers.

Other reactivity properties (i.e. properties of the form $\Box\Diamond p \Rightarrow \Box\Diamond q$) can be encoded similarly.

2.6 General reactivity

Suppose the echo server should always eventually emit a heartbeat signal, even if receiving fails persistently:

$$\left\{ \begin{array}{l} P_\mathsf{r} * P_\mathsf{s} * P_\mathsf{h} \wedge \mathsf{recv\_stream}'(P_\mathsf{r}, \tilde{\nu}, \mu) \wedge \\ \mathsf{send\_all}'(P_\mathsf{s}, k, \mu) \wedge \mathsf{heartbeats}'(P_\mathsf{h}, j) \wedge (\tilde{\nu} \in \mathbb{N}^\omega \wedge 0 \leq k \vee 0 \leq j) \end{array} \right\}$$
$$\textbf{loop}\ (\textbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \textbf{in}\ \mathsf{heartbeat}(); \textbf{if}\ \mathsf{msg} \neq \bot\ \textbf{then}\ \mathsf{send}(\mathsf{msg}))$$
$$\{\mathsf{False}\}$$

where $\tilde{\nu} \in (\mathbb{N} \cup \{\infty\})^\omega$ and $\mathsf{heartbeats}'$ is defined as follows:

$$\mathsf{heartbeats}'(P, j) =$$
$$\exists P_1, P_2.\ (P \Rrightarrow P_1 * P_2) \wedge \mathsf{heartbeat\_}(P_1, j \neq 0) \wedge \mathsf{heartbeats}'(P_2, j-1)$$

Other general reactivity properties (i.e. properties of the form $\bigwedge_i (\Box\Diamond p_i \Rightarrow \Box\Diamond q_i)$) can be encoded similarly.

2.7 Persistence

Suppose the echo server is allowed to drop a finite number of requests:

$$\{ P_\mathsf{r} * P_\mathsf{s} \wedge \mathsf{recv\_stream}(P_\mathsf{r}, \mu) \wedge \mathsf{send\_all}''(P_\mathsf{s}, \mu) \}$$
$$\mathsf{commit}(\mathsf{false}); \mathsf{commit}(\mathsf{false}); \mathsf{commit}(\mathsf{false});$$
$$\mathsf{recv}(); \textbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \textbf{in}\ \mathsf{send}(\mathsf{msg}); \mathsf{recv}();$$
$$\mathsf{commit}(\mathsf{true});$$
$$\textbf{loop}\ (\textbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \textbf{in}\ \mathsf{send}(\mathsf{msg}))$$
$$\{\mathsf{False}\}$$

which uses the following auxiliary definition:

$$\begin{array}{l}\mathsf{send\_all''}(P, m \cdot \mu) = \\ \quad (\exists Q_1, Q_2.\ \mathsf{commit\_}(P, \mathsf{false}, Q_1 * Q_2) \wedge \mathsf{send\_}(Q_1, m, \mathsf{True}) \wedge \mathsf{send\_all''}(Q_2, \mu)) \wedge \\ \quad (\exists Q, k \geq 0.\ \mathsf{commit\_}(P, \mathsf{true}, Q) \wedge \mathsf{send\_all'}(Q, k, m \cdot \mu)\end{array}$$

To terminate, the program has to finish the phase where it is allowed to drop requests (represented by predicate $\mathsf{send\_all''}$) and enter the phase where it responds to each request (represented by predicate $\mathsf{send\_all'}$); it needs to signal this by performing the *ghost I/O action* $\mathsf{commit}(\mathsf{true})$.[6]

Other persistence properties (i.e. properties of the form $\Diamond\Box p$) can be encoded similarly.

