A History-based Verification of Distributed Applications

Bruno Langenstein, Andreas Nonnengart, Georg Rock, and Werner Stephan

German Research Center for Artificial Intelligence (DFKI GmbH)
Saarbrücken, Germany
{langenstein,nonnengart,rock,stephan}@dfki.de

Abstract. Safety and security guarantees for individual applications in general depend on assumptions on the given context provided by distributed instances of operating systems, hardware platforms, and other application level programs that are executed on these platforms. The problem for formal approaches is to formalize these assumptions without having to look at the details of the (formal) model of the operating system (including the machines that execute applications).
The work described in this paper presents a modular approach which uses histories of observable events to specify runs of distributed instances of the system. The overall verification approach decomposes the given verification problem into local tasks along the lines of assume-guarantee reasoning. In this paper we focus on this methodology and on its realization in the Verification Support Environment (VSE). We also illustrate the proposed approach with the help of a suitable example, namely the specification and verification of an SMTP server whose implementation makes extensive use of various system calls as e.g. fork and socket commands.

1 Introduction

The theory developed in the following aims at a modular approach for the specification and verification of concurrent systems with heterogeneous components. *Concurrency* typically results from the actual parallel execution of independent systems and the abstraction from a concrete scheduler within the context of a given platform. Like the systems themselves their formal models will consist of various types of components specified by different types of state transition systems. In the composed (global) system the components interact with each other by certain communication mechanisms.

In this paper we consider an instantiation of the general approach which is taken from the context of the Verisoft project where a pervasive formal model of a distributed collection of hardware-software platforms with application level programs running on each of these was established, [1].

Instead of verifying application level programs directly on the Verisoft model, we propose to use traces of *observable events* that according to a given view are attached to steps of computations (or runs) of the distributed system as they have been formally defined in Verisoft. Since for a state in a run we collect all events that have happened so far we call these (finite) lists of events *histories*. The behavior of the global system as well as that of single components is then specified by sets of histories thereby abstracting from the local state spaces of the components. Like an input-output specification for a sequential piece of software sets of histories describe the concurrent, typically non-terminating computation of a global system or component thereof. Event traces defined by a certain view on a given model provide an appropriate interface for an inductive analysis of cryptographic protocols, [2, 3] or an information flow analysis [4].

Our approach is modular in the sense that the task of verifying a history specification against runs of the global system can be decomposed into local verification tasks for its components. Following the general assume-guarantee approach, see [5] for comprehensive discussion of these approaches, for each event specified in a history there is exactly one component which may *violate* the specification at this point. Therefore, our events allow to determine the component considered to be *responsible* for the event. Events in a history a particular component is not responsible for are considered as assumptions of that component w.r.t. its environment.

Each history specification, possibly obtained by a combination of sub-specifications, has to be verified against all components of the overall model. Here we focus on the verification of C0[1] machines that execute (and give meaning to) C0 programs extended by *external calls* that allow to exchange information with the corresponding operating system and, via a network component, with C0 machines that run on different (remote) instances of the operating system. As an example we consider the implementation of an e-mail system consisting of an e-mail client, a (so-called) mail bag, a SMTP-server, and a SMTP-client.

In the next section we summarize the instantiation of the general approach by the *Verisoft model* of distributed systems. As a concrete example in section 3 we provide the *specification of the SMTP server* by a set of histories. In Section 4 we describe the verification of application-level C0 programs, like the implementation of the SMTP server, by means of a transformation to abstract sequential programs that *replay and extend* histories. Section 5 summarizes the described approach and outlines possible future work.

2 Events and Histories

2.1 The Verisoft Model

In the Verisoft project a formal model of an operating system was developed and verified with respect to its implementation, [6]. Application level programs run on (abstract) machines that are part of the model. They interact by a kind of RPC like mechanism and access resources (of the OS) by external calls.

[1] C0 is an imperative C-like programming language with some restrictions compared to standard C. In C0 there are besides others no side effects in expressions and pointers are strictly typed (see [1] for a complete description of C0).

A model of a distributed system consists of instances of (the same) system and an abstract network component connecting them. Each system is identified by a unique *network address* $na \in Na$. It basically consists of two parts. The *simple operating system* provides resources like input-output devices, a file system, and sockets for the communication over the network component. A *process* is identified by a *process identifier* $pid \in Pid$ and the network address of the system instance it is running on. For each process given by $p = mk\_proc(pid, na)$ a machine (interpreter) is part of the system na. In this paper we are interested in applications implemented in C0, the subset of C considered in Verisoft. Hence our processes represent abstract execution mechanisms where the program part of a configuration is a C0 program π.

From point of view of an application programmer the system context consists of all operating systems and the network connecting them. Hence we consider this complete context as a single component in our decomposition. It will be denoted by the constant SOS.

The view we have chosen is depicted in Figure 1.

