FortranTestGenerator: Automatic and Flexible Unit Test Generation for Legacy HPC Code

Christian Hovy, *Universität Hamburg*, Email: christian.hovy@uni-hamburg.de
Julian Kunkel, *Deutsches Klimarechenzentrum GmbH*, Email: kunkel@dkrz.de

Abstract

Unit testing is an established practice in professional software development. However, in high-performance computing (HPC) with its scientific applications, it is not widely applied. Besides general problems regarding testing of scientific software, for many HPC applications the effort of creating small test cases with a consistent set of test data is high.

We have created a tool called **FortranTestGenerator** to reduce the effort of creating unit tests for subroutines of an existing Fortran application. It is based on **Capture & Replay (C&R)**, that is, it extracts data while running the original application and uses the extracted data as test input data. The tool automatically generates code for capturing the input data and a basic test driver which can be extended by the developer to a meaningful unit test. A static source code analysis is conducted, to reduce the number of captured variables. Code is generated based on flexibly customizable templates. Thus, both the capturing process and the unit tests can easily be integrated into an existing software ecosystem.

Unit Tests in HPC

There are several definitions of what a unit test is and what is not Some of are very strict regarding the duration of execution or the degree of isolation. Our pragmatic and HPC compatible approach to define unit tests is as follows:

1. The *code under test (CUT)* is a **defined subset** of the application code, usually a certain module or routine. It is tested **isolated** from the parts of the application that call the CUT.
2. Testing does not focus on scientific validation but on **finding defects** in the implementation and/or preventing code regression.
3. Execution is **significantly faster** than running the whole application.
4. It is **automated**, that is, the check whether or not a test is passed is done by the test program itself and not by a person.

Obstacles for Unit Tests

Besides general problems regarding testing of scientific software such as the lack of test oracles, we found several other obstacles for introducing unit tests in HPC software development:

Effort

- Tests need test data as input.
- HPC applications handle large data sets.
- Manual set-up of test data can be exhausting.
- Even worse when the data layout is hardly reproducible (e.g. for unstructured grids or when using cache blocking).

Parallelism

- HPC applications usually run in parallel on multiple nodes and CPUs.
- Communication and synchronization introduce additional classes of errors such as deadlocks and race-conditions.
- When testing a single subroutine, we have to reproduce its MPI communication correctly.

Legacy Code

- HPC applications can be 100k to million lines of code and can have a long history.
- The older the code base, the harder it becomes to introduce significant changes.
- When changes are made to a piece of software, it needs to be tested that these changes don't have unintended side effects.
- Often software needs to be changed again to make it testable.

Tool Support

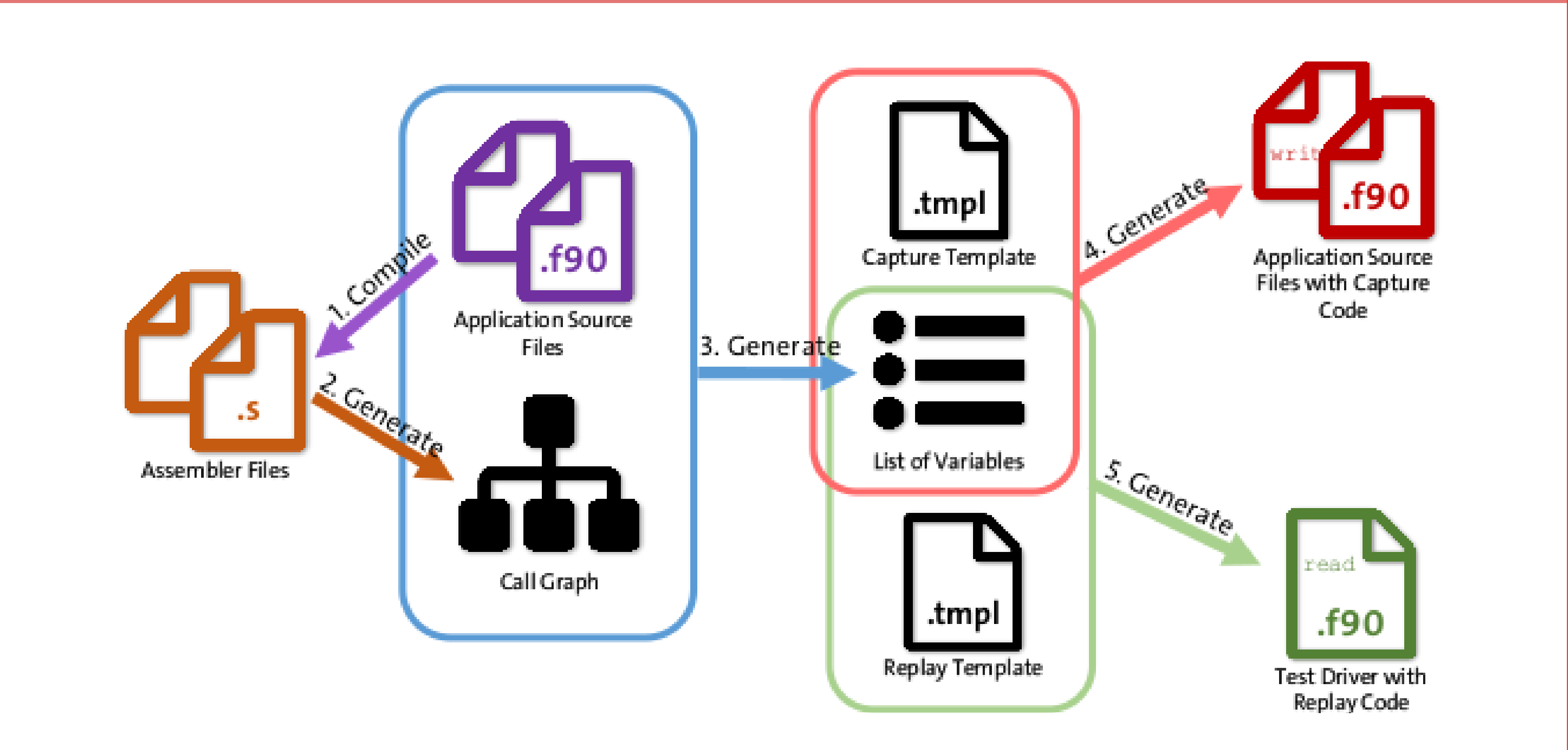
- Fortran is widely used in HPC
- Tool support cannot keep up with languages like Java or C#.
- Since the main goal for HPC is to solve grand challenges computationally, tools that optimize performance are usually of more interest than for testing.
- In recent years, a handful of unit testing frameworks for Fortran have emerged (e.g., pFunit[a]).