3 A Programming Language with I/O

We present our approach in the context of a simplified ML-like programming language with support for I/O. Its grammar is as follows:

$$\begin{array}{l} \qquad\qquad v \in \mathit{Vals}, x \in \mathit{Vars}, t \in \mathit{IOPrims} \\ e \in \mathit{Exprs} ::= v \mid x \\ c \in \mathit{Cmds} ::= e \mid t(e) \mid \mathbf{if}\ e = e\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{let}\ x = c\ \mathbf{in}\ c \mid \mathbf{loop}\ c \end{array}$$

We assume a set *Vals* of values, *Vars* of program variables, and *IOPrims* of *I/O primitives*.[7] We assume *Vals* contains at least the unit value ().

We define $c; c' = \mathbf{let}\ x = c\ \mathbf{in}\ c'$ where x does not appear in c'.

We define the I/O actions $\alpha \in \mathit{IOActions} ::= t(v, v)$; in $t(v, v')$, we call v the *argument* and v' the *result*. We coinductively define the set of traces $\tau \in \mathit{Traces} ::= \mathsf{Div} \mid \mathsf{Ret}(v) \mid t(v, v) \cdot \tau$. A trace ending in Div is called a *diverging trace*; it represents the behavior where the program eventually neither terminates nor performs I/O. A trace ending in $\mathsf{Ret}(v)$ is a finite trace; it represents the behavior where the program eventually terminates with result value v.

We define concatenation of traces coinductively as follows:

$$\mathsf{Ret}(v);_v \tau = \tau \qquad \mathsf{Div};_v \tau = \mathsf{Div} \qquad \frac{\tau;_v \tau' = \tau''}{t(v', v'') \cdot \tau;_v \tau' = t(v', v'') \cdot \tau''}$$

We define the language's semantics by means of a big-step relation $c \Downarrow \tau$, which relates a command c to a trace τ. We define the relation coinductively by

[6] The commit *ghost commands* are inserted into the program text for verification purposes only. Since they do not have any observable effect, any properties proven about the ghost-instrumented program hold also for the original program (also known as the *erased* program).

[7] In the examples, we assume $\mathsf{send}, \mathsf{recv}, \mathsf{beep}, \mathsf{log}, \mathsf{heartbeat}, \mathsf{commit} \in \mathit{IOPrims}$.

means of the following rules:

$$v \Downarrow \mathsf{Ret}(v) \qquad \frac{c \Downarrow \tau}{\mathbf{if}\ v = v\ \mathbf{then}\ c\ \mathbf{else}\ c' \Downarrow \tau} \qquad \frac{v \neq v' \qquad c' \Downarrow \tau}{\mathbf{if}\ v = v'\ \mathbf{then}\ c\ \mathbf{else}\ c' \Downarrow \tau}$$

$$\frac{c \Downarrow \tau \qquad c'[v/x] \Downarrow \tau' \qquad \tau ;_v \tau' = \tau''}{\mathbf{let}\ x = c\ \mathbf{in}\ c' \Downarrow \tau''} \qquad \frac{c; \mathbf{loop}\ c \Downarrow \tau}{\mathbf{loop}\ c \Downarrow \tau} \qquad t(v) \Downarrow t(v, v') \cdot \mathsf{Ret}(v')$$

Notice that we have $\mathbf{loop}\ () \Downarrow \mathsf{Div}$. We also have $\mathbf{loop}\ () \Downarrow \tau$ for any other trace τ as well; this imprecision is harmless, since we will be proving liveness.

3.1 Liveness properties

If we define $c_{\mathsf{echo}} = \mathbf{loop}\ (\mathbf{let}\ \mathsf{msg} = \mathsf{recv}(())\ \mathbf{in}\ \mathsf{send}(\mathsf{msg}))$, we can state the responsiveness property of the echo server as follows:

$$\forall \tau, \mu.\ c_{\mathsf{echo}} \Downarrow \tau \wedge \tau|_{\mathsf{recv}} \preceq \mu \Rightarrow \forall m \in \mu.\ \mathsf{send}(m, ()) \in \tau$$

where we define $\tau|_{\mathsf{recv}} \preceq \mu$ coinductively as follows:

$$\mathsf{Div}|_{\mathsf{recv}} \preceq \mu \qquad \mathsf{Ret}(v)|_{\mathsf{recv}} \preceq \mu \qquad \frac{\tau|_{\mathsf{recv}} \preceq \mu}{(\mathsf{recv}(\_, m) \cdot \tau)|_{\mathsf{recv}} \preceq m \cdot \mu}$$

$$\frac{\tau|_{\mathsf{recv}} \preceq \mu}{(\mathsf{send}(\_, ()) \cdot \tau)|_{\mathsf{recv}} \preceq \mu}$$

