Rely/Guarantee Reasoning for Asynchronous Programs

Ivan Gavran[1], Filip Niksic[1], Aditya Kanade[2], Rupak Majumdar[1], and Viktor Vafeiadis[1]

1 **Max Planck Institute for Software Systems (MPI-SWS), Germany**
2 **Indian Institute of Science, Bangalore, India**

Abstract

Asynchronous programming has become ubiquitous in smartphone and web application development, as well as in the development of server-side and system applications. Many of the uses of asynchrony can be modeled by extending programming languages with *asynchronous procedure calls* – procedures not executed immediately, but stored and selected for execution at a later point by a non-deterministic scheduler. Asynchronous calls induce a flow of control that is difficult to reason about, which in turn makes formal verification of asynchronous programs challenging. In response, we take a *rely/guarantee* approach: Each asynchronous procedure is verified separately with respect to its rely and guarantee predicates; the correctness of the whole program then follows from the natural conditions the rely/guarantee predicates have to satisfy. In this way, the verification of asynchronous programs is modularly decomposed into the more usual verification of sequential programs with synchronous calls. For the sequential program verification we use Hoare-style deductive reasoning, which we demonstrate on several simplified examples. These examples were inspired from programs written in C using the popular Libevent library; they are manually annotated and verified within the state-of-the-art Frama-C platform.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases asynchronous programs, rely/guarantee reasoning

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2015.483

1 Introduction

Asynchronous programming is a technique to efficiently manage multiple concurrent interactions with the environment. Application development environments for smartphone applications provide asynchronous APIs; client-side web programming with Javascript and AJAX, high-performance systems software (e.g., nginx, Chromium, Tor), as well as embedded systems, all make extensive use of asynchronous calls. By breaking long-running tasks into individual procedures and posting callbacks that are triggered when background processing completes, asynchronous programs enable resource-efficient, low-latency management of concurrent requests.

In its usual implementation, the underlying programming system exposes an *asynchronous* procedure call construct (either in the language or using a library), which allows the programmer to post a procedure for execution in the future when a certain event occurs. An event scheduler manages asynchronously posted procedures. When the corresponding event occurs, the scheduler picks the associated procedure and runs it to completion. These procedures are sequential code, possibly with recursion, and can post further asynchronous procedures.

26th International Conference on Concurrency Theory (CONCUR 2015).
Editors: Luca Aceto and David de Frutos Escrig; pp. 483–496

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Unfortunately, while asynchronous programs can be very efficient, the manual management of resources and asynchronous procedures can make programming in this model quite difficult. The natural control flow of a task is obscured and the programmer must ensure correct behavior for all possible orderings of external events. Specifically, the global state of the program can change between the time an asynchronous procedure is posted and the time the scheduler picks and runs it.

In recent years, a number of automatic static analyses for asynchronous programs have been proposed. The main theoretical result is the equivalence between an abstract model of asynchronous programs with Boolean variables and Petri nets, thus showing that safety and liveness verification problems are decidable for this model [14, 9, 6]. In practice, this equivalence has been the basis for several automatic tools [11, 3]. Unfortunately, existing tools still fall short of verifying "real" asynchronous programs. First, existing tools often ignore important features such as passing data as arguments to asynchronous calls or heap data structures in order to find a Boolean abstraction. Second, existing tools perform a *global* coverability analysis of the Petri net equivalent to the abstracted program. Despite the use of sophisticated heuristics, global coverability analysis tools scale poorly, especially when there are many Boolean variables [4].

In this paper, we provide a modular proof system for asynchronous programs based on rely/guarantee reasoning [10, 1, 5]. For each asynchronous procedure, we use a ("local") precondition and a postcondition, similar to modular proofs for sequential recursive programs. In addition, we use a *rely* and a *guarantee*. Intuitively, the rely is the assumption about the global state that the procedure makes about all other procedures that may happen in parallel with it. The guarantee is what the procedure ensures about the global state. In addition to predicates over global state, our rules also use predicates *posted* and *pending* that track if a task was posted asynchronously in the current call stack, or if it is pending, respectively. With these additional predicates, our modular proof rules are extremely simple:

- running each task from its precondition establishes its guarantee and postcondition;
- the rely of each task must preserve its precondition;
- if a procedure posts task h and does not cancel it, it establishes the precondition of h at the end of its execution; and finally,
- the guarantee of each task that may run between the time h is posted and h is executed establishes the rely of h.

We prove soundness of these rules, based on an invariant that ensures that if a procedure is pending, then its precondition remains valid at every schedule point.

It is possible to simulate asynchronous programs using multi-threaded programs and vice versa [12]. Thus, in principle, rely/guarantee reasoning for multi-threaded programs [10, 5, 8, 7] – extended with rules for dynamic thread creation – could be used to reason about asynchronous programs. However, by focusing on the specific concurrency model, we can deal with programming features such as recursive tasks, as well as more advanced asynchronous programming features such as deletion of tasks. To support these features, the reduction to multi-threaded programs would add additional data structures to the program, losing the structure of the program. Thus, "compiling" to threads, while theoretically possible, is not likely to preserve the local, and often simple, reason why a program is correct.

We have implemented our proof system on top of the Frama-C framework and show modular proofs of partial correctness on two asynchronous programs written in C using the Libevent library. The programs are simple but realistic examples of common asynchronous idioms. We show that one can verify these idioms by constructing "small" modular proofs, using generic rely and guarantee predicates that can be automatically derived from preconditions.

```
1  struct client_state { ... };
2
3  async main() {
4    // prepare a socket for
5    // incoming connections
6    int socket = prepare_socket();
7    post accept(socket);
8  }
9
10 //@ requires \valid(s);
11 async read(struct client_state *s) {
12   if (/* s->fd ready */) {
13     // receive a chunk and store a
14     // rot13'd version into s->buffer
15     post write(s);
16     post read(s);
17   }
18   else { // connection closed
19     delete write(s);
20     free(s);
21   }
22 }
```

```
23 async accept(int socket) {
24   if (/* socket ready */) {
25     struct client_state *s = malloc(...);
26     s->fd = accept_connection(socket);
27     // initialize s->buffer
28     post read(s);
29   }
30   post accept(socket);
31 }
32
33 //@ requires \valid(s);
34 async write(struct client_state *s) {
35   if (/* s->fd ready */) {
36     // send a chunk
37     if (/* there's more to send */)
38       post write(s);
39   }
40   else { // connection closed
41     delete read(s);
42     free(s);
43   }
44 }
```

Figure 1 Snippet of the ROT13 program. In this and the subsequent figures, parts of the code are omitted and replaced by comments for brevity.