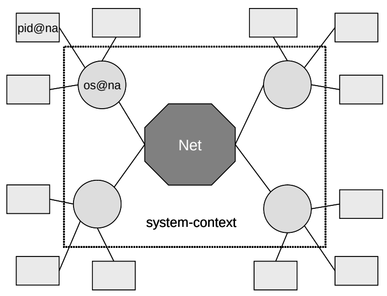

Fig. 1. Verisoft Model

The communication between user programs and the surrounding operating system (instance) is by so called *external calls*. External calls $c_{ext}(\bar{\tau} : \bar{z}, res)$ take the same syntax as ordinary function calls. We use $\bar{\tau}$ as a sequence of value parameters and $\bar{z}, res$ as a sequence of return parameters. Typically the value of res will indicate success or failure. Whenever a system call is reached during the computation of a process p the normal execution as given by the (small step) semantics $R_{C0} \subseteq Conf \times Conf$ is interrupted (stopped) and a request is sent to the corresponding operating system. With these steps of a global computation we associate *events* of the form $mk\_ev(p, SOS, m)$ where the message m encodes the particular call given by c_{ext} and the values of the parameters $(\bar{\tau} : \bar{z})$ in mem'. For a call of socket read $\texttt{socket\_read}(\texttt{sid} : \texttt{length}, \texttt{buffer}, \texttt{ec})$ the corresponding message will by $Sread(sid, length)$ where $Sread$ is a constructor symbol for an abstract data type and $sid, length$ are the values of the programming variables $\texttt{sid}, \texttt{length}$ in mem'. They indicate the socket and the length of the string to be read.

To model the return of external calls the standard C0 machines have to be extended by steps where the resulting configuration is determined by an answer (message) from the corresponding operating system. The (answer) information intended for process p will be written to the return parameters (of the pending call). The event we associate with these steps is of the form $mk\_ev(SOS, p, m)$ where the message m represents the return information. For example a successful call of socket read the message will be $Succ\_sread(length, buffer)$ where $length$

indicates the elements in the fixed length array $buffer$ that have actually been read. These values uniquely determine the values of the result parameters after return of that external call.

2.2 History Specifications

Having defined events for all external calls (and also RPC calls) we may specify global system runs by a set (or unary predicate) H of finite sequences of events. A global System $SOS(\pi_0,,\dots,\pi_{n-1})$ consisting of arbitrary many instances of the operating system with arbitrary many C0-processes executing $\pi_0,,\dots,\pi_{n-1}$.

By $SOS(\pi_0,,\dots,\pi_{n-1}) \models H$ we denote the fact that for each state in a global run of $SOS(\pi_0,,\dots,\pi_{n-1})$ the sequence of events that have happened so far satisfies H. Given an event $e = mk\_ev(s,r,m)$ we say that s (the sender) is *responsible* for that event. In addition we define that e is *relevant* for s as well as the receiver r. By $SOS(\pi_0,,\dots,\pi_{n-1}) \downarrow i \models H$ we denote the fact that no process p executing $\pi_i \in \{\pi_0,,\dots,\pi_{n-1}\}$ violates H first, i.e. in a step p is responsible for. Similarly we use $SOS(\pi_0,,\dots,\pi_{n-1}) \downarrow SOS \models H$ for a projection to the operating systems themselves.

To establish $SOS(\pi_0,,\dots,\pi_{n-1}) \downarrow i \models H$ locally outside the context of a global system we transform π_i into a program $\tilde{\pi}_i$ where the external calls manipulate histories. Altogether $\tilde{\pi}_i$ replays and possibly extends histories.

Using $\tilde{\pi}_i \models H$ for the fact that $\tilde{\pi}_i$ preserves H, soundness of this method (w.r.t. the verification of application level programs) is demonstrated by a kind of *simulation theorem* that allows to conclude that $\tilde{\pi}_i \models H \Rightarrow SOS(\pi_0,,\dots,\pi_{n-1}) \downarrow i \models H$ holds. The simulation theorem is application independent and established by looking at the C0 execution mechanism. As opposed to that $\tilde{\pi}_i \models H$ is concerned with the verification of *individual* programs.

Before we present the specification of the SMTP-server and the verification technique given by the $\tilde{\ }$-transformation we have to discuss special events that provide the binding between programs π_i and processes in a given history.

2.3 Life Cycle Events

Histories cover the whole life cycle of processes. This includes the association of process identifiers with the programs that are executed. This binding takes place upon *creation* of a new process. The lifetime of a process ends by an explicit *termination* event.

In the verification process we are interested in a particular program (text) π. To associate programs with process identifiers we assume a fixed enumeration of programs. In histories $\pi = \pi_i$ is then represented by the constant i.

Create events $\langle SOS, p, Create(i)\rangle$ are *caused* by the corresponding instance of the SOS while the identifier for the new process $p = mk\_proc(na, pid)$ is considered as the recipient of the message indicating the program text π_i to be executed starting with a fixed initial state.

Clone create events $\langle SOS, p, Clone\_Create(p', b)\rangle$ are caused by the corresponding SOS as possible positive reaction to a call of fork. $p = mk\_proc(na, pid)$ indicates the *child process* which *continues* to execute the program of the *parent process* given by p' in the message. $b \in \{0, 1\}$ is a flag indicating access to the terminal. The initial state of p is the state of p' reached before execution of fork.

A process p (executing some π_i) is terminated by *exit* or *kill events*. Exit events are caused by p while kill events are caused by the corresponding instance of the SOS.

A sub-history h_0 of h is called a *thread* of p in h if there is exactly one create event for p in h_0 which is the first event (in h_0) relevant for p. The thread is called open if h_0 does not contain a terminating event for p.

Now in defining $\tilde{\pi}_i \models H$ by a program $\tilde{\pi}(i)$ that replays and (possibly) extends histories h we have to consider all threads in h that execute π_i. Since a given specification H can only be violated if some $h \in H$ is actually extended by $\tilde{\pi}$ we restrict ourselves to open threads executing π_i. It is a simple observation that for each p there is at most one open thread in a given h. Therefore we provide a (guess of) p as an argument to the replay and extend procedure. If there is no open thread of p that executes π_i, then the given history h is delivered unchanged as the result.