[a]http://sourceforge.net/projects/pfunit

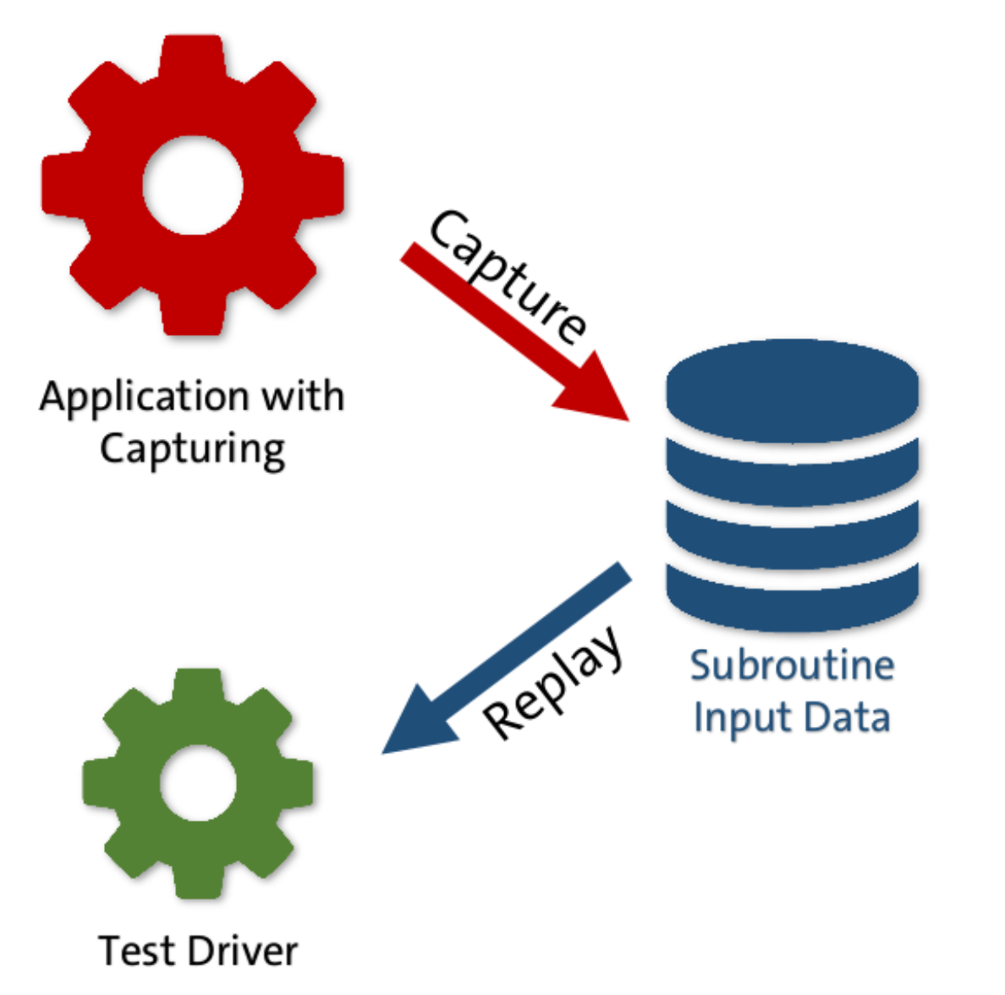
The Tool

FortranTestGenerator (FTG) is a tool for automatically generating unit tests for subroutines of existing Fortran applications.
The main effort for creating unit tests for HPC applications is the set-up of consistent input data. When working with legacy code, we can make use of the existing infrastructure and extract test data from the running application. FTG generates code for serializing and storing a subroutines input data and inserts this code temporarily into the subroutine *(capture code)*.
In addition, FTG generates a basic test driver which loads this data and runs the subroutine *(replay code)*. Meaningful checks and test data modification needs to be added by the developer.