To make the link with temporal logic: this is more or less equivalent to

$$\forall \tau.\ c_{\mathsf{echo}} \Downarrow \tau \Rightarrow \tau \vDash \Box\Diamond(\exists m.\ \mathsf{recv}(\_, m)) \wedge \Box(\forall m.\ \mathsf{recv}(\_, m) \Rightarrow \Diamond\mathsf{send}(m, \_))$$

(It is equivalent if we ignore the possibility of messages being sent before they are received.)

In the next section, we define a separation logic for modularly verifying properties such as this one, and argue formally that it is indeed adequate for this purpose.

4 A Program Logic for I/O Liveness

4.1 Exit actions and exit traces

We use the notation $\dot{v} \in \mathit{Vals} \cup \{\top\}$ to denote either a program value or the special *exit value* $\top$. An *exit action* $\dot{\alpha} = t(v, \dot{v})$ is like an action except that its result may be the exit value. An *exit trace* $\dot{\tau}$ is a finite sequence of exit actions. We say an exit action is *exiting* if its result is the exit value.

The I/O safety logic from our earlier work [8] proves that a program's partial traces are prefixes of a given set of traces. A total logic (such as [5]) proves that a

program's traces end with $\mathsf{Ret}(\_)$, or, equivalently, that they start with a partial trace that ends with $\mathsf{Ret}(\_)$. By combining and slightly generalizing these, we can obtain a logic whose correctness judgments imply that each of a program's traces starts with one of the partial traces from a given set of *exit traces*, i.e. partial traces that, as soon as an exiting I/O action is performed, cause the program to be considered "terminated". Examples of such *exit sets* are: the set of partial traces T_1 where the program has responded to the first request, the set of partial traces T_2 where the program has responded to the second request, etc. By proving a universally quantified correctness judgment, we can obtain that a program's traces start with a partial trace from T_1 *and* with a partial trace from T_2, etc., that is, that the program responds to the first request *and* to the second request, etc., allowing us to conclude that the program satisfies the responsiveness property.

In the remainder of this section, we elaborate this approach.

4.2 Petri nets

The I/O safety logic is based on the idea that the program must own particular resources in order to be allowed to perform a given I/O action. When using the logic, there is no need to specify the particular nature of those resources. However, for the sake of proving adequacy of the logic, we do need to introduce a particular ontology of resources. For this purpose, we here use a particular type of *Petri nets*. Indeed, these resources serve very much like *tokens* in a Petri net. If, in a Petri net, there is a token in each *pre-place* of a *transition*, the transition can *fire*, which removes one token from each pre-place and adds one to each *post-place*. This, in turn, can enable a new transition, and so on. If we label the transitions by I/O actions, we obtain that a *marking* $V \in \mathcal{P} \to \mathbb{N}$ of a Petri net (which maps each place to the number of tokens present at that place) defines a set of traces.

Specifically, we will be labeling transitions by exit actions, so that a marking defines a set of exit traces.

Let $\mathcal{P}$ be a set of places. We use p and q to range over places. We consider Petri nets where the set of transitions N is a subset of the set $\mathcal{N}$ defined as

$$\chi \in \mathcal{N} ::= t(V_{\mathsf{pre}}, v, \dot{v}, V_{\mathsf{post}}) \mid \mathbf{noop}(V_{\mathsf{pre}}, V_{\mathsf{post}})$$

where V_{pre} and V_{post} are multisets of places, called the *pre-places* and *post-places* of the transition, respectively.

