Towards Automatic and Flexible Unit Test Generation for Legacy HPC Code

Christian Hovy
Department of Informatics
Universität Hamburg
Hamburg, Germany
Email: christian.hovy@uni-hamburg.de

Julian Kunkel
Deutsches Klimarechenzentrum GmbH
Hamburg, Germany
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, that significantly reduces 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 an appropriate 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.

Since most HPC applications use message passing for parallel processing, we also present an approach to extend our C&R model to MPI communication. This allows extraction of unit tests from massively parallel applications that can be run with a single process.

I. Introduction

Testing is an essential activity to ensure code quality in software development [1]. Although testing of scientific HPC applications is generally considered to be difficult [2]–[4], you cannot get around it. Since software almost always contains bugs (*defects*), the main purpose of testing is to find as many defects in a piece of software as possible. Usually this is done by checking whether the code is behaving correctly in some defined, exemplary situations. Therefore, the code is executed with a given set of input data and afterwards the behavior is compared with the expected behavior which is given by some kind of *test oracle* [5]. Such a test oracle can be the expertise of the developer or another person, the results of an alternative implementation, a manually solution, or a table in book, or what so ever.

Beyond that, tests can check exceptional situations or corner cases that are likely to end up in a failure. For example, it can be checked what happens if a certain input value exceeds a given threshold, or if some leading sign is inverted. Even if the expected result is unknown, it can be checked that the code does not compute obiously wrong results in such situations. Tests are valuable, even when no test oracle exists; this is important as in scientific HPC applications the outcome is often a-priori unknown [3], [6].

Tests are also a way to learn more about a problem by describing it in a different way, complementary to the code which implements the problem's solution. Furthermore, a test can act as a documentation which shows other developers what a piece of code is intended to do and what are the cases that are already accounted.

A. Levels of Testing

Software Tests can be classified by many dimensions. One classification of tests is the level of abstraction on which a software is tested. The highest levels of abstraction is assessed by *Acceptance Tests* and *System Tests* [1] where the component under test is a complete, integrated system. Those tests are often conducted under real world conditions. They are time-consuming and often done by designated test persons or teams.

The other extreme are *Unit Tests* which only check small pieces of code, for example a single module or a single function. Unit Tests became popular with object-oriented programming where the tested units are classes or methods. Normally, unit tests are written by the developer of the tested component and not by dedicated testers. They are designed to run frequently, automated, and fast.

System tests are an integral part of the *quality assurance (QA)* in a software project. While Unit Tests also have QA benefits, their purpose is more to support developers at their daily work. Unit tests can provide quick, fine-grain feedback on the behavior of the component a developer is just working on. Unit tests are especially useful for daily tasks such as debugging, code refactoring, or optimization to prevent *code regression* (i.e., unintended side effects that break existing functionality).

There are several definitions of what a unit test is and what is not [7]–[9]. Some of them 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

SE-HPCCSE16; Salt Lake City, Utah, USA; November 2016
978-1-5090-5224-0/16/31.00

call the CUT, but don't have to be separated from the modules which are used by the CUT itself.
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. For most HPC applications that means that a test shouldn't contain the main loop, for example, in numerical simulations with time integration, a test should cover only one time step. As a consequence, unit tests are not a means to test the stability of numerical algorithms but they can test the stability of one integration step. Similar techniques may be used to perform tests across time steps, but we won't call it a unit test.
4) Execution 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.

B. Automatic Unit Test Generation for HPC

In HPC context, unit tests can be as useful as in other kinds of software projects. For example when optimizing a piece of code, it is very useful to have a fine-grained regression test which tells the developer whether or not code behavior changes with the modifications. Without automated tests it can be much harder and time-consuming to reproduce situations in an HPC application which normally occur only after hours of simulation time.

In legacy HPC applications such tests are often missing. The usual means to test code in HPC is to run scientific experiments and validate the accuracy of the results [2], [6]. But the time to develop additional unit tests cannot be justified easily by the domain scientists. There are several obstacles which we describe in the following Section II. For making it easier for HPC developers to start writing unit tests for an existing Fortran application despite these problems, we have created a tool called *FortranTestGenerator* which automatically generates a basic test driver for a given subroutine and extracts a set of test data from executing the existing application. The tool is presented in detail in Section III. It is not a framework for fully automated testing such as [10] but automates the setup of consistent test data as starting point for manually written tests. In Section IV we show the problems of this approach regarding MPI communication and how we are going to assess these issues. In Section V, we have a look on related work in this area – followed by an outline of our next steps in Section VI.