General function of FortranTestGenerator

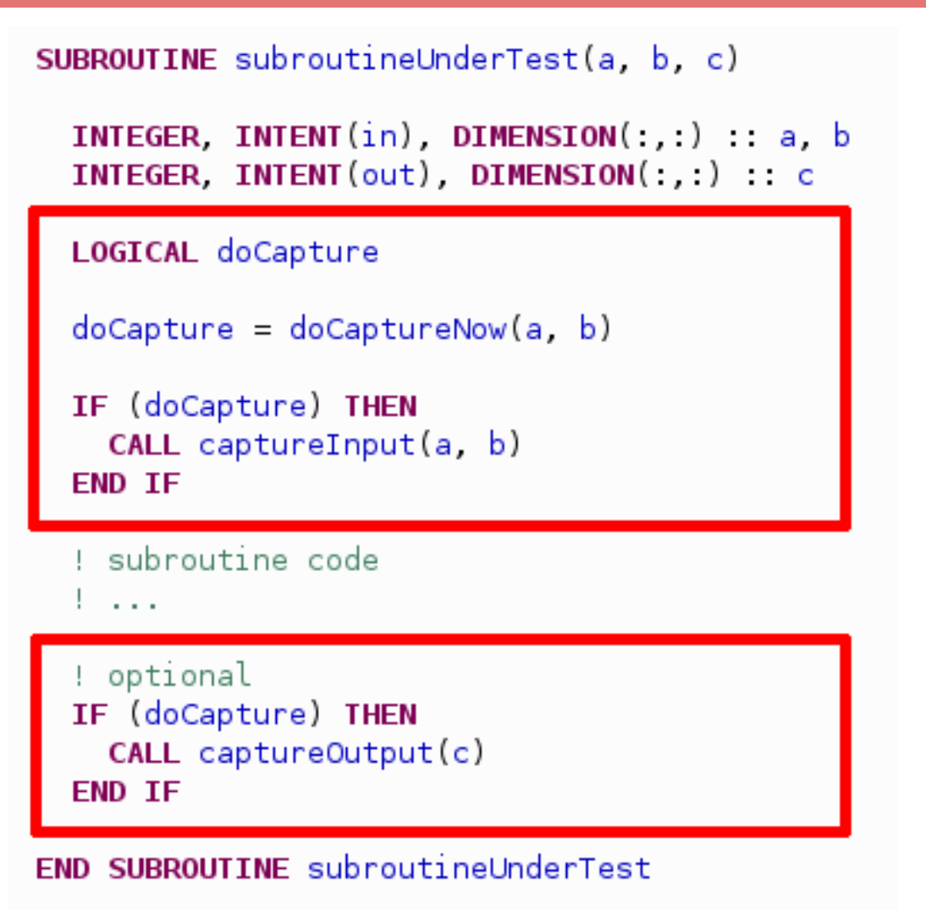


Source Code Analysis and Generation In a first step, the user has to provide assembler files (Step 1) from which FTG builds up a call graph starting from the *subroutine under test (SUT)* (Step 2). The code for capturing all the needed variables is automatically generated using static code analysis (Step 3) and inserted by FTG into the original application code (Step 4). The code analysis is conducted to find out which variables are actually needed by the subroutine. These are the input variables but also global or module local variables are considered. Finally, a basic test driver for loading the input data and running the subroutine in isolation is generated (Step 5). All the generated code *can* be customized by templates.