Moreover, reasoning about asynchronous programs can be effectively reduced to modular reasoning about sequential programs, for which sophisticated verification environments already exist.

2 Main Idea

Asynchronous Programs. Figure 1 shows a version of the ROT13 server from the Libevent manual [13]. The server receives input strings from a client, and sends back the strings obfuscated using ROT13. The execution starts in the procedure `main`, which prepares a non-blocking socket for incoming connections, and passes it to the procedure `accept` via an asynchronous call. The asynchronous call, denoted by the keyword **`post`**, schedules `accept` for later execution. In general, a procedure marked with the keyword **`async`** can be *posted* with some arguments. The arguments determine an instance of the procedure; the instance is stored in a set of pending instances. After some pending instance finishes executing, a scheduler non-deterministically selects the next one and executes it completely. In case of the ROT13 server, after `main` is done, the scheduler selects the single pending instance of `accept`. `accept` checks whether a client connection is waiting to be accepted; if so, it accepts the connection and allocates memory consisting of a socket and a buffer for communication with the client. The allocated memory, addressed by the pointer `s`, is then asynchronously passed to the procedure `read`. Finally, regardless of whether the connection has been accepted or not, `accept` re-posts itself to reschedule itself for processing any upcoming connections.

While the client connection is open, the corresponding memory allocated by `accept` is handled by a reader-writer pair: the reader (`read`) receives an input string and stores an obfuscated version of it into the buffer. It then posts the writer (`write`), which sends the content of the buffer back to the client. An interesting thing happens when the client disconnects, which can happen during the execution of either the reader or the writer. When one of the procedures notices that the connection has been closed, it releases the allocated memory. However, the procedure does not know whether an instance of its counterpart is still pending; if it is, it would try to access the deallocated memory. To make sure this does not happen, before releasing the memory, the procedure deletes (keyword **`delete`**) the potentially pending instance of its counterpart.

The example shows that control structures for asynchronous programs can be complex: tasks may post other tasks for later processing, arguments can be passed on to asynchronously posted tasks, and an unbounded number of tasks can be pending at a time.

Safety Verification. We would like to verify that every memory access in this program is safe; that is, we want to verify that both `read` and `write` can safely dereference the pointer `s`. We assume this property is expressed by the predicate valid(`s`). We write valid(`s`) as a precondition for `read` and `write` in lines 10 and 33.

Let us focus only on `read`. Its precondition clearly holds at each call site: it holds at line 28 since the memory addressed by `s` has just been freshly allocated (for simplicity, we assume malloc succeeds), and it holds at line 16 assuming `read`'s precondition holds. However, between the point `read` is posted and the point it is executed, two different things might invalidate its precondition. First, the caller may still have code to execute after the call. Second, there may be pending instances of `accept`, `read`, and `write` concurrent with `read(s)` that get executed before `read(s)` and deallocate the memory addressed by `s`.

To deal with the code of `read`'s callers, also referred to as `read`'s *parents*, we introduce predicates $\text{posted}_r(s)$ and $\text{pending}_r(s)$. (We also introduce a pair of predicates for every other asynchronous procedure, namely `main`, `accept`, and `write`.) Predicate $\text{posted}_r(s)$ holds if and only if `read(`s`)` has been posted during the execution of the current asynchronous procedure (and not deleted). Predicate $\text{pending}_r(s)$ holds if and only if `read(`s`)` is in the set of pending instances. Note that if an asynchronous procedure posts and afterwards deletes `read(`s`)`, neither $\text{posted}_r(s)$ nor $\text{pending}_r(s)$ will hold. Using the introduced predicates, `read`'s parents can now express the following parent-child postcondition:

$$\forall s.\ \text{posted}_r(s) \implies \text{valid}(s)\,. \tag{PC}$$

Informally, this postcondition says that every instance of `read` that has been posted during the execution of the procedure, and that has not been deleted afterwards, has been posted with the argument s that is valid, i.e., that can be safely dereferenced.

Rely/Guarantee. To deal with the procedures whose instances are concurrent with `read`, also referred to as `read`'s *concurrent siblings*, we employ rely/guarantee reasoning. We introduce a *rely condition* for `read`:

$$\forall s.\ \text{pending}'_r(s) \wedge \text{pending}_r(s) \wedge \text{valid}'(s) \implies \text{valid}(s)\,, \tag{R}$$

where the primed versions of the predicates denote their truth at the beginning of execution of a procedure. Informally, the rely condition says that if `read(`s`)` was pending with a valid s when a concurrent sibling started executing, and `read(`s`)` is still pending at the end of that execution, then s is still valid. In other words, `read` *relies* on the assumption that its precondition is preserved by its concurrent siblings. Any of `read`'s concurrent siblings, namely `accept`, `read`, and `write`, must *guarantee* `read`'s rely condition. This is achieved by ensuring the concurrent siblings' postconditions imply the rely condition.

As shown formally in the next section, the rely/guarantee conditions ensure the following global invariant:

$$\forall s.\ \text{pending}_r(s) \implies \text{valid}(s)\,. \tag{I}$$

This invariant holds at the beginning of execution of every asynchronous procedure. Consequently, `read`'s precondition holds at the moment `read` is executed.

The benefit of the described approach is that it abstracts away reasoning about the non-deterministic scheduler and the order in which it dispatches pending instances. We only need to verify that each asynchronous procedure satisfies its postcondition. This can be achieved using a sequential verification tool (e.g., Frama-C in our case).