The main contribution of this paper is the presentation of an approach to significantly reduce the effort of creating unit test for existing Fortran applications. The approach is based on capturing input variables of a certain subroutine while executed inside the existing application and replaying the subroutine in a generated test driver. Further, we show how this approach can be extended by also capturing and replaying data communicated via MPI.

II. Obstacles for Unit Tests in HPC

Unit testing is an established method in professional software development, especially with object-oriented programming languages. But many HPC developers neither have a formal software engineering education nor experience in industrial software development and they are often hesitant to adopt software engineering methods [2]. While many HPC developers might have heard about unit testing, they may not be aware of how to use such technique in their area.

Testing in general is a difficult field in scientific software development. Literature describes several problems regarding testing of scientific software [11], [12]. Presumably mostly mentioned is the lack of test oracles [5] due to the fact that the correct results of scientific code is often unknown. However, there are several technique for testing software without oracles [6]. And for regression testing there is always a natural oracle – the previous behaviour of the software. In the end, the acceptance of a method is strongly dependent upon the trade-off between cost and value.

A. Effort

Tests need test data as input for the CUT. HPC applications are handling large data sets, which need to be set up for a test case. Doing this manually can be exhausting. Scaling down the domain is an option but often only feasible to some extend as it may conceal the defect to find. This is even worse when the data layout is hardly manually reproducible, for example, when applications are working on *unstructured grids* or using *cache blocking* across multiple variables concurrently.

Some applications also contain *user-defined data types* with long member lists and pointers to such data structures are passed from subroutine to subroutine. For running a test for a single subroutine it is only necessary to initialize those members properly which are actually used by this subroutine. But when writing a test for an existing routine, it can be exhausting to find out manually which members are actually needed.

B. Parallelism

Parallel HPC applications are distributed across large systems and their processes are concurrently executed to contribute to a shared computation. Programming models such as MPI for distributed memory and OpenMP for shared memory need to provide means to exchange data on such systems. This is required for the concurrent execution of code to contribute to the computation goals. Communication and synchronization introduces additional classes of errors such as deadlocks and race-conditions that naturally are difficult or impossible to identify at compile time.

Debugging processes running on thousands of nodes can be overwhelming and led to the development of scalable debuggers such as DDT [13] and programming model specific tools like Marmot for MPI [14]. When considering a unit test, with MPI we have to deal with code that may require input that stems from another process or sends parts of its output. Running such a function with one process is not possible as

the important input is not available, also checking the output of a function is not sufficient. Therefore, we have to consider all communicated data.

C. Legacy Code

Clune and Rood have shown that unit testing can be successfully applied to scientific software, but their case studies were completely new or rewritten applications [15]. HPC applications can be 100k to million lines of code and can have a long history and gone through the hands of generations of researchers [2]. 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. When such tests are not available which is often the case in legacy software, the effort for inducing changes can increase dramatically. But creating a test for a piece of legacy software can be hard and often the software needs to be changed again to make it testable (e.g., by splitting long procedures into smaller ones).

D. Tool support

Fortran is widely used in HPC [2], [3]. But since the overall community is still very small, especially in industry, the 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. However, in recent years, a handful of unit testing frameworks for Fortran emerged (e.g., pFunit[1] [16]).

III. Fortran Test Generator

For the mentioned reasons, creating unit tests in typical HPC environments is not straightforward. Especially the high effort of creating such tests is an obstacle for developers. To significantly reduce the effort of creating unit tests in Fortran, we developed a tool – the *FortranTestGenerator (FTG)* – which automatically creates a basic test driver for a given subroutine isolated from the surrounding application and feeds it with fitting test data.

FTG is written in Python. Its operation is illustrated in Figure 1; it is based on an approach called *Capture & Replay (C&R)*, that is, input data for the subroutine under test is extracted by executing the original application. 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 (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, clearly 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 also generated (Step 5). All the generated code *can* be customized by templates. FTG works with modern Fortran from Fortran95 or later, but it needs the GCC assembler files.

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

A. Capture & Replay

Most effort for creating a unit test is the setup of input data for calling the CUT. In some cases it is almost impossible to manually create a consistent data set. Many HPC application start with a sophisticated initialization phase loading huge data sets.