Open threads chosen in this way include those where a child p of p' is created by an event $\langle SOS, p, Clone\_Create(p', b)\rangle$ following a call of fork during the execution of π_i. Since we have to know the correct internal state the replay and extend procedure has to start with the first ancestor p'' of p. The computation of this process is initiated by a create event $\langle SOS, p'', Create(i)\rangle$.

In the start of the replay and extend procedure for π_i as well as in the implementation of calls of fork we use the function $anc(i, p, h)$ which computes the sequence of ancestor threads for a given p and h. In case there is no open thread of p in h executing π_i $anc(p, h) = []$. For $anc(p, h) = [h_0, \ldots, h_{n_1}]$, h_{n_1} is the open thread of p in h with $\langle SOS, p, Create(i)\rangle$ or $\langle SOS, p, Clone\_Create(p', b)\rangle$ as first event and h_0 is the first ancestor thread with first element $\langle SOS, p'', Create(i)\rangle$.

3 SMTP-Server

As already mentioned earlier we considered a non-trivial example in the Verisoft context, namely the full implementation (in a C-like language) and the full specification (in terms of histories) of an SMTP-Server as part of an Simple Mail Transfer Scenario. All in all this implementation required about 7.500 lines of code.

The SMTP server listens for connections from SMTP clients. If a connection has been established, it spawns a child process, which inherits the socket granting access to that new connection. The child communicates with the remote SMTP client while obeying the so-called SMTP Protocol. In the meantime, the main SMTP server process listens again for new connections and spawns child processes to handle the session. This behaviour can be formalised by a step by step description of the main process and its child processes. For the formalisation we fix the constant SOS (Simple Operating System) representing the operating system. Any process – and thus the SOS as well – is determined by a network address (the host) and a process id on this host. We assume that – for a given process p – we can access the network address by $get\_na(p)$.

For simplicity we make use of the following definitions: For every history h and process p we define $h \downarrow p$ as the *projection* of the history h on process p. I. e.,

$$\begin{aligned}
() \downarrow p &= () \\
(\langle s, r, m\rangle \circ h) \downarrow p &= \langle s, r, m\rangle \circ h \downarrow p \text{ if s=p or r=p} \\
(\langle s, r, m\rangle \circ h) \downarrow p &= h \downarrow p \text{ otherwise}
\end{aligned}$$

With $h^+ = \{h' \in \mathit{Hist} \mid h' \downarrow p = h\}$ we can describe (for a given history h and an implicitly given process p) the set of histories whose projection on p is just h. Recall that we defined the binary operator $\circ$ on histories as the concatenation of its two arguments. In what follows we also use this operator for the "concatenation" of two history sets: $H_1 \circ H_2 = \{h_1 \circ h_2 \mid h_1 \in H_1, h_2 \in H_2\}$.

The top-level specification (in terms of histories) of the SMTP-Server then looks as follows: we consider the prefix-closure of the set $H_{SMTP\_Server}(p)$ where $H_{\text{SMTP\_Server}}(p) = H_{\text{INIT}}(p) \circ H_{\text{LOOP}}(p, sid)$ for some process p representing the SMTP-Server process and some socket id *sid*.
The history set $H_{\text{INIT}}(p)$ describes the initialization phase of the SMTP-Server. I. e., the SOS first creates the SMTP-process (represented by the message $\mathit{Create}(SMTP\_Server, SOS, 1)$). Then the newly created SMTP-process sends a message to the SOS that it wants a socket to be opened on port 25 (the standard SMTP-port). After the SOS successfully responds with a new socket id the SMTP-Server requests to listen to this new socket. The history set $H_{\text{INIT}}(p)$ is thus easily defined as

$$\begin{aligned}
H_{\text{INIT}}(p) = (&\langle SOS, p, \mathit{Create}(SMTP\_Server, SOS, 1)\rangle \circ \\
&\langle p, SOS, \mathit{Sopen}(25)\rangle \circ \langle SOS, p, \mathit{Succ\_sopen}(\mathit{sid})\rangle \circ \\
&\langle p, SOS, \mathit{Slisten}(\mathit{sid})\rangle \circ \langle SOS, p, \mathit{Succ}\rangle)^+
\end{aligned}$$

for a process p representing the SMTP-Server process and a socket id *sid*. The history set $H_{\text{LOOP}}(p, sid)$ is supposed to cover the parent process of the SMTP-server together with all the children processes that might be initiated. It is defined as the smallest set of histories that satisfies the equation

$$\begin{aligned}H_{\text{LOOP}}(p, sid) = \quad & (\langle p, SOS, Saccept(sid)\rangle)^{+} \cup \\ & (H_{\text{ACC}}(p, sid, sid') \circ H_{\text{FORK\_CALL}}(p) \circ \\ & \quad ((H_{\text{FORK\_ANS\_C}}(p') \circ H_{\text{CHILD}}(p', sid)) \cap \\ & \quad\; (H_{\text{FORK\_ANS\_P}}(p) \circ H_{\text{CLOSE}}(p, sid') \circ H_{\text{LOOP}}(p, sid))))\end{aligned}$$

for some socket id $sid' \neq sid$ and some process $p' \neq p$.
This history set $H_{\text{LOOP}}(p, sid)$ might require some more explanation. First the SMTP-Server issues a socket-accept command. This command might never be answered and thus the SMTP-Server might wait forever (first line in the definition of $H_{\text{LOOP}}(p, sid)$). If, however, there is an answer to the accept-request (another process issued a corresponding connect-request) then the SMTP-Server calls a fork-command, thus producing a child of its own process. Now, both the SMTP-Server and its child run concurrently as indicated by the intersection of the two history sets in the last two lines of the $H_{\text{LOOP}}(p, sid)$ definition.