A natural question to ask is why we have two predicates – posted_r and pending_r – when it seems that pending_r alone should be sufficient? If the global invariant (I) is what we are after, why not just make it a precondition and a postcondition of every asynchronous procedure? While this is sufficient, in order to prove (I) as a postcondition of an asynchronous procedure, one would need to do a case split, and separately consider `read`'s instances posted during the execution of the procedure, and instances that had been pending before the procedure started executing. In the first case, the procedure *knows* why `read`'s precondition holds, while in the second case it *assumes* that `read`'s precondition holds. By having the special predicate posted_r, we can make this case split explicit: the two separate cases correspond to the parent-child condition (PC) and the rely condition (R). The asynchronous procedures assume only their original preconditions, making the overall reasoning more local.

3 Technical Details

We formalize the rely/guarantee proof rules on SLAP, a *s*imple *l*anguage with *a*synchronous *p*rocedure calls. The main result of the paper is Theorem 1, which says that in order to verify a program with asynchronous procedure calls, it suffices to modularly verify each procedure of a sequential program.

3.1 SLAP: Syntax and Semantics

Syntax. A SLAP program consists of a set of program variables Var, a set of procedure names H, a subset $AH \subseteq H$ of *asynchronous* procedures including a special procedure *main*, and a mapping Π from procedure names in H to commands from a set $Cmds$ of commands (defined below).

We distinguish between global variables, denoted by $GVar$, and local variables, denoted by $LVar$. Local variables also include parameters of procedures. We also introduce a set of *logical variables*, disjoint from the program variables, which are used for constructing quantified formulas. We write x, y, z for single variables, and $\vec{x}, \vec{y}, \vec{z}$ for vectors of variables. We usually use $x, \vec{x}$ to denote global variables, and $y, z, \vec{y}, \vec{z}$ to denote local variables. We use the same letters for logical variables; this should not cause confusion.

We use a disjoint, *primed*, copy of the set of variables Var. Primed variables are used to denote the state of the program at the beginning of execution of a procedure, while the unprimed variables denote the current state. Logical variables do not have primed counterparts, although we often abuse notation and write them primed.

Variables are used to construct *expressions*. We leave the exact syntax of expressions unspecified. We just distinguish between Boolean expressions (usually denoted by B), and all expressions (usually denoted by E).

The set of *commands*, denoted $Cmds$, is generated by the grammar:

$$\begin{aligned} C \quad ::= \quad & x := E \mid \mathbf{assume}(B) \mid \mathbf{assert}(B) \mid g(E_1, \ldots, E_k) \\ & \mid \mathbf{post}\ h(E_1, \ldots, E_k) \mid \mathbf{delete}\ h(E_1, \ldots, E_k) \mid \mathbf{enter}\ h \mid \mathbf{exit}\ h \\ & \mid C_1; C_2 \mid C_1 + C_2 \mid C^* \end{aligned}$$

The atomic commands are assignments ($x := E$), assumptions ($\mathbf{assume}(B)$), assertions ($\mathbf{assert}(B)$), and synchronous calls ($g(\ldots)$ for $g \in H \setminus AH$), as in a sequential imperative

$$
\begin{array}{ll}
\sigma', \sigma, o', o, p', p \models \text{posted}'_h(E_1, \ldots, E_k) & \text{iff } (h, \bot[y_1 \mapsto [\![E_1]\!]_{\sigma',\sigma}, \ldots, y_k \mapsto [\![E_k]\!]_{\sigma',\sigma}]) \in o' \\
\sigma', \sigma, o', o, p', p \models \text{posted}_h(E_1, \ldots, E_k) & \text{iff } (h, \bot[y_1 \mapsto [\![E_1]\!]_{\sigma',\sigma}, \ldots, y_k \mapsto [\![E_k]\!]_{\sigma',\sigma}]) \in o \\
\sigma', \sigma, o', o, p', p \models \text{pending}'_h(E_1, \ldots, E_k) & \text{iff } (h, \bot[y_1 \mapsto [\![E_1]\!]_{\sigma',\sigma}, \ldots, y_k \mapsto [\![E_k]\!]_{\sigma',\sigma}]) \in p' \\
\sigma', \sigma, o', o, p', p \models \text{pending}_h(E_1, \ldots, E_k) & \text{iff } (h, \bot[y_1 \mapsto [\![E_1]\!]_{\sigma',\sigma}, \ldots, y_k \mapsto [\![E_k]\!]_{\sigma',\sigma}]) \in p \\
\sigma', \sigma, o', o, p', p \models B & \text{iff } [\![B]\!]_{\sigma',\sigma} = \text{true}
\end{array}
$$

Figure 2 Semantics of atomic formulas.

language, together with the additional commands for asynchronous calls (**post** $h(\ldots)$ for $h \in AH$), deletions of pending instances (**delete** $h(\ldots)$), and special commands **enter** h and **exit** h marking the entrance to and exit from a procedure. Starting with the atomic commands, complex commands are built using sequential composition (;), non-deterministic choice (+), and iteration (*).

Most of the commands in the language have their expected semantics. The command **post** $h(E_1, \ldots, E_k)$ posts an asynchronous call of procedure $h \in AH$ with arguments $E_1, \ldots, E_k$ for future execution, and **delete** $h(E_1, \ldots, E_k)$ deletes the pending occurrence of the asynchronously posted procedure h with arguments $E_1, \ldots, E_k$ if it exists. The **enter** h and **exit** h commands are there for a technical reason: they mark the entry and exit of procedure h. We assume that the command $\Pi(h)$ of each procedure h starts with **enter** h, ends with **exit** h, and that those two commands do not appear anywhere in between.

Formulas are generated as first order formulas whose atomic predicates are Boolean expressions as well as the predicates posted'_h, posted_h, $\text{pending}'_h$, and pending_h, for each asynchronous procedure $h \in AH$. Intuitively, posted_h is used for reasoning about the accumulated asynchronous calls to h made during the execution of a single asynchronous procedure, and pending_h is used for reasoning about the pending asynchronous calls to h, not necessarily made during the execution of a single asynchronous procedure. Like program variables, these predicates have the corresponding primed versions, used for reasoning about the state at the beginning of execution of a procedure. For every formula F we write F' for the formula obtained by replacing all unprimed occurrences of program variables, as well as the predicates posted_h and pending_g, by their primed counterparts.