When working with legacy code, we make use of the existing infrastructure and extract the test data from the running application: FTG captures input data (arguments and global module variables) required by the CUT and called routines recursively when called during execution of the original application. This is done by generating the code which is needed for serializing and storing the input data and inserting it temporarily into the subroutine to be tested. We call this code the *capture code*. The time of capturing can be defined by the developer (or tester) by modifying a special routine generated by FTG. Usually, the input is captured after x times of execution but other criteria can be used.

In a next step, FTG generates a basic test driver for the subroutine to be tested which contains the code for loading the input data and calling the subroutine. This is the *replay code* or *test code*. Currently, the basic test template doesn't contain any kind of validation; checks of the output data need to be included by the developer. Further, the generated code skeleton can be replicated manually to create additional tests by modifying the loaded input data.

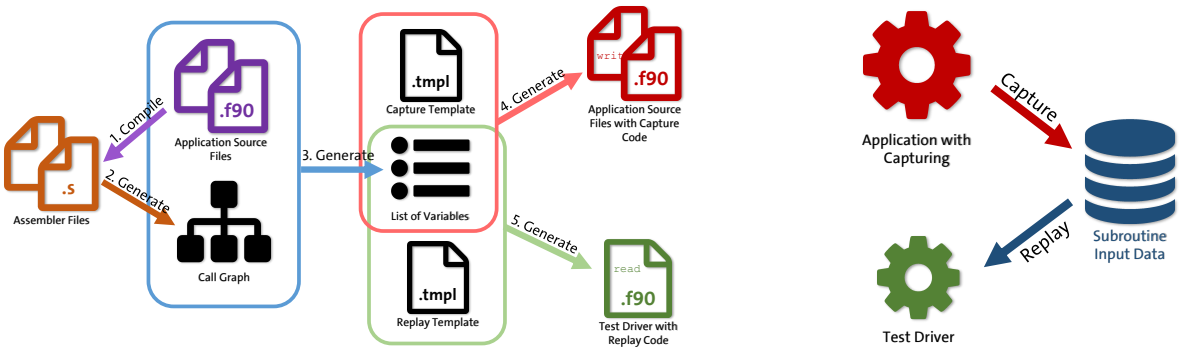
To create a reasonable unit test for a particular case, the capturing should be carried out in the application configuration that creates the best understandable input data. For instance, for grid based applications the grid resolution could be scaled down.

B. Static Source Code Analysis

While our approach is based on collecting "real world" data as test data, it is not useful to capture every single variable existing in an application since HPC applications usually contain huge amounts of data. Instead, the number of captured variables should be reduced to a minimum. For this purpose, FTG conducts a static source code analysis[2] to find out which variables are needed to replay a given subroutine. There are several kinds of data that can be accessed by a Fortran subroutine [17]: variables passed as arguments, module variables of the same module or imported by `USE` statements, common blocks, and external resources (I/O, MPI etc.).

1) 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

[2]Static means that it inspects source code, while dynamic code analysis refers to analysis conducted at runtime.



(a) Source Code Analysis and Generation

(b) Runtime

Fig. 1: General function of FortranTestGenerator

one of its called subroutines or functions. Only those structure components that are actually needed are then captured by FTG.

2) 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 routine or be imported by *USE* statements.

3) Common Blocks: 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.

4) External resources: The access to external resources is hard to address with static analysis since they are often linked during runtime.

MPI communication is an essential topic for HPC applications that we cover in more detail in Section IV. Currently FTG assumes that the subroutine under test can be replayed in the same MPI configuration as during capturing and that it runs more or less synchronized. However, we are working on a solution to also capture and replay MPI communication. This will allow testing of the CUT in a single process but emulating the code as executed in the context of $> 10k$ cores. Therewith, reducing the effort for repeating the test significantly.

For files read by the CUT, we assume that the developer will be able to provide them for the test the same way as for normal application runs. Theoretically, the strategy that we plan for MPI can be applied to I/O similarly allowing to reduce the required data to the actual needed information.

C. Pragmatic Analysis Approach

FTG uses a pragmatic approach for the source code analysis (step 2 and 3). It does not create a complete internal model of the source code (e.g., an abstract syntax tree), but only searches for elements in the code that access data or invoke subroutines or functions.