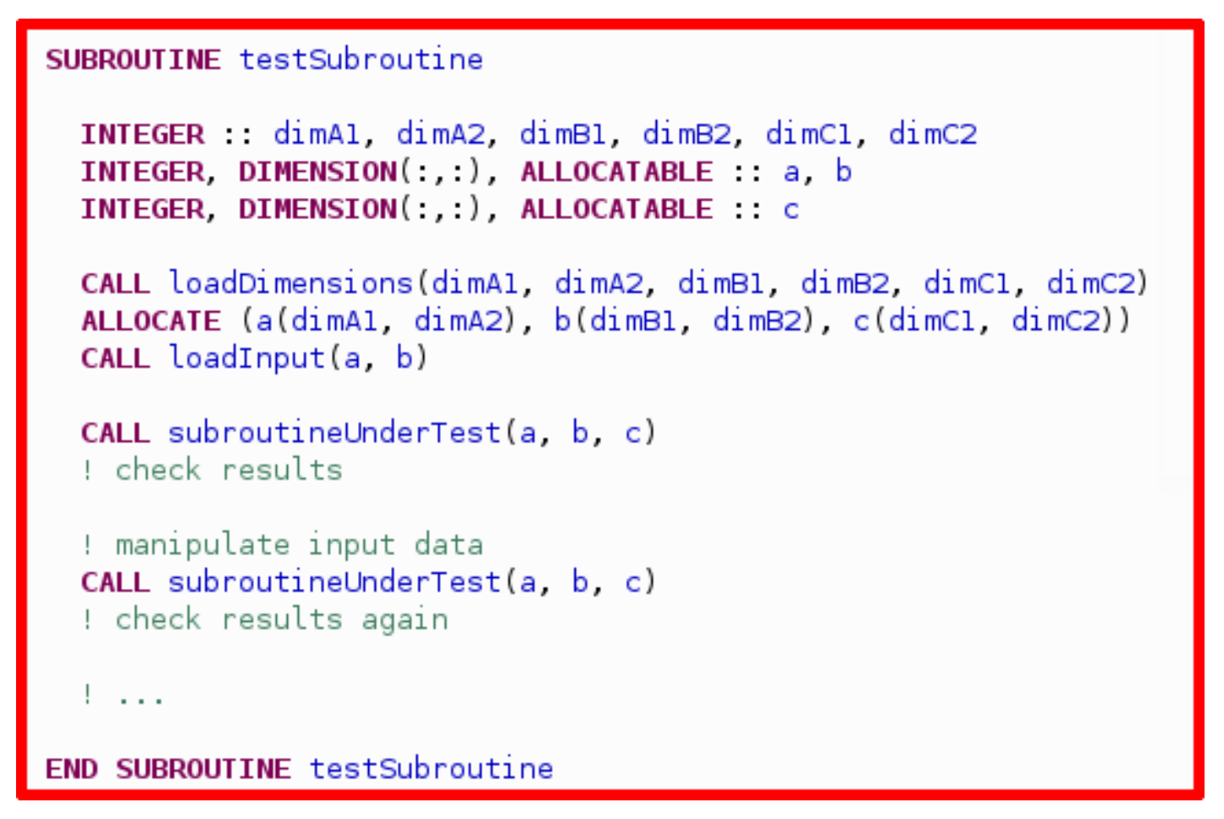
Runtime Now, the user can compile and run the original application, now containing the capture code, to extract a consistent set of input data for the SUT. The default time of capturing (e.g. the 1st or any nth execution of the SUT) is defined by the template, but can easily be changed in the generated code. Afterwards the test driver can executed with the captured data. When the first replay was successful, the user can go on and extend the test driver with meaningful checks or add additional tests by calling the SUT again with altered data.

Example



```
SUBROUTINE subroutineUnderTest(a, b, c)

  INTEGER, INTENT(in), DIMENSION(:,:) :: a, b
  INTEGER, INTENT(out), DIMENSION(:,:) :: c

  LOGICAL doCapture

  doCapture = doCaptureNow(a, b)

  IF (doCapture) THEN
    CALL captureInput(a, b)
  END IF

  ! subroutine code
  ! ...

  ! optional
  IF (doCapture) THEN
    CALL captureOutput(c)
  END IF

END SUBROUTINE subroutineUnderTest
```

Capture The code generated by FTG (red boxes) inserted into the existing subroutine.



```
SUBROUTINE testSubroutine

  INTEGER :: dimA1, dimA2, dimB1, dimB2, dimC1, dimC2
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: a, b
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: c

  CALL loadDimensions(dimA1, dimA2, dimB1, dimB2, dimC1, dimC2)
  ALLOCATE (a(dimA1, dimA2), b(dimB1, dimB2), c(dimC1, dimC2))
  CALL loadInput(a, b)

  CALL subroutineUnderTest(a, b, c)
  ! check results

  ! manipulate input data
  CALL subroutineUnderTest(a, b, c)
  ! check results again

  ! ...

END SUBROUTINE testSubroutine
```

Replay Test driver generated by FTG

Source Code Analysis

FortranTestGenerator conducts a static source code analysis to find out which variables are needed to replay a given subroutine:

Variables passed as arguments FTG distinguishes between arguments of *intrinsic* (primitive) and *derived* (user-defined) types. While arguments of intrinsic types are assumed to be needed by the subroutine anyway, the usage of `TYPE` arguments is analyzed to find out which *components* (members) of a structure are actually accessed by the subroutine or within one of its called subroutines or functions.

Module variables Besides the analysis of accessed `TYPE` components, FTG is able to find the *module variables* which are accessed by the subroutine or by called code. Those module variables can either belong to the same module of the current subroutine/function or be imported by *USE* statements.

Common blocks are currently not supported by FTG since they are not considered as good practice any more and, thus, rarely used in modern Fortran. But since many HPC applications contain very old legacy code, it is planned to support them in the future.