With this explanation the definition of the history sets $H_{\text{ACC}}(p, sid, sid')$, $H_{\text{FORK}}(p)$, and $H_{\text{CLOSE}}(p, sid')$ should be fairly obvious, namely

$$\begin{aligned}H_{\text{ACC}}(p, sid, sid') &= (\langle p, SOS, Saccept(sid)\rangle\langle SOS, p, Succ\_saccept(sid', rna, rpn)\rangle)^{+} \\ &\quad \text{for some remote network address } rna \text{ and port number } rpn \\ H_{\text{FORK\_CALL}}(p) &= (\langle p, SOS, Afork(1)\rangle)^{+} \\ H_{\text{FORK\_ANS\_P}}(p) &= (\langle SOS, p, Succ\_afork(hdl)\rangle)^{+} \text{ for some handle } hdl \neq none \\ H_{\text{FORK\_ANS\_C}}(p) &= (\langle SOS, p, Create\_Clone(i_C, p, 1)\rangle\langle SOS, p, Succ\_afork(none)\rangle)^{+} \\ H_{\text{CLOSE}}(p, sid) &= (\langle p, SOS, Sclose(sid)\rangle\langle SOS, p, Succ\rangle)^{+}\end{aligned}$$

Note that the first argument of the *Create_Clone* message indicates the (index of the) program that is supposed to run as the child process.
Remains the most complicated case, namely the specification of the child process which is responsible for carrying out the SMTP protocol. As above, we consider only the successful case here.

$$\begin{aligned}H_{\text{CHILD}}(p, sid) = H_{\text{GREETING}}(p, sid) \circ H_{\text{ReadEmails}}(p, sid) \circ \\ H_{\text{QUIT}}(p, sid) \circ H_{\text{CLOSE}}(p, sid)\end{aligned}$$

The history set for $H_{\text{CLOSE(p,sid)}}$ is already defined above. $H_{\text{GREETING}}(p, sid)$ and $H_{\text{QUIT}}(p, sid)$ look as follows:

$$\begin{aligned}H_{\text{GREETING}}(p, sid) &= H_{\text{READY}}(p, sid) \circ H_{\text{ReadLine}}(p, sid, \text{"EHLO "} + ip_r) \circ \\ &\quad H_{\text{GREETS}}(p, sid, ip_r) \quad \text{for some remote ip address } ip_r \\ H_{\text{READY}}(p, sid) &= (\langle p, SOS, Swrite(sid, \text{"220 "} + get\_na(p) + \\ &\qquad \text{" SMT Service Ready"})\rangle\langle SOS, p, Succ\rangle)^{+} \\ H_{\text{GREETS}}(p, sid, ip_r) &= (\langle p, SOS, Swrite(sid, \text{"250 "} + get\_na(p) + \text{" greets "} + ip_r)\rangle \\ &\quad \langle SOS, p, Succ\rangle)^{+} \\ H_{\text{QUIT}}(p, sid) &= H_{\text{ReadLine}}(p, sid, \text{"QUIT"}) \circ \\ &\quad (\langle p, SOS, Swrite(sid, \text{"221 "} + \text{p} + \text{" closing"})\rangle \circ \\ &\quad \langle SOS, p, Succ\rangle \circ \langle p, SOS, Exit\rangle)^{+}\end{aligned}$$

ReadLine consists essentially of successively reading one character after the other. A slight complication arises as it may be possible that the attempt to read a single character may be successful, yet results in an empty string (i. e., we assume the socket-read command to be non-blocking).

$$H_{\text{ReadLine}}(p, sid, string) = \begin{cases} H_{\text{ReadString}}(p, sid, string) & \text{if } \exists s : \; string = s \,\hat{}\, CR \,\hat{}\, LF \\ & \text{and } s \text{ does not contain } CR \,\hat{}\, LF \\ \emptyset & \text{otherwise} \end{cases}$$

i. e., reading a line means to read a string that (uniquely) ends with a carriage return (CR) followed by a line feed (LF).
$H_{\text{ReadString}}$ is defined as the smallest set satisfying the equations

$$\begin{aligned} H_{\text{ReadString}}(p, sid, \text{""}) &= ()^+ \\ H_{\text{ReadString}}(p, sid, c \,\hat{}\, s) &= H_{\text{ReadChar}}(p, sid, c) \circ H_{\text{ReadString}}(p, sid, s) \end{aligned}$$

where

$$\begin{aligned} H_{\text{ReadChar}}(p, sid, c) &= H_{\text{ReadEmpty}}(p, sid) \circ H_{\text{ReadChar1}}(p, sid, c) \\ H_{\text{ReadChar1}}(p, sid, c) &= (\langle p, SOS, Sread(sid, 1)\rangle \langle SOS, p, Succ\_sread(1, \text{"}c\text{"})\rangle)^+ \end{aligned}$$

and $H_{\text{ReadEmpty}}(p, sid) = \mu H.(H = ()^+ \cup H_{\text{ReadEmpty1}}(p, sid) \circ H)$ where

$$H_{\text{ReadEmpty1}}(p, sid) = (\langle p, SOS, Sread(sid, 1)\rangle \langle SOS, p, Succ\_sread(0, \text{""})\rangle)^+$$

Remains to specify the history set $H_{\text{ReadEmails}}$ (which in addition covers writing the email to the Inbox file). $H_{\text{ReadEmails}}$ is the smallest set satisfying the equation $H_{\text{ReadEmails}}(p, sid) = ()^+ \cup \big(H_{\text{ReadEmail}}(p, sid) \circ H_{\text{ReadEmails(p,sid)}}\big)$, i. e.,

$$H_{\text{ReadEmails}}(p, sid) = \mu H. \big(H = ()^+ \cup H_{\text{ReadEmail}}(p, sid) \circ H\big)$$

where $H_{\text{ReadEmail}}(p, sid)$ splits into several parts, namely in reading the sender's address, the recipient's address, the email data and the writing of the email to the file system.