The call graph needed for traversing all involved subroutines and functions is built up by searching for CALL statements in the assembler code created by the compiler. The assembler code is used, because it contains the resolved names for overloaded procedures. This requires assembler files to be created before running the analysis and currently only the assembler files created by the *GCC* are supported.

Then the Fortran source files are parsed using regular expressions to find interface and type definitions and all kind of accesses to potential input variables. To keep it easy, the source code is normalized before being analyzed, i.e., comments, string literals and line breaks inside long statements are removed.

Solutions based on traversal of abstract syntax trees often have the problem that they cannot deal with language features not supported by the underlying parser. The drawback of our pragmatic approach is that it has the potential to miss something. The general structure of Fortran is quite regular-expression friendly, but it also has some strange characteristics, e.g., that one can use language keywords as identifiers. But for the source files tested so far, our analysis approach worked.

FTG is using static and not dynamic analysis for creating the list of needed variables since the goal is to have a basic input data set that can easily be modified to test different paths of the subroutine. Therefore, we need all variables that could just potentially be needed. Dynamic code analysis can only cover branches that are actually taken at runtime. Thus, it is possible that some variables would be left out and code paths requiring them will not be testable.

D. Customizable Code Generation

The generated code, both for capturing and replay, is based on customizable *templates* using the *Cheetah Template Engine*[3] [18]. That gives us a maximum of flexibility.

Currently we are using a serialization framework developed at CSCS in Switzerland called *Serialbox*[4] for writing variables to the file system. But we also consider other implementations, especially using *HDF5*[5] [19] to store the variables in a portable fashion, or even generate plain Fortran with hard-coded variable initialization containing the captured data.

[3]https://github.com/cheetahtemplate/cheetah
[4]https://github.com/C2SM-RCM/serialbox
[5]https://www.hdfgroup.org/HDF5

The generated test driver code is also customizable. One can tailor it in a way that it is integrated into to the main application or an existing test suite, or such that a stand-alone program is created. One can prepare some predefined checks or build a test case for an existing unit test framework such as *pFUnit* [16]. The public release of FTG will contain a set of templates that can be used without further modifications.

E. User Interface

The user interface of the FortranTestGenerator consists of a configuration file and a command-line interface. In the configuration file the location of the source and assembler files are specified. Further, one can define modules, routines or variables that should be excluded from the analysis. Then, only the subroutine to be tested and the module it belongs to need to be given via the command line interface, for example for generating the capture code:

```
$> ./FortranTestGenerator -c <module>
<subroutine>
```

Compiling and running the generated code must be done manually by the user. Thus, one will be able to change the code, for example to define the time of capturing. Beyond that, the user is able to edit the code templates which requires some familiarization with the Cheetah template syntax [18]. To improve the tool handling, the configuration and command-line options as well as the template API are still a matter of change.

F. Limitations

The source code analysis only works for Fortran as we are aiming at HPC applications written in Fortran. Further, we don't believe that our analysis approach with regular expressions would work for C or C++. Libraries that are called by the CUT written in other languages are usually not a problem as long as the library's internal state doesn't influence a subroutines result. This is especially true for numerical linear algebra or other numerical solvers. Also, the generated code can always be edited by the developer if some variables are missed or a library needs explicit initialization.

There are some inherent drawbacks of static source code analysis as it cannot handle any kind of runtime polymorphism such as procedure pointers or object-oriented inheritance. We haven't implemented any solution yet but we can think of a human-tool cooperative approach [20] where the tool asks the developer for advice during analysis.

Another general problem are recursive data types such as linked lists or trees which require knowledge about the data type to be serialized correctly. Our approach is based on capturing only variables of primitive datatypes which are usually the leaves in a data structure tree (e.g. something like `t1%t2%t3%i4` where `t1`, `t2`, and `t3` are variables of derived types and `i1` an integer array). In recursive data structures, there are no such natural leaves. At runtime, a linked list might have an end, but in a graph you can end up in a cycle. There are solutions for that problem, but we haven't implemented anything yet.

While members of a structure are only serialized if needed, FTG always captures arrays completely and does not support any kind of limitation even if a subroutine only uses certain parts or just a single element of an array. Since indices of accessed array elements are usually computed at runtime, static source code analysis cannot find out which parts of an array is needed in a subroutine. Here again, one can think of some kind of assistance by the developer. However, having the complete input data at disposal allows developers to change input parameters which leads to accesses of regularly not accessed parts of the array. Therewith, it increases the flexibility for the tester to conduct tests which have not been captured explicitly with the full application.