Code Templates

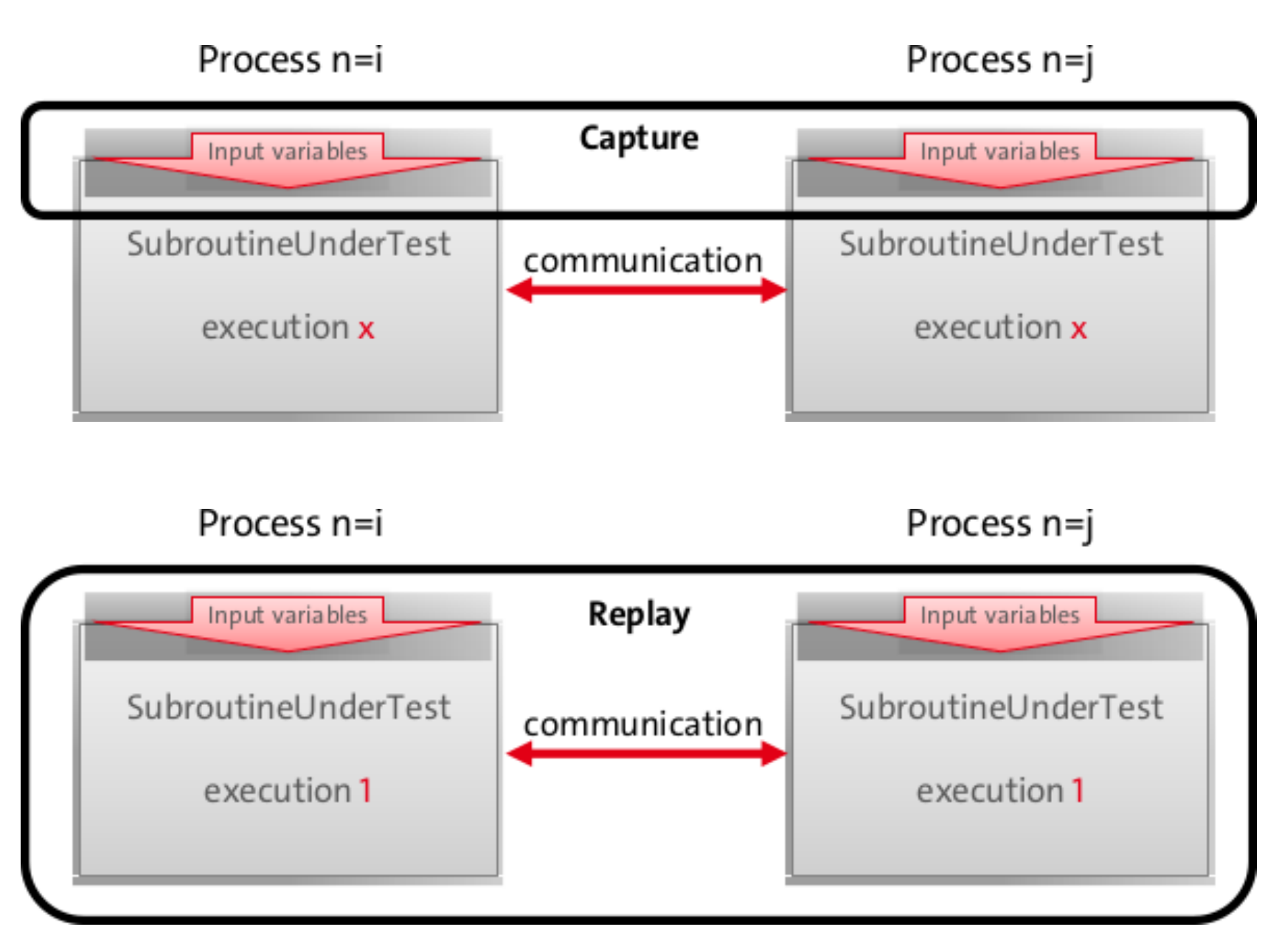
All of the code generated by FTG is based on customizable templates. So, one can for example...

- replace the serialization framework
- include and initialize additional libraries
- make use of a unit testing framework
- add some standardized checks
- choose between embedding into the main application/a test suite, or generating standalone test programs