A Petri net defines a labeled step relation $\to$ and a corresponding labeled reachability relation $\twoheadrightarrow$ on markings:

$$\frac{t(V_{\mathsf{pre}}, v, \dot{v}, V_{\mathsf{post}}) \in N}{V \uplus V_{\mathsf{pre}} \overset{t(v,\dot{v})}{\to} V \uplus V_{\mathsf{post}}} \qquad \frac{\mathbf{noop}(V_{\mathsf{pre}}, V_{\mathsf{post}}) \in N}{V \uplus V_{\mathsf{pre}} \overset{\epsilon}{\to} V \uplus V_{\mathsf{post}}} \qquad V \overset{\epsilon}{\twoheadrightarrow} V$$

$$\frac{V \overset{\dot{\tau}}{\to} V' \qquad V' \overset{\dot{\tau}'}{\twoheadrightarrow} V''}{V \overset{\dot{\tau}\cdot\dot{\tau}'}{\twoheadrightarrow} V''}$$

where $V \uplus V' = \lambda p.\, V(p) + V'(p)$. We define $\mathsf{Traces}_N(V) = \{\dot{\tau} \mid \exists V'.\, V \xrightarrow{\dot{\tau}} V'\}$.

4.3 Assertions, correctness judgments, view shifts

We define the set of *assertions* semantically as the set of sets of markings. That is, an assertion describes a marking. An assertion $\mathbf{tokens}(V)$ describes a marking that includes V: $\mathbf{tokens}(V) = \{V' \mid \forall p.\, V'(p) \geq V(p)\}$. The separating conjunction $P * P'$ describes a marking that can be split into one that satisfies P and one that satisfies P': $P * P' = \{V \uplus V' \mid V \in P \wedge V' \in P'\}$.

We define the meaning of a correctness judgment $\{P\}\ c\ \{Q\}$, where precondition P is an assertion, c is a command, and postcondition Q maps a value to an assertion, as follows:

$$\{P\}\ c\ \{Q\} \Leftrightarrow \forall V, \tau.\, V \in P \wedge c \Downarrow \tau \Rightarrow \mathsf{safe}(\tau, \mathsf{Traces}_N(V), Q)$$

where $\mathsf{safe}(\tau, T, Q)$ is defined inductively by the following rules:

$$\frac{\exists V \in Q(v).\, \mathsf{Traces}_N(V) \subseteq T}{\mathsf{safe}(\mathsf{Ret}(v), T, Q)} \qquad \frac{t(v, \top) \in T}{\mathsf{safe}(t(v, v') \cdot \tau, T, Q)}$$

$$\frac{t(v, v'') \in T \qquad v' \neq v''}{\mathsf{safe}(t(v, v') \cdot \tau, T, Q)} \qquad \frac{t(v, v') \in T \qquad \mathsf{safe}(\tau, \{\dot{\tau}' \mid t(v, v') \cdot \dot{\tau}' \in T\}, Q)}{\mathsf{safe}(t(v, v') \cdot \tau, T, Q)}$$

The set T can be seen as a *specification* that expresses safety and liveness properties of the program, as well as assumptions about the environment. Notice that infinite or diverging traces are safe only if they contain an input that conflicts with the specification (i.e. the environment assumptions are violated) or perform an action that corresponds to an exiting action of the specification.

Informally, $\{P\}\ c\ \{Q\}$ states that for every marking V that satisfies P, assuming that the environment provides only inputs allowed by the traces of V, command c performs only outputs allowed by the traces of V and terminates, either by performing an exiting action, or by returning with a result v and a marking V' such that $V' \in Q(v)$.

We say that P view-shifts to Q, written as $P \Rrightarrow Q$, if $\forall V \in P.\, \exists V' \in Q.\, \mathsf{Traces}_N(V') \subseteq \mathsf{Traces}_N(V)$. That is, we can replace the current marking by another one that is equivalent or more restrictive in terms of the program outputs it allows.[8]

[8] Note: reducing the set of traces does not strengthen the assumptions on the environment, because if two traces of a specification make conflicting assumptions about environment behavior, the resulting assumption is the *conjunction* of these, i.e. False.