Semantics. Assuming there is a set of values *Val*, a function $\sigma\colon Var \to Val$ is called a *valuation*. We use notation $\sigma_G := \sigma|_{GVar}$ for the restriction of σ to global variables, and $\sigma_L := \sigma|_{LVar}$ for the restriction of σ to local variables. We call the restrictions *global valuation* and *local valuation*, respectively. We use notation $\sigma[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$ to denote a valuation that differs from σ only in variables $x_1, \ldots, x_k$, which are mapped to values $v_1, \ldots, v_k$. We assume there is a special value $\bot \in Val$ denoting a non-initialized value. We also use $\bot$ to denote a constant valuation that maps every variable to $\bot$.

Given valuations σ' and σ, we denote the value of an expression E by $[\![E]\!]_{\sigma',\sigma}$. Here, σ' is used for evaluating the primed variables (the values at the beginning of execution of the current procedure), and σ is used for evaluating the unprimed variables (the current values).

Next, we define a *configuration* $\Phi = (s, \sigma_G, o, p)$ of a SLAP program, where s is a stack that keeps track of synchronous calls, σ_G is a valuation that describes the global state, o is a set of instances asynchronously posted within the current asynchronous procedure, and p is a set of pending instances. Stack s holds *stack frames* – tuples of the form $(C, \sigma', \sigma_L, o', p')$, where C is the command that needs to be executed in the current stack frame, σ' is the

$$\text{[Enter]}\ \frac{}{((\textbf{enter}\ h; C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s ((C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p)}$$

$$\text{[Exit]}\ \frac{}{((\textbf{exit}\ h, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s (s, \sigma_G, o, p)}$$

$$\text{[Assume]}\ \frac{\sigma', \sigma, o', o, p', p \models F}{((\textbf{assume}(F); C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s ((C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p)}$$

$$\text{[Assert OK]}\ \frac{\sigma', \sigma, o', o, p', p \models F}{((\textbf{assert}(F); C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s ((C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p)}$$

$$\text{[Assert Wrong]}\ \frac{\sigma', \sigma, o', o, p', p \not\models F}{((\textbf{assert}(F); C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s \textbf{wrong}}$$

$$\text{[Assign]}\ \frac{\rho = \sigma[x \mapsto [\![E]\!]_{\sigma', \sigma}]}{((x := E; C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s ((C, \sigma', \rho_L, o', p') :: s, \rho_G, o, p)}$$

$$\text{[Choice]}\ \frac{i \in \{1, 2\}}{((C_1 + C_2; C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s ((C_i; C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p)}$$

$$\text{[Loop Skip]}\ \frac{}{((C^*; C', \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s ((C', \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p)}$$

$$\text{[Loop Step]}\ \frac{}{((C^*; C', \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s ((C; C^*; C', \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p)}$$

$$\text{[Sync Call]}\ \frac{h \in H \setminus AH \qquad \rho = \bot[y_1 \mapsto [\![E_1]\!]_{\sigma', \sigma}, \ldots, y_k \mapsto [\![E_k]\!]_{\sigma', \sigma}]}{((h(E_1, \ldots, E_k); C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s ((\Pi(h), \sigma_G \cup \rho_L, \rho_L, o, p) :: (C; \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p)}$$

$$\text{[Async Call]}\ \frac{h \in AH \qquad \rho = \bot[y_1 \mapsto [\![E_1]\!]_{\sigma', \sigma}, \ldots, y_k \mapsto [\![E_k]\!]_{\sigma', \sigma}] \qquad q = o \cup \{(h, \rho_L)\} \qquad r = p \cup \{(h, \rho_L)\}}{((\textbf{post}\ h(E_1, \ldots, E_k); C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s ((C, \sigma', \sigma_L, o', p') :: s, \sigma_G, q, r)}$$

$$\text{[Async Delete]}\ \frac{h \in AH \qquad \rho = \bot[y_1 \mapsto [\![E_1]\!]_{\sigma', \sigma}, \ldots, y_k \mapsto [\![E_k]\!]_{\sigma', \sigma}] \qquad q = o \setminus \{(h, \rho_L)\} \qquad r = p \setminus \{(h, \rho_L)\}}{((\textbf{delete}\ h(E_1, \ldots, E_k); C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p) \xrightarrow{\Pi}_s ((C, \sigma', \sigma_L, o', p') :: s, \sigma_G, q, r)}$$

Figure 3 Semantics of SLAP – sequential part.

valuation at the beginning of execution in the current stack frame, σ_L is the valuation that describes the current local state, and o' and p' are sets of posted and pending instances at the beginning of execution in the current stack frame. Sets o, p, o', and p' hold pairs of the form (h, σ_L), where h is the posted or pending procedure, and σ_L is a valuation that describes the values passed to h. We use notation $t :: ts$ to denote a stack consisting of a head t and a tail ts, and $\emptyset$ to denote both an empty stack and an empty set. Apart from configurations of the form (s, σ_G, o, p), which are part of the correct program execution, there is also a special configuration **wrong**.

At this point we have introduced all the concepts and terminology needed to give semantics to SLAP programs. First, the semantics of formulas is given in terms of valuations σ', σ, and sets o', o, p', p. The semantics of atomic formulas is shown in Figure 2, and the semantics of complex formulas is defined inductively. We write $\sigma', \sigma, o', o, p', p \models F$ if F holds with respect to $\sigma', \sigma, o', o, p', p$. We also write $\Phi \models F$ if $\Phi = ((C, \sigma', \sigma_L, o', p') :: s, \sigma_G, o, p)$ and $\sigma', \sigma, o', o, p', p \models F$. If $\Phi = (\emptyset, \sigma_G, o, p)$, the truth of F containing local or primed variables, or the predicates posted'_h and $\text{pending}'_h$ is undefined. Finally, **wrong** $\models F$ for any F.