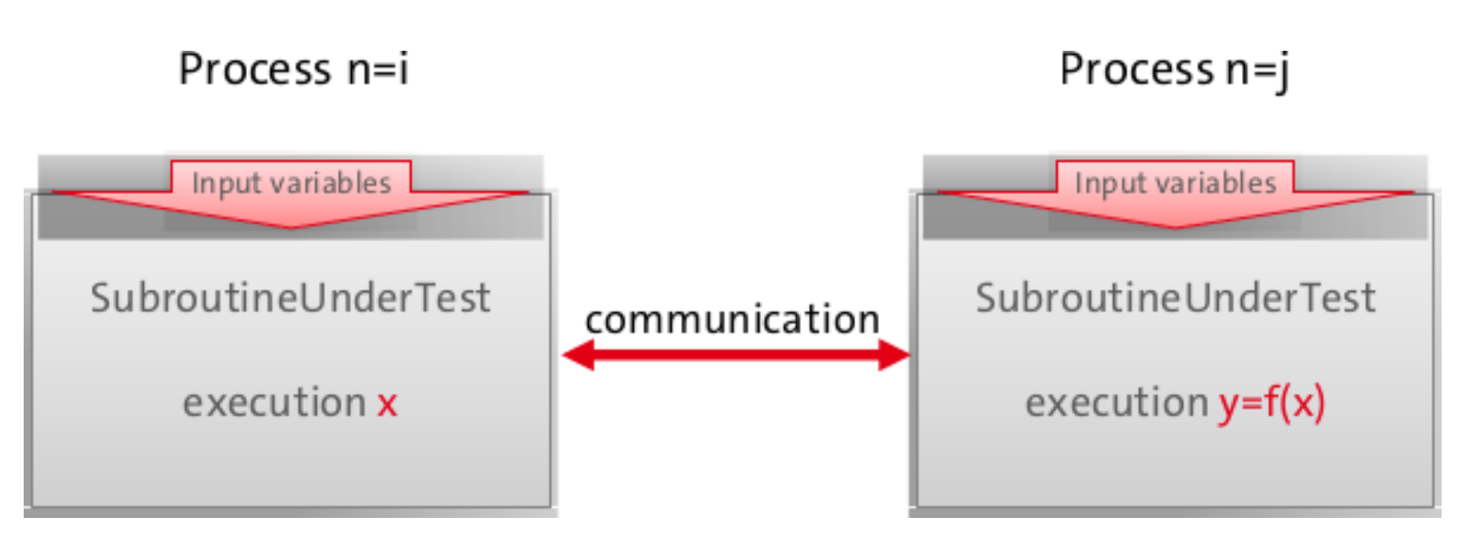
User Interface

- Configuration file
  - Location of source and assembler files
  - Exclude modules, routines or variables
- Code templates (optional, default templates available)
- Commandline interface, for example to generate capture code:
  `$> ./FortranTestGenerator -c <module> <subroutine>`
- Generated Code
  - Modifiable (optional)
  - Compiled and run manually