4.4 Proof rules

Given these definitions, the following proof rules are admissible:

$$\{Q(v)\}\ v\ \{Q\} \qquad \frac{\{P \wedge v = v'\}\ c\ \{Q\} \qquad \{P \wedge v \neq v'\}\ c'\ \{Q\}}{\{P\}\ \mathbf{if}\ v = v'\ \mathbf{then}\ c\ \mathbf{else}\ c'\ \{Q\}}$$

$$\frac{\{P\}\ c\ \{Q\} \qquad \forall v.\ \{Q(v)\}\ c'[v/x]\ \{R\}}{\{P\}\ \mathbf{let}\ x = c\ \mathbf{in}\ c'\ \{R\}} \qquad \frac{\forall n.\ \{P_n\}\ c\ \{\exists n' < n.\ P_{n'}\}}{\{P_m\}\ \mathbf{loop}\ c\ \{\mathsf{False}\}}$$

$$\{\mathbf{tokens}(V_{\mathsf{pre}}) \wedge t(V_{\mathsf{pre}}, v, \dot{v}, V_{\mathsf{post}}) \in N\}\ t(v)\ \{\mathbf{tokens}(V_{\mathsf{post}}) \wedge \dot{v} \neq \top \wedge \mathsf{res} = \dot{v}\}$$

$$\frac{P \Rrightarrow P' \qquad \{P'\}\ c\ \{Q\} \qquad Q \Rrightarrow Q'}{\{P\}\ c\ \{Q'\}} \qquad \frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

$$\frac{\forall i \in I.\ \{P_i\}\ c\ \{Q\}}{\{\exists i \in I.\ P_i\}\ c\ \{Q\}} \qquad \frac{P \Rightarrow Q}{P \Rrightarrow Q} \qquad \frac{P \Rrightarrow Q}{P * R \Rrightarrow Q * R}$$

$$\frac{\mathbf{noop}(V_{\mathsf{pre}}, V_{\mathsf{post}}) \in N}{\mathbf{tokens}(V_{\mathsf{pre}}) \Rrightarrow \mathbf{tokens}(V_{\mathsf{post}})} \qquad \frac{P \Rrightarrow P' \qquad P' \Rrightarrow P''}{P \Rrightarrow P''}$$

We sometimes write postconditions as assertions with a free variable res.

4.5 Abstract nested Hoare triples notation

We define the abstract nested Hoare triple notation $t\_(P, v, v', Q)$ as follows:

$$\begin{array}{l} t\_(P, v, v', Q) = \\ \quad P \Rrightarrow \exists V_{\mathsf{pre}}, V_{\mathsf{post}}.\ \mathbf{tokens}(V_{\mathsf{pre}}) \wedge \\ \qquad (t(V_{\mathsf{pre}}, v, v', V_{\mathsf{post}}) \in N \wedge \mathbf{tokens}(V_{\mathsf{post}}) \Rrightarrow Q \vee t(V_{\mathsf{pre}}, v, \top, V_{\mathsf{post}}) \in N) \end{array}$$

It follows that we have $\{P \wedge t\_(P, v, v', Q)\}\ t(v)\ \{Q \wedge \mathsf{res} = v'\}$.

4.6 Example: simple responsiveness

We show now how we can use our logic to verify the responsiveness property of the echo server, repeated here:

$$\forall \tau, \mu.\ c_{\mathsf{echo}} \Downarrow \tau \wedge \tau|_{\mathsf{recv}} \preceq \mu \Rightarrow \forall m \in \mu.\ \mathsf{send}(m, ()) \in \tau$$

First of all, we fix μ. Again, we assume no message appears more than once in μ.

It is sufficient to prove

$$\forall k \geq 0, \tau.\ c_{\mathsf{echo}} \Downarrow \tau \wedge \tau|_{\mathsf{recv}} \preceq \mu \Rightarrow \mathsf{send}(\mu_k, ()) \in \tau$$

where μ_k denotes the message at index k in μ.