Next, we define the sequential semantics of a SLAP program $\Pi\colon H \to \mathit{Cmds}$ as a transition system over configurations. The rules that define the sequential transition relation

$$\frac{\Phi \xrightarrow{\Pi}_s \Phi'}{\Phi \xrightarrow{\Pi}_a \Phi'}\ [\textsc{Extend}] \qquad \frac{h \in AH \qquad (h, \sigma_L) \in p \qquad r = p \setminus \{(h, \sigma_L)\}}{(\emptyset, \sigma_G, o, p) \xrightarrow{\Pi}_a ((\Pi(h), \sigma, \sigma_L, \emptyset, r) :: \emptyset, \sigma_G, \emptyset, r)}\ [\textsc{Dispatch}]$$

Figure 4 Semantics of SLAP – asynchronous part.

$$\forall \vec{x}', \vec{x}, \vec{y}', \vec{y}.\ P'_h(\vec{x}', \vec{y}') \wedge Q_h(\vec{x}', \vec{y}', \vec{x}, \vec{y}) \implies G_h(\vec{x}', \vec{x}) \tag{1}$$

$$\forall \vec{x}', \vec{x}, \vec{y}, \vec{z}', \vec{z}.\ \text{posted}_h(\vec{y}) \wedge P'_g(\vec{x}', \vec{z}') \wedge Q_g(\vec{x}', \vec{z}', \vec{x}, \vec{z}) \implies P_h(\vec{x}, \vec{y}), \tag{2}$$
where $g \in \text{parents}(h)$

$$\forall \vec{x}', \vec{x}, \vec{y}.\ \text{pending}'_h(\vec{y}) \wedge \text{pending}_h(\vec{y}) \wedge P'_h(\vec{x}', \vec{y}) \wedge R_h(\vec{x}', \vec{x}) \implies P_h(\vec{x}, \vec{y}) \tag{3}$$

$$\forall \vec{x}', \vec{x}.\ G_g(\vec{x}', \vec{x}) \implies R_h(\vec{x}', \vec{x}), \tag{4}$$
where $g \in \text{siblings}(h)$

Figure 5 Rely/guarantee conditions. Variables $\vec{x}', \vec{x}$ represent global variables, and variables $\vec{y}', \vec{y}, \vec{z}', \vec{z}$ represent parameters.

$\xrightarrow{\Pi}_s$ are given in Figure 3. The asynchronous semantics extends the sequential semantics by integrating the behavior of the non-deterministic scheduler. The rules that define the asynchronous transition relation $\xrightarrow{\Pi}_a$ are given in Figure 4.

Note that we are modeling the pool of pending procedure instances using a set. Therefore, posting an instance that is already pending has no effect. An alternative would be to use a multiset and count the number of pending instances. We chose the first option for three reasons. First, it corresponds to the semantics of Libevent's function `event_add()`. Second, it simplifies the semantics of deleting procedure instances. And third, one can always simulate the counting semantics by extending the procedure with an extra parameter that acts as a counter.

3.2 Rely/Guarantee Decomposition

In order to reason about an asynchronous program $\Pi\colon H \to \mathit{Cmds}$ modularly, for each of its asynchronous procedures $h \in AH$ we require a specification in terms of formulas P_h, R_h, G_h, and Q_h. Formulas P_h and Q_h are h's precondition and postcondition in the standard sense: P_h is a formula over Var that is supposed to hold at the beginning of h's execution, while Q_h is a formula over $\mathit{Var}' \cup \mathit{Var}$ that is supposed to hold at the end of h's execution. Predicates R_h and G_h are formulas over $\mathit{GVar}' \cup \mathit{GVar}$, and they represent the procedure's rely and guarantee conditions. Intuitively, R_h tells what h *relies on* about the change of the global state, while G_h tells what h *guarantees about* the change of the global state. We require that the predicates posted'_g, posted_g, $\text{pending}'_g$, and pending_g appear in the specification only in negative positions. Furthermore, in P_h we allow only the unprimed predicates, and we require $P_{main} \equiv \text{true}$.

We will say that the specification $(P_h, R_h, G_h, Q_h)_{h \in AH}$ is a *rely/guarantee decomposition* of Π if the four conditions in Figure 5 are satisfied. Condition (1) requires procedure h to establish its guarantee. Condition (2) requires that each parent of h, i.e. each asynchronous procedure that posts h, establishes h's precondition. Condition (3) is the *stability condition*: it requires the rely predicate R_h to be strong enough to preserve preconditions of procedure

h's pending instances. Finally, condition (4) requires that h's rely predicate is guaranteed by each of h's concurrent siblings, i.e. asynchronous procedures that may be executed between the point when h is posted and the point when h itself is executed. Together, conditions (1)–(4) imply the following lifecycle of an asynchronous procedure instance: once posted by its parent, its precondition is established. Before it is executed, its precondition is preserved by its concurrent siblings. When the procedure instance is finally executed, its precondition holds.

Given a rely/guarantee decomposition $(P_h, R_h, G_h, Q_h)_{h \in AH}$ of a program $\Pi\colon H \to \mathit{Cmds}$, we define a transformation of commands $\tau\colon \mathit{Cmds} \to \mathit{Cmds}$ that inserts assumptions and assertions of preconditions and postconditions at the right places:

$$\begin{aligned}
\tau(\textbf{enter } h) &:= \textbf{enter } h; \textbf{assume}(P_h), && \text{for } h \in AH,\\
\tau(\textbf{exit } h) &:= \textbf{assert}(Q_h); \textbf{exit } h, && \text{for } h \in AH,\\
\tau(C_1; C_2) &:= \tau(C_1); \tau(C_2)\\
\tau(C_1 + C_2) &:= \tau(C_1) + \tau(C_2)\\
\tau(C^*) &:= \tau(C)^*\\
\tau(C) &:= \tau(C), && \text{otherwise.}
\end{aligned}$$

The definition of τ is naturally lifted to configurations (s, σ_G, o, p) and **wrong**: in case of (s, σ_G, o, p), τ transforms all commands that await execution on the stack s, while $\tau(\textbf{wrong}) = \textbf{wrong}$.