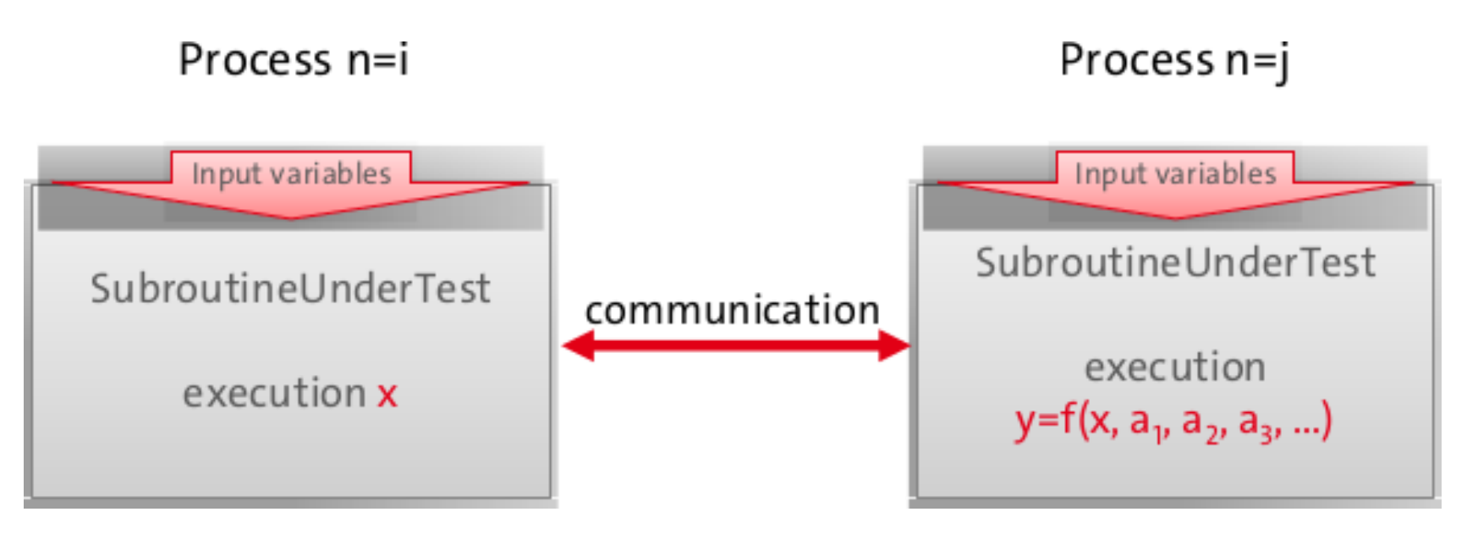
Capture & Replay with MPI

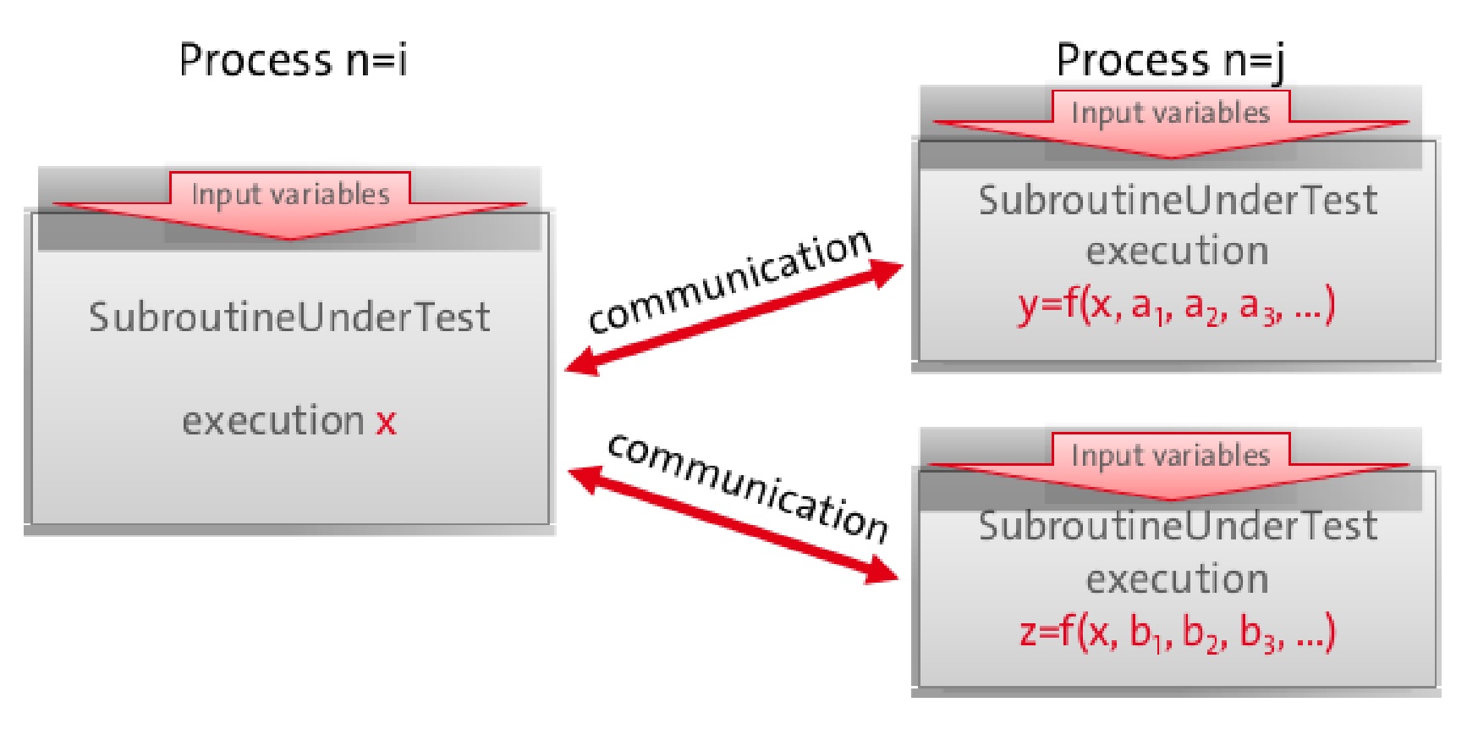


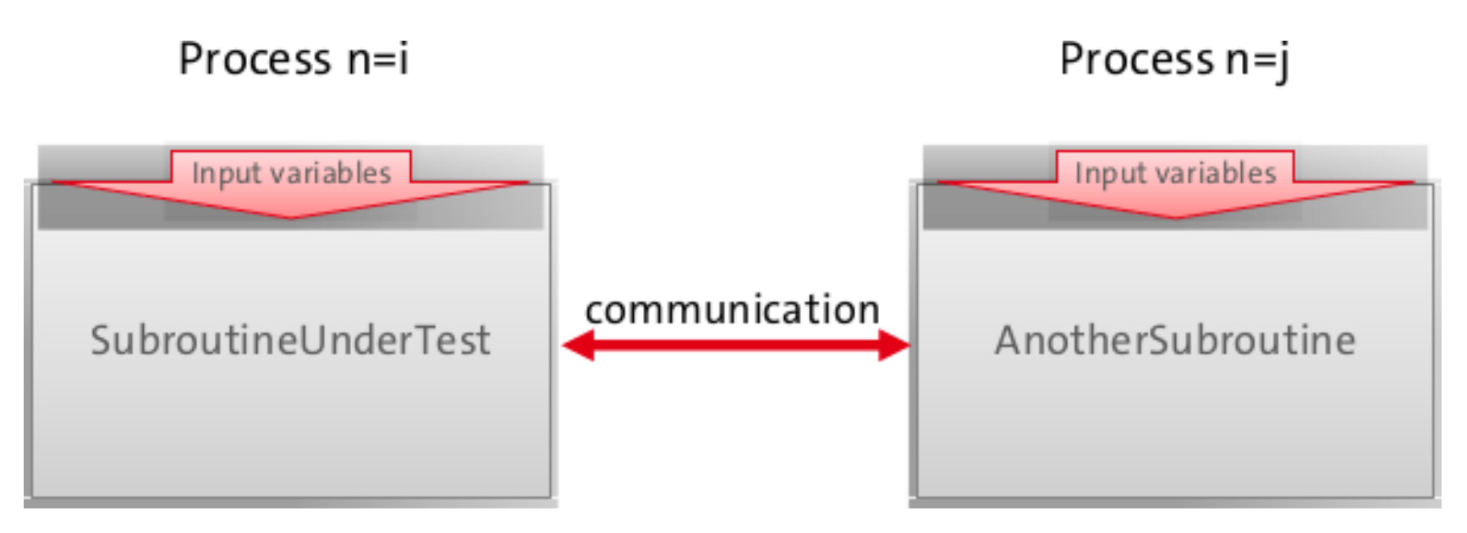
The easy case The subroutine under test (SUT) runs synchronized and execution number x on process i communicates with execution x on process j. So, FTG is able capture and replay execution x on both processes in parallel.