$$\begin{aligned} H_{\text{ReadEmail}}(p, sid) &= H_{\text{ReadS}}(p, sid, s) \circ H_{\text{ReadR}}(p, sid, r) \circ H_{\text{ReadD}}(p, sid, d) \circ \\ &\quad\; H_{\text{WriteEmail}}(p, s \,\hat{}\, r \,\hat{}\, d) \quad \text{for some } s, r, d \\ H_{\text{ReadS}}(p, sid, s) &= H_{\text{ReadLine}}(p, sid, \text{"MAIL FROM: "} + s) \circ \\ &\quad\; (\langle p, SOS, Swrite(sid, \text{"}OK\text{"})\rangle \circ \langle SOS, p, Succ\rangle)^+ \\ H_{\text{ReadR}}(p, sid, r) &= H_{\text{ReadLine}}(p, sid, \text{"RCPT TO: "} + r) \circ \\ &\quad\; (\langle p, SOS, Swrite(sid, \text{"}OK\text{"})\rangle \circ \langle SOS, p, Succ\rangle)^+ \\ H_{\text{ReadD}}(p, sid, d) &= H_{\text{ReadLine}}(p, sid, \text{"DATA:"}) \circ \\ &\quad\; (\langle p, SOS, Swrite(sid, \text{"354 Start mail input;} \\ &\quad\quad \text{end with CRLF . CRLF"})\rangle \circ \langle SOS, p, Succ\rangle)^+ \circ \\ &\quad\; H_{\text{ReadD}'}(p, sid, d) \circ \\ &\quad\; (\langle p, SOS, Swrite(sid, \text{"}OK\text{"})\rangle \circ \langle SOS, p, Succ\rangle)^+ \\ H_{\text{ReadD}'}(p, sid, \text{"."}) &= H_{\text{ReadLine}}(p, sid, \text{"."}) \\ H_{\text{ReadD}'}(p, sid, l \,\hat{}\, d) &= H_{\text{ReadLine}}(p, sid, l) \circ H_{\text{ReadD}'}(p, sid, d) \quad \text{provided } l \neq \text{"."} \end{aligned}$$

The final step is to specify $H_{\text{WriteEmail}}$.

$$\begin{aligned} H_{\text{WriteEmail}}(p, e) = (&\langle p, SOS, Flock(Inbox)\rangle \langle SOS, p, Succ\rangle \circ \\ &\langle p, SOS, Fseek(Inbox, 1, 0)\rangle \langle SOS, p, Succ\_fseek(pos_1)\rangle \circ \\ &\langle p, SOS, Fwrite(Inbox, e)\rangle \langle SOS, p, Succ\_fwrite(pos_2, n)\rangle \circ \\ &\langle p, SOS, Funlock(Inbox)\rangle \langle SOS, p, Succ\rangle)^+ \end{aligned}$$

for some file positions pos_1 and pos_2.

It is certainly out of the scope of this paper to show all the verification details for the whole SMTP-Server. Instead, we emphasise on a small portion of it, namely the readLine procedure as specified above.
In a VSE-like fashion the procedure is listed below:

```
PROCEDURE readLine(sid:length,buffer,res)
int length, ec;
buffer buffer_array;
char c, cprevious;
bool res;
 BEGIN length := 1; res := true; c := null; cprevious := null;
  cl := nil;
  WHILE ((cprevious /= CR OR c /= LF) AND res = true) DO
   length := 1; socket_read(sid:length,buffer,ec);
   if (ec = SUCC) then res := true else res := false fi;
   if (length = 1 and res = t) then
    cprevious := c; c := buffer[0]; cl := write(cl,c) fi
  OD;
 END
```

The readline procedure is supposed to read characters from the given TCP/IP socket until it finds a CR followed by a LF. This behaviour is described by the history set $H_{\text{ReadLine}}(p, sid, cl)$ for a procedure identifier p, socket id sid and a list of characters (string) cl. The segments of the histories that are members of this set are the results of calling the *readLine* procedure from above. Therefore, for the verification of the SMTP server, we need to make sure that this procedure (implementation) meets its intended semantics (the corresponding history sets from above).

According to the technique described above we have to prove the following property:

$$\begin{array}{l}
h_c^0 = h_c \wedge h_{out}^0 = h_{out} \wedge mode = fin \\
\rightarrow \langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle \\
\qquad mode \neq stop \rightarrow \\
\qquad\qquad \exists h : h_{out} = h_{out}^0 \circ h \wedge ((mode = fin \wedge res = t) \leftrightarrow h \in H_{\text{ReadLine}}(p, sid, cl))
\end{array}$$

The proof of this property is split into three main lemmas (and several small lemmas about the data structures used): The first lemma is formulated close to an invariant used to deal with the (single) while loop occurring in the body of *readLine*.

$$\begin{array}{l}
h_{out}^0 = h_{out} \wedge mode = fin \\
\rightarrow \langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle \\
\quad mode \neq stop \\
\qquad \rightarrow \exists h : \ h_{out} = h_{out}^0 = h \wedge \\
\qquad\qquad ((mode = fin \wedge res = t) \\
\qquad\qquad\quad \rightarrow h \in H_{\text{ReadString}}(p, sid, cl) \circ H_{\text{ReadEmpty}}(p, sid)) \wedge \\
\qquad\qquad (h \in H_{\text{ReadString}}(p, sid, cl) \circ H_{\text{ReadEmpty}}(p, sid) \wedge cl \neq \langle\rangle \\
\qquad\qquad\quad \rightarrow (mode = fin \wedge res = t))
\end{array}$$