Given a program $\Pi\colon H \to \mathit{Cmds}$, we will say that Π is *sequentially correct* with respect to a rely/guarantee decomposition $(P_h, R_h, G_h, Q_h)_{h \in AH}$ if for every valuation $\sigma\colon \mathit{Var} \to \mathit{Val}$, every set of pending instances p, and every asynchronous procedure $h \in AH$ we have

$$((\tau(\Pi(h)), \sigma, \sigma_L, \emptyset, p) :: \emptyset, \sigma_G, \emptyset, p) \xrightarrow{\tau \circ \Pi}\!\!\!\!\!\!\nearrow\,{}^{*}_{s}\ \textbf{wrong}\,.$$

We will say that Π is *correct* if we have

$$(\emptyset, \bot_G, \emptyset, \{(\mathit{main}, \bot_L)\}) \xrightarrow{\Pi}\!\!\!\!\!\!\nearrow\,{}^{*}_{a}\ \textbf{wrong}\,.$$

With these definitions, the soundness of rely/guarantee reasoning is stated in the following theorem.

▶ **Theorem 1.** *Let $\Pi\colon H \to \mathit{Cmds}$ be an asynchronous program. If Π is sequentially correct with respect to a rely/guarantee decomposition $(P_h, R_h, G_h, Q_h)_{h \in AH}$, then it is correct.*

The proof of Theorem 1 is based on four technical results.

▶ **Lemma 2.** *Let $\Pi\colon H \to \mathit{Cmds}$ be an asynchronous program, and let Φ_0, Φ be configurations of Π such that $\Phi_0 = (\emptyset, \bot_G, \emptyset, \{(\mathit{main}, \bot_L)\})$, $\Phi = (s, \sigma_G, o, p)$ and $\Phi_0 \xrightarrow{\Pi}{}^{*}_{a} \Phi$.*

1. *For every stack frame $(C, \sigma', \sigma_L, o', p') \in s$, $p \subseteq o \cup p'$.*
2. *If s is non-empty, then for every $h \in AH$,*

$$\Phi \models \forall \vec{y}.\ \mathrm{pending}_h(\vec{y}) \implies (\mathrm{posted}_h(\vec{y}) \vee \mathrm{pending}'_h(\vec{y}))\,.$$

Proof. The first statement is proved by induction on the length of the trace. A straightforward check shows that every rule preserves the invariant $p \subseteq o \cup p'$. The second statement is a direct corollary of the first statement. ◀

The next lemma states that preconditions of pending instances hold at each dispatch point. Thus, it formalizes the discussion in Section 2, where the invariant (I) is found to hold at each dispatch point.

▶ **Lemma 3.** *Let* $\Pi\colon H \to \mathit{Cmds}$ *be an asynchronous program with a rely/guarantee decomposition* $(P_h, R_h, G_h, Q_h)_{h\in AH}$*. If* $(\emptyset, \bot_G, \emptyset, \{(\mathit{main}, \bot_L)\}) \xrightarrow{\tau\circ\Pi}{}^*_a (\emptyset, \sigma_G, o, p)$*, then for every* $g \in AH$ *we have* $\sigma_G, \sigma_G, \emptyset, o, p, p \models \forall \vec{y}.\ \mathrm{pending}_g(\vec{y}) \implies P_g(\vec{x}, \vec{y})$*.*

Proof. By induction on the number of applications of the rule [Dispatch], using the rely/guarantee conditions (1)–(4) and the invariant from Lemma 2(2). ◀

▶ **Corollary 4.** *Let* $\Pi\colon H \to \mathit{Cmds}$ *be an asynchronous program with a rely/guarantee decomposition* $(P_h, R_h, G_h, Q_h)_{h\in AH}$*, and let* $(\emptyset, \bot_G, \emptyset, \{(\mathit{main}, \bot_L)\}) \xrightarrow{\tau\circ\Pi}{}^+_a \tau(\Phi)$*, with the last step being a dispatch of procedure* $h \in AH$*. Then,* $\tau(\Phi) \models P_h(\vec{x}, \vec{y})$*.*

Proof. From Lemma 3, we know that the state just before the dispatch satisfies $P_h(\vec{x}, \vec{y})$ because h is pending in that state. Our conclusion, therefore, follows because $P_h(\vec{x}, \vec{y})$ can contain predicates posted_g and $\mathrm{pending}_g$ only in negative positions, and the rule [Dispatch] makes the sets of posted and pending instances smaller. ◀

▶ **Lemma 5.** *Let* $\Pi\colon H \to \mathit{Cmds}$ *be an asynchronous program with a rely/guarantee decomposition* $(P_h, R_h, G_h, Q_h)_{h\in AH}$*, and let* $\Phi_0, \Phi'_0, \ldots, \Phi_k, \Phi'_k$ *be configurations of* Π *such that* $\Phi_0 = (\emptyset, \bot_G, \emptyset, \{(\mathit{main}, \bot_L)\})$ *and*

$$\Phi_0 \xrightarrow{\Pi}{}^*_s \Phi'_0 \xrightarrow{\Pi}_a \Phi_1 \xrightarrow{\Pi}{}^*_s \Phi'_1 \xrightarrow{\Pi}_a \cdots \xrightarrow{\Pi}_a \Phi_k \xrightarrow{\Pi}{}^*_s \Phi'_k\,,$$

with all of the asynchronous steps being taken according to the rule [Dispatch]*. Either:*

1. $\forall i \in 0..k.\ \tau(\Phi_i) \xrightarrow{\tau\circ\Pi}{}^*_s \tau(\Phi'_i)$*, or*
2. $\exists i \in 0..k.\ \tau(\Phi_i) \xrightarrow{\tau\circ\Pi}{}^*_s$ **wrong**

Proof. By induction on the length of the trace. A straightforward inspection of the rules in Figure 3 shows that each step of the original trace can be simulated by one or two steps of the transformed trace. The only non-trivial point is showing that the inserted assume statements always hold, which follows from Corollary 4. ◀

Proof of Theorem 1. By contraposition and application of Lemma 5. ◀

Notice that the rely/guarantee decomposition uses two relations: parents and siblings. Formally, $g \in \mathrm{parents}(h)$ if there is a reachable configuration obtained by executing **exit** g in which h is in the set of posted instances. Similarly, $g \in \mathrm{siblings}(h)$ if there is a reachable configuration in which both g and h are in the set of pending instances. While these relations are hard to compute precisely, Theorem 1 holds when we use any over-approximation of these relations. A trivial over-approximation of both relations is the set AH of all asynchronous procedures. In Section 4, we discuss a better approximation obtained through simple static analysis.