We fix k. We define exit set T as follows:

$$T = \{\dot{\tau} \mid \dot{\tau}|_{\mathsf{recv}} \preceq \mu \wedge (\forall \mathsf{send}(v, \dot{v}) \in \dot{\tau}.\ v = \mu_k \vee \dot{v} = ())\}$$

where we define $\dot{\tau}|_{\mathsf{recv}} \preceq \mu$ as follows:

$$\epsilon|_{\mathsf{recv}} \preceq \mu \qquad \frac{\dot{\tau}|_{\mathsf{recv}} \preceq \mu}{(\mathsf{recv}(\_, m) \cdot \dot{\tau})|_{\mathsf{recv}} \preceq m \cdot \mu} \qquad \frac{\dot{v} \in \{(), \top\} \qquad \dot{\tau}|_{\mathsf{recv}} \preceq \mu}{(\mathsf{send}(\_, \dot{v}) \cdot \tau)|_{\mathsf{recv}} \preceq \mu}$$

It is sufficient to prove

$$\forall \tau.\ c_{\mathsf{echo}} \Downarrow \tau \Rightarrow \mathsf{safe}(\tau, T, \mathsf{False})$$

where μ_k denotes the message at index k in μ.

Indeed, we can prove, by induction on the derivation of $\mathsf{safe}(\tau, T, \mathsf{False})$, that for any τ, if $\mathsf{safe}(\tau, T, \mathsf{False})$ and $\tau|_{\mathsf{recv}} \preceq \mu$, then $\mathsf{send}(\mu_k, ()) \in \tau$.

We construct a Petri net $(\mathcal{P}, N)$ and a marking V such that $\mathsf{Traces}_N(V) \subseteq T$. We define

$$\mathcal{P} = \{\mathsf{recv}_i \mid 0 \leq i\} \cup \{\mathsf{send}_i \mid 0 \leq i\}$$

and

$$\begin{aligned} N = {} & \{\mathsf{recv}(\{\!|\mathsf{recv}_i|\!\}, (), \mu_i, \{\!|\mathsf{recv}_{i+1}|\!\}) \mid i \geq 0\} \cup \\ & \{\mathsf{send}(\{\!|\mathsf{send}_i|\!\}, \mu_i, (), \mathbf{0}) \mid i \geq 0 \wedge i \neq k\} \cup \{\mathsf{send}(\{\!|\mathsf{send}_k|\!\}, \mu_k, \top, \mathbf{0})\} \end{aligned}$$

and

$$V = \lambda p.\ \begin{cases} 1 \text{ if } p = \mathsf{recv}_0 \\ 1 \text{ if } \exists i.\ p = \mathsf{send}_i \\ 0 \text{ otherwise} \end{cases}$$

where $\{\!|p|\!\}$ denotes the singleton multiset with a single token at p, and $\mathbf{0}$ denotes the empty multiset.

It is easy to check that indeed $\mathsf{Traces}_N(V) \subseteq T$, and that $V \in P$ where $P = \exists P_\mathsf{r}, P_\mathsf{s}.\ P_\mathsf{r} * P_\mathsf{s} \wedge \mathsf{recv\_stream}(P_\mathsf{r}, \mu) \wedge \mathsf{send\_all}'(P_\mathsf{s}, k, \mu)$.

Then, by the meaning of correctness judgments, we have that $\{P\}\ c_{\mathsf{echo}}\ \{\mathsf{False}\}$ implies the goal.

4.7 General responsiveness

The adequacy of the verification approach for general responsiveness properties presented in §2 follows directly from the adequacy for simple responsiveness properties, combined with the observation that $\{P_1 \vee P_2\}\ c\ \{Q\}$ implies both $\{P_1\}\ c\ \{Q\}$ and $\{P_2\}\ c\ \{Q\}$. Therefore, given a successful verification of the program, each constituent simple responsiveness property can then be established separately as above.