The following lemma states that we can drop the history sets $H_{\text{ReadEmpty}}$, because $H_{\text{ReadEmpty}}(p, sid) \setminus H_{\text{ReadEmpty1}}(p, sid) = \{[]\}$.

$$\begin{array}{l} h^0_{out} = h_{out} \wedge mode = fin \\ \rightarrow \langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle \\ \qquad mode \neq stop \rightarrow \exists h :\ h_{out} = h^0_{out} \circ h \wedge \\ \qquad\qquad ((mode = fin \wedge res = t) \rightarrow h \notin H_{\text{ReadEmpty1}}(p, sid) \circ H) \end{array}$$

Finally, we need a lemma that deals with the fact that end of lines are marked with $\langle CR, LF\rangle$. Notably, the proof for this lemma does not require any knowledge about the external call simulation *socket_read_sim*. Thus this example shows how a proof can be separated into parts dealing with concurrent communication and those dealing with properties independent of the communication, even if the properties are not separated by the program structure.

$$\begin{array}{l} \langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle \\ \qquad\qquad\qquad mode = fin \wedge res = t \rightarrow \exists cl_0 : cl = cl_0 \circ \langle CR, LF\rangle \end{array}$$

4 Application Level Programs

In this section we describe the construction of $\tilde{\pi}_i$ out of π_i. The C0 program π with external calls is transformed into a program $\tilde{\pi}$ that takes histories as input and produces histories as output but uses only standard function calls. Since histories describe initial segments of nonterminating behaviors the new program is intended always to terminate. We consider its result as an *approximation* of the computation of π following the general replay and extend strategy outline above.

We suggest a uniform transformation of the program into an approximation exhibiting the same behavior as the original program with respect to prefixes of event histories. The transformation preserves the structure of the program. Thus it is possible to use a verification approach that follows the structure of the implementation. Moreover this approach enables us to employ well known verification techniques for sequential programs as described in, for instance, [7] and [8]. The latter system has been used for the verification of SMTP.

4.1 Computing Approximations

In this section the uniform procedure to convert programs π_i into their approximations $\tilde{\pi}_i$ is described. Let the program π_i be given as $\pi_i = (\delta_i | \alpha_i(\bar{x}))$, where $\bar{x}$ are the (program) variables occurring free in α_i and δ_i is the list of procedure declarations used in α_i. The function $approx_{\pi_i}(p_0, h_0)$ will be computed by the program $\tilde{\pi_i}$ given by

```
( $approx\_i(p, h_{in} : h_{out}, mode) \Leftarrow$ **declare** $h_c := h_{in}; \bar{x} := \bar{\sigma}$
    **begin** $mode := fin; h_{out} := []; start\_i(: p, h_c, h_{out}); \widetilde{\alpha_i}(\bar{x});$
        $stop(: h_{out}, h_c, mode)$ **end**,
  $ext\_call(\pi_i), \tilde{\delta_i} \mid approx\_i(p_0, h_0 : h_1, m_0)$ )
```

where the initial values of h_1 and m_0 (used to return the results) are not relevant.

The sequence $ext\_call(\pi_i)$ contains declarations for the procedures that simulate the external calls occurring in π_i together with additional start and stop procedure, *start_i* and *stop*, respectively.

In the computation of the approximation a local variable h_c is used that contains the *currently remaining history* during the execution of $\widetilde{\alpha_i}$. It is set to h_{in} initially. The output history is collected in h_{out} as the computation proceeds while the mode is kept in *mode*.

The construction is guided by the following general idea. An initial segment of the computation of α_i executed by p is replayed using (consuming) h and extending h_{out}.

External calls $c(\bar{\tau} : \bar{z}, res)$ are replaced (or simulated) by procedures with declarations $c\_sim(p, \bar{x} : \bar{y}, res, h_c, h_{out}, mode) \Leftarrow body_c$. The simulating procedures analyze and shorten (consume) the current history h and extend the current output h_{out}. The first argument indicates the process that is executing α_i. Let $\bar{v}$ and $\bar{w}$ be the values of $\bar{\tau}$ and $\bar{z}$, respectively.

If in h_c there is no event generated by p, then the computation (of $\tilde{\alpha_i}$) stops with $h_{out} \circ h \circ [ev_c(p, \bar{v} : \bar{w})]$ as final output, where $ev_c(p, \bar{v} : \bar{w})$ is the event generated by this call of c.

If in h_c there is a further event generated by p, then it has to be $ev_c(p, \bar{v} : \bar{w})$. Otherwise the computation stops signalling a failure. In that case the particular h_c is not realized by π_i (and $\tilde{\pi_i}$) which might happen due to over specification.

For $h_c = h_0 \circ h'_1$, where in h_0 there is no event generated by p and $fst(h'_1) = ev_c(p, \bar{v} : \bar{w})$, h'_1 is scanned for a matching answer event. If there is no such answer, then the computation stops with $h_{out} \circ h$ as the final output history and *mode* being set to *stop*.

In all these cases the procedure simulating the external call leaves the result parameters untouched since they are not needed anymore.