Subroutines extracted by FTG have to be replayed in the same MPI configuration as during capturing. In the next section, we present an approach to overcome this limitation.

IV. Handling Parallelism

When testing parallel applications with unit tests, we want to prevent running large configurations. Sometimes we want to study only one process in isolation. Therefore, it can be necessary to fake the input and check the output of communication conducted by the CUT directly or indirectly. MPI applications spawn concurrently executed processes; typical communication patterns are point-to-point between two peers, one-to-many/many-to-one, and many-to-many.

When testing such an application, the input of one process depends on the data communicated by others, similarly generated output are not only changes in local memory but also data send to peers. Since we have experience in developing tracing tools for performance analysis [21], we are working on an MPI intercepting library and a dummy MPI implementation to enable capturing of MPI applications and replay of individual processes.

A. Example MPI Program

In Figure 2, a primitive example code is provided to demonstrate some issues when creating a test driver for MPI. The code includes calls to initialize and shutdown MPI (`MPI_Init()`, `MPI_Finalize()`) and to identify the process number (`MPI_Comm_rank()`). When the compiled program is executed with two processes, certain branches are executed by both processes and some only by Rank 0 or Rank 1. First, the processes exchange information using a single point-to-point communication (`MPI_Send()`, `MPI_Recv()`) where Rank 0 sends the integer value 42 to the other process. Afterwards Rank 1 broadcasts "Hello World!" to all processes of this program. If run with more than two processes, the application will deadlock in the `MPI_Recv()` as all $ranks > 1$ will wait for a message that will never be received.

In this example, the routines `sendResult` and `recvResult` are communicating with each other and the printed values are depending on data received from the counterpart running on another MPI process. That makes it impossible to test a routine individually with the C&R approach without considering the data exchanged via MPI.

```
PROGRAM hello

IMPLICIT NONE

include 'mpif.h'

INTEGER, PARAMETER :: CHAR_LENGTH = 140
INTEGER rank, ierr, status(MPI_STATUS_SIZE)

CALL MPI_Init(ierr)
CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

IF (rank == 0) THEN
  CALL sendResult()
ELSE
  CALL recvResult(5)
END IF

CALL MPI_Finalize(ierr)

CONTAINS

SUBROUTINE sendResult()
  INTEGER calc
  CHARACTER(len=CHAR_LENGTH) buf

  calc = 42 ! some result
  CALL MPI_Send(calc, 1, MPI_INTEGER, 1, 4711, &
    & MPI_COMM_WORLD, ierr)
  ! some data to broadcast to all other processes
  CALL MPI_Bcast(buf, CHAR_LENGTH, MPI_CHAR, 1, &
    & MPI_COMM_WORLD, ierr)
  PRINT*, TRIM(buf) ! process 0 prints Hello World!
END SUBROUTINE sendResult

SUBROUTINE recvResult(factor)
  INTEGER, INTENT(in) :: factor
  INTEGER calc
  CHARACTER(len=CHAR_LENGTH) buf

  CALL MPI_Recv(calc, 1, MPI_INTEGER, 0, 4711, &
    & MPI_COMM_WORLD, status, ierr)
  PRINT*, calc * factor ! process 1 prints 210
  buf = "Hello␣World!"
  CALL MPI_Bcast(buf, CHAR_LENGTH, MPI_CHAR, 1, &
    & MPI_COMM_WORLD, ierr)
END SUBROUTINE recvResult

END PROGRAM hello
```

Fig. 2: MPI code example for two processes

B. Capturing

For the purpose of capturing, a PMPI library is currently in development that intercepts the MPI call. Pseudocode (in C-style) for the implementation of the intercepting functions is given in Figure 3 for `MPI_Recv`. It intercepts all MPI calls and serializes and stores its arguments similar to performance analysis tools.

