Welcome to SESF

Correctness branch

A. Udaya Shankar

May 21, 2021

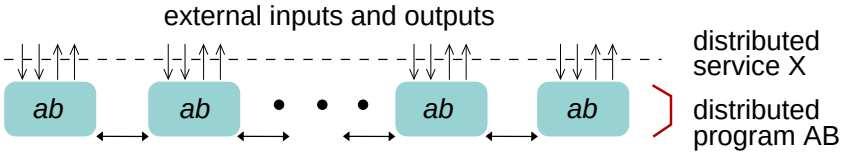
Sesf is a framework to write provably-correct **multi-threaded distributed programs**, test them in distributed execution, and prove their correctness. At the core of sesf is a convenient way to specify the intended behaviour of a multi-threaded distributed program. The resulting specification, referred to as a "service", yields a program for testing implementations of the service as well as a program for testing users of the service. In a distributed system, any distributed subsystem can be replaced by another distributed subsystem satisfying the same service without disturbing correctness.

Sesf can be applied in any programming language. Here, we focus on proofs of correctness, using assertional reasoning on programs in Python-like pseudocode. There is a parallel testing branch that focuses on testing of programs in Python.

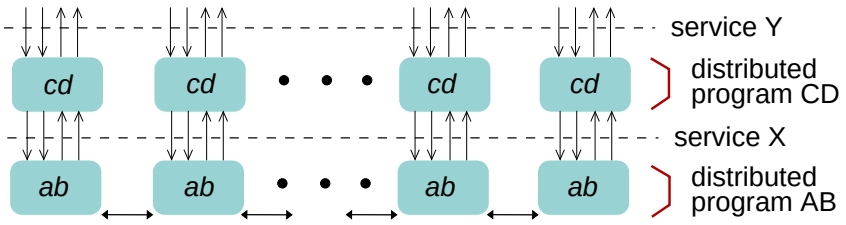
Multi-threaded distributed programs

An executing multi-threaded distributed program consists of one or more processes, each with one or more threads. Threads within a process typically interact via shared memory. Threads in different processes interact via message passing. Each process executes a program, and these programs together constitute the **distributed program**.

The illustration below shows a distributed program AB consisting of multiple processes, each executing a program *ab*. In addition to the *internal* interactions between its *ab* processes, the distributed program has *external inputs and outputs* via which it provides a distributed service X. For example, *ab* could be a TCP program and X could be stream internet sockets.



The illustration below shows a distributed program CD, consisting of processes each executing a program *cd*, that makes use of the service X provided by the distributed program AB, and in turn provides a distributed service Y (eg, file transfer).



Services

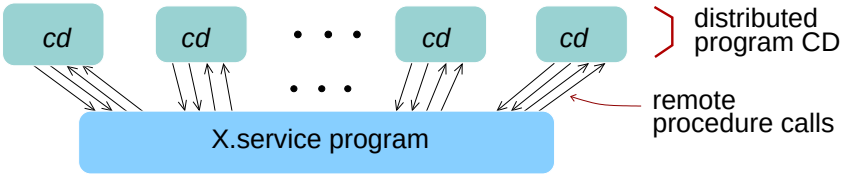
A key step in writing a correct program is to come up with a precise description of what the program is supposed to do, that is, its intended service. Fundamentally, this amounts to a precise description of the **acceptable sequences of the external inputs and outputs** of the program. The program is **correct** if every one of its possible input-output sequences is acceptable to the service: that is, at any point in the program's execution, it can take in any acceptable input and only generate an acceptable output. (This is made precise later.)

Defining the service for a multi-threaded program is more complicated than for a single-threaded program. Because of concurrent threads in the program, a variety of outputs can happen at any point. Similarly, because of concurrent threads in the environment, a variety of inputs can happen at any point. The service must allow for these possibilites when defining the set of acceptable input-output sequences.

We have additional requirements of a service. It should be easily understandable to humans, in particular, much more so than a typical implementation. It should enable testing and verification of programs that implement the service and programs that use the service.

In Sesf, a service is defined by a special kind of program, referred to as a **service program**. There are actually two versions of the service program: an "abstract" version, where the set of possible outputs are defined by predicates; and a "concrete" version, where these predicates are replaced by code that randomly selects a possible output. Only the abstract version is used in this correctness branch. The concrete version is for testing.

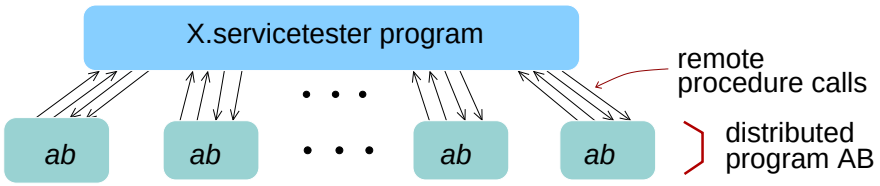
Correctly using a service: Given an application program that makes use of a service, the composite program of the application program and the abstract service program can be analyzed to *prove* that the application correctly uses the service. For the above example of application program CD using service X, this composite program is illustrated below.



Because of its special structure, a service program can be "inverted" to yield a so-called **servicetester program**, which can send arbitrary acceptable inputs to an implementation and check whether the implementation's outputs are acceptable. The servicetester program also comes in abstract and concrete versions, and only the abstract version is relevant for correctness.

Correctly implementing a service: Given an implementation program for a service, the compos-

ite program of the abstract servicetester program and the implementation program can be analyzed to *prove* the correctness of the implementation. For the above example of implementation program AB for service X, this composite program is illustrated below.



Compositionality is the payoff for correctly using services and correctly implementing services. If a distributed program correctly uses a service X to achieve a correctness property, which could include implementing another service Y, then replacing the service X by any correct implementation of X preserves the correctness property. In our example above, program CD will implement service Y correctly if service X is replaced by any program that implements X correctly.

Establishing correctness

A typical multi-threaded program, whether in one process or multiple processes, has an unbounded number of possible executions. Testing can examine only a small subset of these executions. To claim correctness requires program analysis.

There are various analysis methods. We use **assertional reasoning**. Here, one invents a sequence of assertions, ultimately ending with assertions that imply the intended service. Each assertion is shown to hold for the program given previous assertions, via operational arguments or proof-rule applications. The latter can be mechanically checked (by theorem provers or, in the case of finite-state programs, by model checkers).

An **assertion** is a boolean condition evaluated on an execution of the program. It can relate variables within a process and also across processes, eg, an account balance displayed in a web client tracks the account balance stored in the server's database. An assertion holds for the program if it holds for *every* possible execution. (There are different kinds of assertions: we mostly use "invariant" assertions and "leads-to" assertions.)

Assertions are useful even if they are not formally proved. They are an unambiguous and convenient way of stating properties of the program. Furthermore, assertions, even global assertions relating variables across different processes, can be checked in the servicetester during testing.

Background

The theory and proofs here are extracted, with some changes in treatment and terminology, from the text Distributed Programming: Theory and Practice.

Sesf stands for "Services and Systems Framework". The term **service** has been introduced above. We use the term **system** to refer to an executing program, a part of an executing program, or a (perhaps dynamic) collection of executing programs. So unlike a process in operating systems terminology, a system may span a part of one address space or multiple address spaces.

What's next

The rest of the correctness branch is made up of the following documents. The first two give background. The remainder are applications of sesf. Each application gives a service program defining a service, the corresponding servicetester program, and one or more implementation programs and proofs of correctness.

- Sesf basics (`sesf_basics.pdf`) (background material)
- Assertional reasoning (`a.pdf`) (background material)
- Read-write lock (`rwlock/rwlock.pdf`)
-