4 Rely/Guarantee in Practice

4.1 Implementation for Libevent

We focused on C programs that use the Libevent library[1]. Libevent is an event notification library whose main purpose is to unify OS-specific mechanisms for handling events that occur on file descriptors. From this it also extends to handling signals and timeout events. The

[1] `http://libevent.org/`

$$R_h(\vec{x}', \vec{x}) \equiv \forall \vec{y}.\ \text{pending}'_h(\vec{y}) \wedge \text{pending}_h(\vec{y}) \wedge P'_h(\vec{x}', \vec{y}) \implies P_h(\vec{x}, \vec{y})$$

$$G_h(\vec{x}', \vec{x}) \equiv \bigwedge_{g \in \text{siblings}(h)} R_g(\vec{x}', \vec{x})$$

$$Q_h(\vec{x}', \vec{z}', \vec{x}, \vec{z}) \equiv G_h(\vec{x}', \vec{x}) \wedge \bigwedge_{g \in \text{children}(h)} \forall \vec{y}.\ \text{posted}_g(\vec{y}) \implies P_g(\vec{x}, \vec{y})$$

Figure 6 Generic rely/guarantee predicates.

library is used in asynchronous applications such as the *Chromium* web browser, *Memchached*, *SNTP*, and *Tor*.

We abstract away the details of the events by assuming their handlers are dispatched non-deterministically, instead of when the events actually occur. Thus, registering an event handler for a specific event corresponds to calling the handler asynchronously in our model.

Even with this abstraction, Libevent remains too complex for reasoning about directly. Therefore, we hide it behind a much simpler interface that corresponds to SLAP: For each asynchronous procedure h, we provide two (synchronous) functions called `post_`h and `delete_`h, with the same parameters as h. As the prefixes suggest, these functions are used for posting and deleting h's instances. With these functions, the C code we are analyzing directly resembles the code of the ROT13 server in Figure 1; the difference is that instead of the keywords **`post`** and **`delete`**, we use the functions with the corresponding prefixes.

We implemented the rely/guarantee rules on top of the Frama-C verification platform [2]. We use ACSL, Frama-C's specification language, which is expressive enough to encode the predicates posted and pending, with their state being maintained using ghost code. The specification is a fairly straightforward encoding of the semantics of SLAP. After the transformation that inserts appropriate preconditions and postconditions (along with the necessary ghost code), we use Frama-C's WP (*weakest-precondition*) plugin to generate verification conditions that are discharged by Z3.

In order to over-estimate the sets of parents and concurrent siblings, we manually perform a simple static analysis. In this analysis, we ignore deletes, and only look at posts. For each procedure h we perform a 0–1–ω abstraction of the number of asynchronous procedures' instances posted at each location. Specifically, at h's exit point this gives us an abstracted number of instances of each procedure posted by h. From this information, we can directly construct sets of children, or equivalently parents. Furthermore, if h posts f and g, then $f \in \text{siblings}(g)$ and $g \in \text{siblings}(f)$. Also, if h posts more than one instance of g, then $g \in \text{siblings}(g)$. We use these two facts to bootstrap the following recursion that computes siblings: if $f \in \text{siblings}(g)$, then $f' \in \text{siblings}(g)$ and $f \in \text{siblings}(g')$ for every $f' \in \text{children}(f)$ and $g' \in \text{children}(g)$.

An archive with the verified programs can be found at the URL: `http://www.mpi-sws.org/~fniksic/concur2015/rely-guarantee.tar.gz`.

4.2 Generic Rely/Guarantee Predicates

Instead of asking the programmer to manually specify rely/guarantee predicates and post-conditions, and then checking that they satisfy the rely/guarantee conditions (1)–(4), we can use generic predicates shown in Figure 6. These predicates trivially satisfy conditions (1)–(4); in fact, they are the weakest predicates to do so.

Note that the generic predicates, while convenient, might not be sufficient for verifying correctness of all programs. The reason is that the proof obligations for the sequential

```
1  struct device {
2    int owner;
3    // ...
4  } dev;
5
6  async main() {
7    dev.owner = 0;
8    int socket = prepare_socket();
9    post listen(socket);
10 }
11
12 /*@ requires id > 0;
13   @ requires global_invariant_write;
14   @*/
15 async new_client(int id, int fd) {
16   if (dev.owner > 0)
17     post new_client(id, fd);
18   else {
19     dev.owner = id;
20     post write(id, fd);
21   }
22 }
```

```
23 async listen(int socket) {
24   if (/* socket ready */) {
25     int id = new_client_id();
26     int fd = accept_connection(socket);
27     post new_client(id, fd);
28   }
29   post listen(socket);
30 }
31
32 /*@ requires
33   @   id > 0 ∧
34   @   dev.owner = id ∧
35   @   ∀ int id1, int fd1;
36   @     pending_write(id1, fd1)
37   @       ⇒ id = id1 ∧ fd = fd1
38   @*/
39 async write(int id, int fd) {
40   if (transfer(fd, dev))
41     post write(id, fd);
42   else // write complete
43     dev.owner = 0;
44 }
```

Figure 7 Snippet of the Race program.

programs obtained by applying the transformation τ may not be provable. The generic predicates are sufficient for the ROT13 server from Section 2, where the preconditions are: $P_{\texttt{main}} \equiv P_{\texttt{accept}} \equiv \text{true}$, and $P_{\texttt{read}} \equiv P_{\texttt{write}} \equiv \text{valid}(\texttt{s})$. Plugging these preconditions into the generic predicates from Figure 6 gives rise to proof obligations that can be discharged by Z3. However, the generic predicates are not sufficient for the program we discuss next.