Additionally, we store the actual input and output data and retrieve a *unique data identifier (UDID)* that points to the serialized data. For `MPI_Recv()`, a special case is that its semantic allows to receive less elements than specified with count. Therefore, the status has to be checked to retrieve the actual number of elements that have been received. Then we record that a receive has been executed to the data table – together with the arguments and the identifier. The number of elements retrieved are stored sequentially in the data table starting with the UDID. All wrapped functions look structural similar.

```
int PMPI_Recv(void *buf, int count, MPI_Datatype type,
  int source, int tag, MPI_Comm com, MPI_Status *status){
  int ret;
  // call the original function
  ret = MPI_Recv(buf, count, type, source, tag, com, status);

  if (ret == MPI_SUCCESS){
    int rec_count;
    MPI_Get_count(status, type, & rec_count);
    // store all data into a table:
    int udid = capture_record_data(buf, count, type);
    // store the receive args into a table
    capture_record_recv(count, rec_count,
        type, source, tag, com, status, udid);
  }
  return ret;
}
```

Fig. 3: Pseudocode for MPI tracing to record unit tests

```
int MPI_Comm_rank(MPI_Comm comm, int *rank){
  int ret;
  // return the recorded rank within comm
  replay_comm_rank(comm, rank, & ret);
  return ret;
}

int MPI_Recv(void *buf, int count, MPI_Datatype type,
  int source, int tag, MPI_Comm com, MPI_Status *status){
  int ret, udid;
  int rec_count; // recorded count
  // check if we find a matching receive
  read_recv(count, type, source, tag, com,
              status, & ret, & rec_count, & udid);

  if (ret == MPI_SUCCESS){
    // read rec_count elements into the buffer
    read_data(udid, buf, rec_count, type);
  }
  return ret;
}
```

Fig. 4: Pseudocode for MPI library to run unit tests

C. Replay

For replay of code, we will provide a new MPI-test library. In contrast to existing libraries, it does not conduct any communication but reads the data from recorded information. Figure 4 shows examples for `MPI_Comm_rank()` and `MPI_Recv()`. Whenever a call to an MPI function is issued, the data stems from the capture file instead of the runtime information. The implementation is trivial for `MPI_Comm_rank()`, that can also take into account invalid communicators. However, the message matching of MPI requires that certain information, i.e., communicator, receiver and tag must match to receive the message. The implementation will simply return matching messages sequentially. If the user changes the CUT to issue combinations for which no input exist, or to receive more messages than have been recorded, the implementation will return an error.

V. Related Work

There are quite some papers discussing software testing in scientific and high performance computing. Systematic literature reviews have been conducted by Kanewala and Biemann [11] and by Heaton and Carver [12].

The idea of storing and reloading the state of a program is not new. It is the essence of checkpoint and restart strategies [22] which are widely used in HPC, both on system and/or application level. These mechanisms not only increase a programs fault tolerance but can also support testing since they allow a faster reproduction of certain states of an application. But they usually store the state of the whole application at defined checkpoints and cannot not focus on specific code parts. That means that the amount of data is usually much higher than with our approach and you still have to run the whole application to test a single subroutine. General checkpoint/restart mechanisms are more useful for system level testing but suboptimal for unit testing.

There are also various tools which capture and replay small code parts. Most of them are focused on performance analysis and tuning.

The Code Isolator by Lee and Hall [23] is based on the *Stanford SUIF compiler* and was primarily used for studying cache behavior. Therefore, it even preserves the cache state of the original execution when replaying a code slice. But since it relies on the SUIF compiler, it doesn't fit very well into the standard tool chain and anyway, it doesn't seem to be available anymore.

CERE by Castro et al. [24] is available on Github[6] and also focuses on performance analysis. It doesn't analyze and modify the original source code of an application but the *intermediate representation (IR)* provided by the *LLVM compiler*. Due to this, it works with any language supported by LLVM. In contrast to FTG, it does not let the user pick the code part to be extracted but identifies performance hotspots itself. Since it works on the LLVM-IR, it is not intended that the user modifies the generated code.

The latest tool is KGen[7] by Kim et al [25]. As FTG, it is also written in Python and uses static source code analysis and a C&R approach to extract stand-alone kernels out of a large Fortran application. It has a lot of configuration options and is more automated than FTG, but that also means that the scope of user interaction is limited to the command line options. The generated code is spread across all involved modules and is not intended to be modified by the user. Other than FTG, it doesn't analyze the usage of type components but stores all referenced data structures completely. In some applications this can lead to a much higher data volume.

All of the mentioned tools have in common that the code generation or instrumentation can not be influenced by the user while FTG uses customizable templates for the code generation.

Clune et al. have proposed that it would be useful to consider MPI parallel applications for testability [26]. The work has been published concurrently to our development and doesn't contain a design proposal or even a proof of concept.