In the following paragraph we make use of the predicate $Match\_ev(e_1, e_2)$ which checks whether the event e_2 represents a matching answer for the event e_1. $\langle p, SOS, Sread(sid, 1)\rangle$ and $\langle SOS, p, Succ\_sread(1, "c")\rangle$ represents an example for a pair of matching messages. These two messages represent the call of a socket read on the socket identified by *sid* and the corresponding answer message containing the read string c (see also chapter 3).

For $h'_1 = h_1 \circ h_2$, where $rst(h_1)$ contains no answer matching $fst(h'_1) = fst(h_1) = ev_c(p, \bar{v} : \bar{w})$ and $fst(h_2) = e$ such that $Match\_ev(ev_c(p, \bar{v} : \bar{w}), e)$ the procedure returns values for the result parameters according to the message contained in e and the computation of $\tilde{\alpha_i}$ continues with $rst(h_2)$ as the new remaining history and $h_{out} \circ h_0 \circ h_1 \circ [fst(h_2)]$ as the new current output.

The above mentioned analysis of the current history h with respect to an external call $c(\bar{\tau} : \bar{z}, res)$ of p is given by $parse_c(p, h, \bar{v}, \bar{w}) \in His \times His \times His$, where again $\bar{v}$ and $\bar{w}$ are the values of $\bar{\tau}$ and $\bar{z}$, respectively.

$$\begin{aligned}
&parse_c(p, h, \bar{v}, \bar{w}) = (h_0, h_1, h_2) \leftrightarrow (h = h_0 \circ h_1 \circ h_2 \land \\
&\quad ev_c(p, \bar{v} : \bar{w}) \notin h_0 \land \\
&\quad (h_1 \neq [] \rightarrow (fst(h_1) = ev_c(p, \bar{v} : \bar{w}) \land \\
&\qquad\qquad\qquad \forall e \in rst(h_1). \neg Match\_ev(ev_c(p, \bar{v} : \bar{w}), e))) \land \\
&\quad (h_2 \neq [] \rightarrow Match\_ev(ev_c(p, \bar{v} : \bar{w}), fst(h_2))))
\end{aligned}$$

The body of the procedure is given below.

```
$body_c$ :≡ **declare** $h_0 := parse_c(p, h_c, \bar{x}, \bar{y})._0$;
                 $h_1 := parse_c(p, h_c, \bar{x}, \bar{y})._1$;
                 $h_2 := parse_c(p, h_c, \bar{x}, \bar{y})._2$
          **begin**
          **if** $mode \neq fin$ **then skip else**
           **if** $\exists e \in h_0.Gen(p, e)$ **then** $mode := fail$ **else**
            **if** $h_2 = []$ **then** $mode := stop$;
             **if** $h_1 = []$ **then**
               $h_{out} := h_{out} \circ h \circ [ev_c(p, \bar{x}, \bar{y})]$;
               $h_c := []$ **fi**
            **else**
                 $h_{out} := h_{out} \circ h_0 \circ h_1 \circ [fst(h_2)]$;
                 $h_c := rst(h_2); y_0 := ret\_val_c^1(fst(h_2))$
                 ...
                 $y_{n-1} := ret\_val_c^{n-1}(fst(h_2))$
                 $res := ret\_res_c(fst(h_2))$
             **fi   fi   fi**
```

where $Gen(p, e)$ is true if the event e is generated by the process represented by p. The function $ret\_val_c^i(e)$ extracts the result parameters from the event e and $ret\_res_c(e)$ returns the result value of the corresponding external call c.

Before the execution of $\tilde{\alpha_i}$ is started the begin of an active thread of p has to be determined by the start procedure, or if p has been started by a fork call, the start of the ancestor's thread who was running the very beginning of the program has to be found. If there is no p-thread executing π_i (or no suitable ancestor), then the given input history is returned as output and *mode* is set to *term*.

The procedure that simulates the start of a process π_i is given by the declaration $start\_i(:\, p, h_c, h\_out, mode) \Leftarrow body_{start\_i}$. It *parses* the given history h according to the definition of *Proc*.

$$\begin{aligned}&parse_{start\_i}(p, h_c) = (h_0, h_1) \leftrightarrow h = h_0 \circ h_1 \wedge \\ &\quad (h_1 \neq [] \rightarrow (Create(i, p, fst(h_1)) \wedge \\ &\qquad\qquad \forall e \in rst(h_1).\neg Term(i, p, e)))\end{aligned}$$

The procedure body then is given below.

```
$body_{start\_i} :\equiv$ declare $ah := anc(i, p, h); h_0 := []; h_1 := [];$
               begin
               if $ah = []$ then $mode := term; h_{out} := h_c;$ else
                   $h_1 := fst(ah); h_0 := \Delta(h_c, h_1);$
                   $h_c := rst(h_1); h_{out} := h_0 \circ [fst(h_1)];$
                   $p := get\_rec(fst(h_1))$
               fi
```

Finally we need a stop procedure $stop(:\, h_{out}, h, mode) \Leftarrow body_{stop}$ that finalizes the simulation. It restores the original history by appending the remaining h to h_{out}. Note that in those cases where a new (final) event was generated h_c will be $[]$. If we have reached the end of $\tilde{\alpha_i}$, indicated by $mode = fin$, we check whether according to the remaining history something needs to be done a signal the result by setting *mode* to fin or *term*, respectively. This information is needed for decomposing verification problems. The body of the stop procedure is then given as

```
$body_{stop} :\equiv h_{out} := h_{out} \circ h;$
              if $mode = fin \wedge \forall e \in h.\neg Gen(p, e)$ then $mode := term$ fi
```

Whenever *mode* is changed (to $m \in \{stop, fail\}$) by a procedure simulating an external call the rest of $\tilde{\alpha_i}$ has to be skipped. This is achieved by adding a kind of guards to while loops and (possibly) recursive procedures. In addition h, h_{out}, and *mode* have to be passed as arguments to the procedures declared in δ_i.