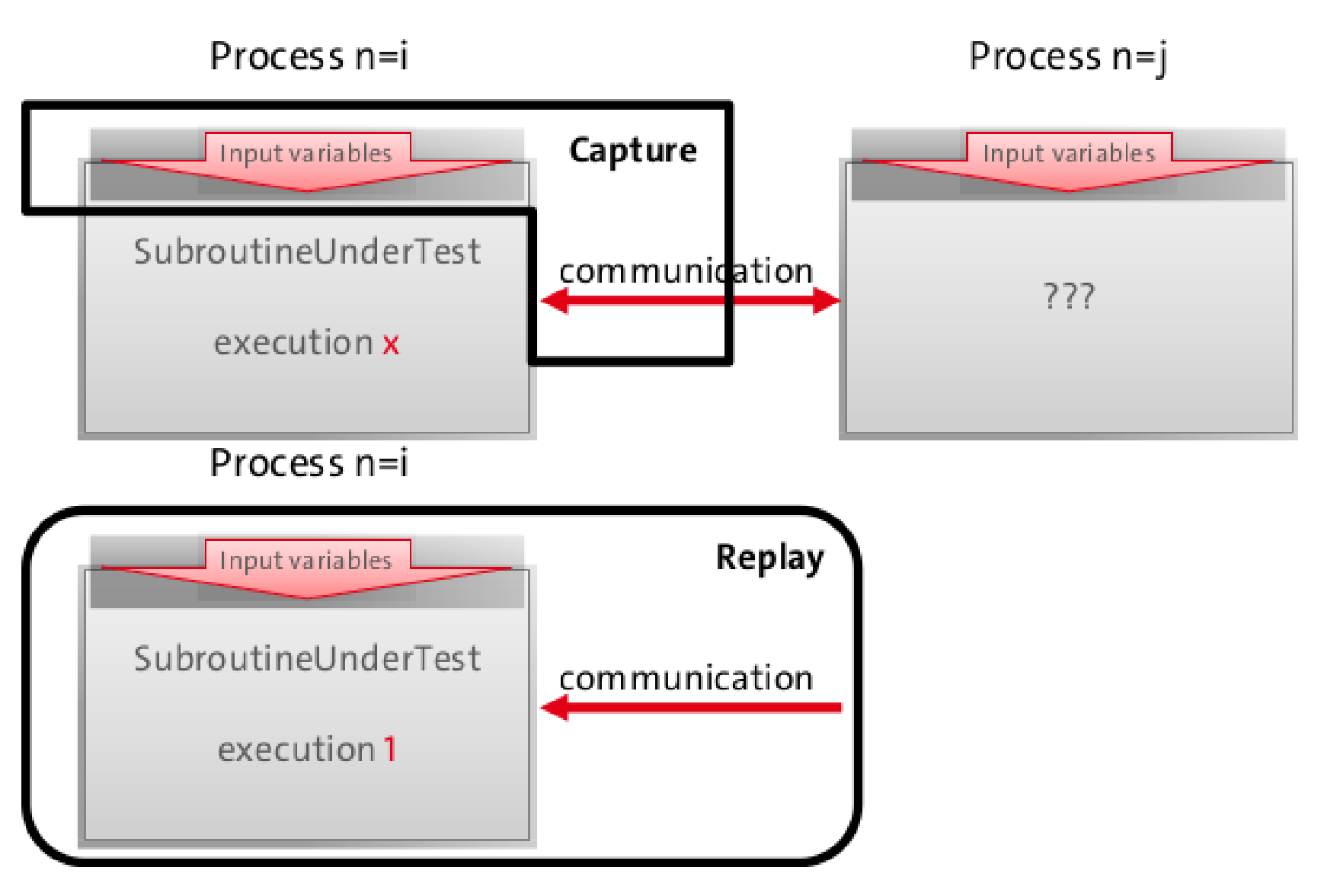
Another feasible case The communication is a bit more complicated, but foreseeable. Here, execution x on process i communicates with execution y on process j, where y is calculable from x.







Unfeasible or difficult cases The communication is unforeseeable, or one execution of the SUT communicates with more than one other execution or even with another subroutine.



The solution Also capture MPI communication (not yet implemented). Currently we are working on an MPI wrapper library to enable capture & replay of individual processes.

Facts

- Written in Python
- Templates based on the Cheetah Template Engine
- Works with Fortran90 or later
- Needs GCC assembler files for analysis
- **http://github.com/fortesg**

References

[1] C. Hovy and J. Kunkel. Towards automatic and flexible unit test generation for legacy hpc code. In *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*, pages 1–8, Nov 2016.