Consider the Race program [9] in Figure 7. Initially, procedure `main` sets up a global resource called `dev` to which multiple clients will transfer data by setting the `dev.owner` flag to zero. `main` then prepares a socket and posts the procedure `listen` to listen to client connections. `listen` checks whether the socket is ready; if so, it accepts the connection to get a file descriptor `fd`, and generates a unique positive `id` for the client. It then passes `id` and `fd` to the procedure `new_client`. `new_client` checks whether the device is currently owned by some client (`dev.owner>0`); if so, it re-posts itself. If the device is free (`dev.owner==0`), `new_client` takes the ownership for the client identified by `id` and posts the procedure `write` that performs the transfer of the client's data. `write` operates in multiple steps, re-posting itself until the transfer is done. At the end, it releases the device by setting `dev.owner` back to zero.

In this example, since multiple clients are trying to non-atomically write to a single shared resource, the important property is mutual exclusion: there should always be at most one pending instance `write(id, fd)`, and if there is one, `dev.owner` should be set to `id`. We encode this property as a precondition $P_{\texttt{write}}$ to `write` in lines 32–37.

To ensure mutual exclusion, procedure `new_client` assumes `id>0` as its precondition in line 12. However, this precondition is not sufficiently strong for `new_client` to establish `write`'s generic rely predicate. This is due to `write`'s precondition including assumptions about other of `write`'s pending instances. However, `new_client` can establish `write`'s rely if it additionally assumes `write`'s global invariant $\forall id, fd.\ \text{pending}_{\texttt{write}}(id, fd) \implies P_{\texttt{write}}(id, fd)$ (line 13).

In order to justify `new_client`'s additional assumption, and show that there is no hidden circular reasoning, we show that we can weaken the rely/guarantee conditions (1) and (2). Indeed, Lemma 3 still holds if we replace conditions (1) and (2) with the following weaker

versions:

$$\forall \vec{x}', \vec{x}, \vec{y}', \vec{y}. \bigwedge_{f \in AH} \left(\forall \vec{z}.\ \text{pending}'_f(\vec{z}) \implies P'_f(\vec{x}', \vec{z})\right) \tag{1'}$$
$$\wedge\, P'_h(\vec{x}', \vec{y}') \wedge Q_h(\vec{x}', \vec{y}', \vec{x}, \vec{y}) \implies G_h(\vec{x}', \vec{x})$$

$$\forall \vec{x}', \vec{x}, \vec{y}, \vec{z}', \vec{z}. \bigwedge_{f \in AH} \left(\forall \vec{u}.\ \text{pending}'_f(\vec{u}) \implies P'_f(\vec{x}', \vec{u})\right) \tag{2'}$$
$$\wedge\, \text{posted}_h(\vec{y}) \wedge P'_g(\vec{x}', \vec{z}') \wedge Q_g(\vec{x}', \vec{z}', \vec{x}, \vec{z}) \implies P_h(\vec{x}, \vec{y})\,,$$

where $g \in \text{parents}(h)$

The generic postconditions Q_h can now be weakened as follows.

$$Q_h(\vec{x}', \vec{z}', \vec{x}, \vec{z}) \equiv \bigwedge_{f \in AH} \left(\forall \vec{y}.\ \text{pending}'_f(\vec{y}) \implies P'_f(\vec{x}', \vec{y})\right)$$
$$\implies \left(G_h(\vec{x}', \vec{x}) \wedge \bigwedge_{g \in \text{children}(h)} \forall \vec{y}.\ \text{posted}_g(\vec{y}) \implies P_g(\vec{x}, \vec{y})\right)$$

This allows asynchronous procedures to freely assume any of the global invariants ensured by Lemma 3 if it helps them to establish their guarantees. Even though at first glance such additional assumptions might seem vacuous, the Race example shows this is not the case.

4.3 Limitations

In practice, Frama-C and the WP plugin have limitations that are orthogonal to the rely/guarantee approach. One limitation is WP's lack of support for dynamic memory allocation. In fact, in order to verify the ROT13 server, we were not able to use Frama-C's built-in predicate `\valid`. Instead, we had to specify our own validity predicate and use corresponding dedicated malloc and free functions. Generalizing such an approach to more complicated programs is infeasible, as our custom memory model does not integrate well with the built-in one. A related limitation is restricted reasoning about inductive data structures. While Frama-C's specification language ACSL supports inductive predicates, the WP plugin does not fully support them. Moreover, reasoning about even the simplest inductive data structures such as linked lists may require separation predicates that are beyond the expressive power of ACSL. Our rely/guarantee rules work "modulo" a sequential verifier, so better handling of these limitations will allow reasoning about more complex asynchronous programs.

Acknowledgements. This research was sponsored in part by the EC FP7 FET project ADVENT (308830) and an ERC Synergy Award ("ImPACT").

References

1 M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

2 P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C – A software analysis perspective. In *SEFM 2012*, pages 233–247, 2012.

3 E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Automatic verification of Erlang-style concurrency. In *Proceedings of the 20th Static Analysis Symposium*, SAS'13. Springer-Verlag, 2013.

4 J. Esparza, R. Ledesma-Garza, R. Majumdar, P. Meyer, and F. Niksic. An SMT-based approach to coverability analysis. In *CAV 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 603–619. Springer, 2014.

5 C. Flanagan, S.N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *ESOP 2002*, pages 262–277, 2002.

6 P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1):6, 2012.

7 A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL 11*, pages 331–344. ACM, 2011.

8 T.A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI 2004*, pages 1–13. ACM, 2004.

9 R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL 2007*, pages 339–350. ACM, 2007.

10 C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

11 Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Efficient coverability analysis by proof minimization. In *Proceedings of the 23rd International Conference on Concurrency Theory*, CONCUR'12, pages 500–515. Springer-Verlag, 2012.

12 H.C. Lauer and R.M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

13 N. Mathewson. Fast portable non-blocking network programming with Libevent. `http://www.wangafu.net/~nickm/libevent-book/`. Accessed: 2015-03-12.

14 K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV'06*, volume 4144 of *LNCS*, pages 300–314. Springer, 2006.