For declarations we have

$$\begin{aligned}
\emptyset &\mapsto_\sim \emptyset \\
q(\bar{x} : \bar{y}) \Leftarrow \beta \ , \ \delta &\mapsto_\sim \tilde{q}(\bar{x} : \bar{y}, h_c, h_{out}, mode) \Leftarrow \\
&\qquad \textbf{if } mode \neq fin \textbf{ then skip else } \widetilde{\beta} \textbf{ fi} \ , \ \widetilde{\delta}
\end{aligned}$$

Commands are modified as follows.

$$\begin{aligned}
\textbf{skip} &\mapsto_\sim \textbf{skip} \\
x := \tau &\mapsto_\sim x := \tau \\
\alpha_0; \alpha_1 &\mapsto_\sim \widetilde{\alpha_0}; \widetilde{\alpha_1} \\
\textbf{if } \epsilon \textbf{ then } \alpha_0 \textbf{ else } \alpha_1 \textbf{ fi} &\mapsto_\sim \textbf{if } \epsilon \textbf{ then } \widetilde{\alpha_0} \textbf{ else } \widetilde{\alpha_1} \textbf{ fi} \\
\textbf{while } \epsilon \textbf{ do } \alpha \textbf{ od} &\mapsto_\sim \textbf{while } \epsilon \wedge mode \neq fin \textbf{ do } \widetilde{\alpha} \textbf{ od} \\
q(\bar{\tau} : \bar{z}) &\mapsto_\sim \tilde{q}(\bar{\tau} : \bar{z}, h_c, h_{out}, mode) \\
c(\bar{\tau} : \bar{z}, res) &\mapsto_\sim c\_sim(p, \bar{\tau} : \bar{z}, res, h_c, h_{out}, mode)
\end{aligned}$$

5 Conclusion and Related Work

Our work was motivated by the problem of verifying application level programs with certain communication primitives given a complex formal model for distributed instances of an operating system that are connected by a network and include machines for the interpretation of C0 programs. Instead of working directly on the model we have introduced sets of (finite) sequences H of communication events to specify open distributed systems. This is a particular kind of stream specification as discussed in [9]. However, we restrict ourselves to prefix closed sets expressing safety properties.

Apart from abstracting from the local state spaces these histories were used for a reduction to local verification problems (compositionality). In case of application level programs this reduction is provided by a uniform transformation $\pi \mapsto \tilde{\pi}$. Once and for all we had to establish a relation to the original model by a simulation theorem. This proof is based on the Verisoft C0 interpreter and is the only semantic consideration necessary in our approach. As opposed to the Hoare-style proof system presented in [10] we do not need a new semantic interpretation for $\tilde{\pi}$.

An earlier attempt to map the Verisoft model to the temporal framework implemented in VSE failed. Temporal verification techniques, like those mentioned in [11], turned out not to be appropriate for large programs (more than 7.500 lines) and complex internal data structures. The results of the verification of $\tilde{\pi}$ can be viewed as properties of a (total!) function $approx_\pi : Hist \rightarrow Hist$ that (possibly) extends a given history h by a further event (step). Turning this function into an action of TLA, [12], (manipulating variables for histories) allows for a temporal treatment of liveness and reactivity. The underlying safety assertion, $\Box h \in H$ has already been established *outside* temporal logic.

Despite many technical differences the basic idea for the reduction to $\tilde{\pi}$ is similar to the use of (prefix closed) time diagrams in [10]. In particular this holds for the distinction between events caused by $\tilde{\pi}$ and those caused by the environment. However, neither do we need a special semantics for the transformed program $\tilde{\pi}$ nor an explicit composition theorem for the concurrent execution of programs. Composition as well as the inference of additional properties is done entirely at the level of history specifications H. For the latter we might use functions that extract (as a first-order data structure) for example "the last e-mail that was sent" from a given history h.

References

1. The Verisoft Consortium: The verisoft project http://www.verisoft.de.
2. Cheikhrouhou, L., Rock, G., Stephan, W., Schwan, M., Lassmann, G.: Verifying a chip-card-based biometric identification protocol in vse. In: The 25th International Conference on Computer Safety, Security and Reliability (SAFECOMP 2006). (2006)
3. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security **6** (1998) 85–128
4. Mantel, H.: Information flow control and applications — bridging a gap. Lecture Notes in Computer Science **2021** (2001)
5. de Roever, W.P.: Concurrency Verification – Introduction to Compositional and Noncompositional Methods. Cambridge University Press (2001)
6. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In Hurd, J., Melham, T.F., eds.: Proceedings of the TPHOLs 05, Springer (2005) 1–16
7. Schirmer, N.: A verification environment for sequential imperative programs in isabelle/hol. In Baader, F., Voronkov, A., eds.: Proceedings of the LPAR 04, Springer (2005) 398–414
8. Hutter, D., Langenstein, B., Sengler, C., Siekmann, J.H., Stephan, W., Wolpers, A.: Deduction in the Verification Support Environment (VSE). In: Proceedings FME96. Volume 1051., Springer (1996)
9. Broy, M., Stolen, K.: Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces and Refinement. Springer (2001)
10. de Boer, F.S., Hannemann, U., de Roever, W.P.: Hoare-style compositional proof systems for reactive shared variable concurency. In: Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science, London, UK, Springer-Verlag (1997) 267–283
11. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer (1991)
12. Lamport, L.: Specifying Systems – The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2003)