4.8 Example: persistence

We wish to prove that c'_{echo}, defined as

$$\begin{array}{l} c'_{\mathsf{echo}} = \\ \quad \mathsf{recv}(); \mathbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \mathbf{in}\ \mathsf{send}(\mathsf{msg}); \mathsf{recv}(); \\ \quad \mathbf{loop}\ (\mathbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \mathbf{in}\ \mathsf{send}(\mathsf{msg})) \end{array}$$

drops only finitely many messages, i.e. that it eventually always responds. Formally:

$$\forall \mu, \tau.\ c'_{\mathsf{echo}} \Downarrow \tau \wedge \tau|_{\mathsf{recv}} \preceq \mu \Rightarrow \exists k_0 \geq 0.\ \forall k \geq k_0.\ \mathsf{send}(\mu_k, ()) \in \tau$$

We consider a *ghost-instrumented* version $\tilde{c}'_{\mathsf{echo}}$ of c'_{echo}, defined as follows:

$$\begin{array}{l} \tilde{c}'_{\mathsf{echo}} = \\ \quad \mathsf{commit}(\mathsf{false}); \mathsf{commit}(\mathsf{false}); \mathsf{commit}(\mathsf{false}); \\ \quad \mathsf{recv}(); \mathbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \mathbf{in}\ \mathsf{send}(\mathsf{msg}); \mathsf{recv}(); \\ \quad \mathsf{commit}(\mathsf{true}); \\ \quad \mathbf{loop}\ (\mathbf{let}\ \mathsf{msg} = \mathsf{recv}()\ \mathbf{in}\ \mathsf{send}(\mathsf{msg})) \end{array}$$

Obviously, it is sufficient to prove

$$\forall \mu, \tau.\ \tilde{c}'_{\mathsf{echo}} \Downarrow \tau \wedge \mathsf{erasure}(\tau)|_{\mathsf{recv}} \preceq \mu \Rightarrow \exists k_0 \geq 0.\ \forall k \geq k_0.\ \mathsf{send}(\mu_k, ()) \in \tau$$

where $\mathsf{erasure}(\tau)$ removes the commit actions (replacing an infinite sequence of commit actions with Div). We define $\#\tau$ as the index of the first $\mathsf{commit}(\mathsf{true}, \_)$ action in τ, or 0 if it does not contain such an action. It is sufficient to prove

$$\forall \mu, \tau.\ \tilde{c}'_{\mathsf{echo}} \Downarrow \tau \wedge \mathsf{erasure}(\tau)|_{\mathsf{recv}} \preceq \mu \Rightarrow \forall k \geq 0.\ \mathsf{send}(\mu_{\#\tau+k}, ()) \in \tau$$

We fix μ and k and define exit set T as follows:

$$T = \left\{ \dot{\tau} \;\middle|\; \begin{array}{l} \mathsf{erasure}(\dot{\tau})|_{\mathsf{recv}} \preceq \mu \wedge \\ (\forall \mathsf{send}(v, \dot{v}) \in \dot{\tau}.\ \mathsf{commit}(\mathsf{true}, \_) \in \dot{\tau} \wedge v = \mu_{\#\dot{\tau}+k} \vee \dot{v} = ()) \end{array} \right\}$$

It is sufficient to prove

$$\forall \tau.\ \tilde{c}'_{\mathsf{echo}} \Downarrow \tau \Rightarrow \mathsf{safe}(\tau, T, \mathsf{False})$$

Indeed, one can again show, by induction on the derivation of $\mathsf{safe}(\tau, T, \mathsf{False})$ that for all τ, if $\mathsf{safe}(\tau, T, \mathsf{False})$ and $\mathsf{erasure}(\tau)|_{\mathsf{recv}} \preceq \mu$, then $\mathsf{send}(\mu_{\#\tau+k}, ()) \in \tau$.

The remainder of this example proceeds as above: we construct a Petri net and a marking that satisfies the precondition we used for verifying the persistence example in §2 and whose traces are included in T. Successful verification then implies the goal.

5 Related work

We are not aware of existing work on Hoare logics for verifying liveness properties of the I/O behavior of programs. To the best of our knowledge, most approaches for verifying liveness properties of the I/O behavior of systems have so far been based on a representation of the system as some kind of a state machine, or a set of interacting processes or state machines, rather than a program. In these approaches, the liveness properties of interest are very often specified using temporal logic [2].

Even when it comes to Hoare logics for liveness properties of other aspects of program execution, there is very little existing work. We are aware of only two lines of work. Boström and Müller [1] verify that blocked threads in a multithreaded program are always eventually unblocked. In ongoing work, D'Osualdo *et al.* [3] verify termination under a fair scheduler of multithreaded programs that involve synchronization based on busy-waiting.

For a discussion of related work on verifying safety of I/O behavior and on verifying program termination, we refer to our earlier work [9, 5].

6 Conclusion

We presented a Hoare logic-based approach for the specification and verification of liveness properties of the I/O behavior of program modules. Our approach is based on the idea of reducing the problem to a termination verification problem and then applying existing approaches for I/O safety verification and termination verification. Our approach can be applied straightforwardly in existing verification tools that support separation logic, higher-order predicates, and termination verification, such as our VeriFast tool [12].

In this paper, we considered a very simple programming language with no dynamically allocated memory, higher-order functions or dynamic method binding, or concurrency. However, we believe the ideas of this paper can be integrated straightforwardly into the separation logic for total correctness of multithreaded object-oriented programs from our earlier work [5] to verify liveness properties of the I/O behavior of a large class of realistic programs.

A limitation of this earlier work, however, is that it does not support *busy waiting*, a common practice in programs for multiprocessor machines. In recent work [10], we propose a logic for verifying termination under fair scheduling of programs where threads busy-wait for other threads to abruptly terminate the program. By combining that logic with the ideas from this paper, one can obtain a logic for verifying responsiveness of a multithreaded server where one thread receives requests and another responds to them: the first thread can be seen as busy-waiting for the second thread to terminate the program by responding to the k'th request.

We have not addressed the question of completeness: does our approach support all possible liveness properties? Chang *et al.* [2] claim that any temporal logic formula is equivalent to a general reactivity formula. However, their setting is not quite the same as ours, as evidenced by the fact that we need to use

ghost I/O operations to support persistence properties, whereas in their framework persistence properties are subsumed by reactivity properties. A thorough investigation of this question is future work.

References

1. Boström, P., Müller, P.: Modular verification of finite blocking in non-terminating programs. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic. LIPIcs, vol. 37, pp. 639–663. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015)
2. Chang, E., Manna, Z., Pnueli, A.: The safety-progress classification. In: Bauer, F.L., Brauer, W., Schwichtenberg, H. (eds.) Logic and Algebra of Specification. pp. 143–202. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
3. D'Osualdo, E., Farzan, A., Gardner, P., Sutherland, J.: TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs. CoRR **abs/1901.05750** (2019)
4. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
5. Jacobs, B., Bosnacki, D., Kuiper, R.: Modular termination verification of single-threaded and multithreaded programs. ACM Trans. Program. Lang. Syst. **40**(3), 12:1–12:59 (2018)
6. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018)
7. Penninckx, W., Jacobs, B., Piessens, F.: Sound, modular and compositional verification of the input/output behavior of programs. In: Vitek, J. (ed.) Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9032, pp. 158–182. Springer (2015)
8. Penninckx, W., Timany, A., Jacobs, B.: Abstract I/O specification. CoRR **abs/1901.10541** (2019)
9. Penninckx, W., Timany, A., Jacobs, B.: Specifying I/O using abstract nested Hoare triples in separation logic. In: Proceedings of the 21st Workshop on Formal Techniques for Java-like Programs. FTfJP '19, Association for Computing Machinery, New York, NY, USA (2019)
10. Reinhard, T., Timany, A., Jacobs, B.: A separation logic to verify termination of busy-waiting for abrupt program exit. In: FTfJP (2020), to appear.
11. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74. IEEE Computer Society (2002)
12. Vogels, F., Jacobs, B., Piessens, F.: Featherweight VeriFast. Logical Methods in Computer Science **11**(3) (2015)