When looking outside of the HPC area, we also find some similar approaches. There are, for example, a couple of Java tools using C&R for automatically creating unit tests from system tests [27], [28]. Beyond that, Elbbaum et al. have created a general framework for applying this methodology in Java [29].

To evaluate scaling behavior of MPI code, many C&R solutions have been developed such as ScalaTrace [30] or MPIWiz [31]. These tools can also be used to enable deterministic re-running of application to simplify debugging [32]. Simulators allow replaying of MPI application traces in virtual configurations and, thus, to conduct what-if analysis [33], [34]. The existing solutions require to restart the full application and do not allow to only run one process in isolation for testing purposes, also they are purposely not designed to vary the input.

VI. Future Work

We are constantly working on improving the FortranTestGenerator and will soon publish the first release on GitHub. This also includes that we extend our set of code templates. For example, in the future, we will provide test templates that already contain some basic checking of output variables.

We are also planning to migrate from the currently used serialization framework to HDF5 [19]. That will give the user more options to exchange, view and edit the captured data.

Another major milestone will be to integrate the capture and replay of MPI communication into FTG. This will increase the tool's complexity and opens the question how the overall goal of an easy unit test creation can still be fulfilled. This also applies to our plans to integrate a human-tool cooperation approach [20] to enable FTG to handle runtime polymorphism.

So far, we have used FTG only for extracting subroutines from the earth system model ICON [35] to test it under real-world conditions. A more extensive evaluation has to be conducted. Besides the feasibility of the underlying concepts, the usefulness of our approach is of particular interest. Therefore we will cooperate with domain experts to create meaningful test cases.

VII. Conclusion

Although testing of scientific software is generally considered to be problematic (e.g., due to lack of test oracles), we believe that HPC developers would benefit from having fine-grained, automated unit tests as they provide a quick feedback when doing daily tasks such as debugging, refactoring or optimization. But as every other utility, unit testing will only be considered when the benefit will outweigh the effort of creating such tests. One major effort for creating unit tests is the set up of appropriate test data. In HPC, this is especially costly since applications often handling large and complex data structures.

With FortranTestGenerator we have created a Python tool which aims on reducing the effort of creating unit tests for existing Fortran applications. FortranTestGenerator automatically generates a basic test driver and code for extracting a consistent set of input data from the original application. This approach called Capture & Replay (C&R) is not new, but as

[6]https://github.com/benchmark-subsetting/cere

[7]https://github.com/NCAR/KGen

far as we know, FortranTestGenerator is the first tool of its kind which explicitly aims on generating unit tests for Fortran applications. Beyond that it implements a very pragmatic approach of static code analysis: it inspects assembler and source files with regular expressions to find those variables which are actually used by the subroutine under test. The result of the analysis is used to reduce the amount of data captured and replayed for running the test isolated from the embedding application. In contrast to other code generating tools, the code created by FortranTestGenerator is based on templates and, therefore, fully customizable. That makes it easy to integrate both the capturing process and the unit tests into an existing HPC software infrastructure. It must be noted that FTG only creates basic test drivers. It is up the developer to extend the test driver with meaningful checks.

The parallel programming model based on MPI is an integral part of HPC environments. When testing the behavior of a piece of an HPC application, one also has to consider data exchanged via MPI. Therefore, we are also working on applying the C&R approach to MPI communication. We have outlined the design of two MPI wrapper libraries, one for intercepting MPI calls and one for replaying the captured messages. This will enable us to emulate multi-process executions while we actually testing a subroutine only on one process to reduce complexity.

REFERENCES

[1] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Wiley Publishing, 2011.

[2] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz, "Understanding the High-Performance-Computing Community: A Software Engineer's Perspective," *IEEE Software*, vol. 25, no. 4, 2008.

[3] J. Carver, R. Kendall, S. Squires, and D. Post, "Software Development Environments for Scientific and Engineering Software: A Series of Case Studies," in *29th Int. Conf. on Software Engineering (ICSE)*, 2007.

[4] D. Kelly, R. S, R. Saint, and P. Floor, "The Challenge of Testing Scientific Software," in *Proc. of the Conf. for the Association for Software Testing*, 2008.

[5] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, 1982.

[6] U. Kanewala and J. Bieman, "Techniques for testing scientific programs without an oracle," in *5th Int. Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)*, 2013.

[7] M. Feathers, *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[8] M. Fowler, "Unit Test," May 2014, Blog Post. [Online]. Available: http://martinfowler.com/bliki/UnitTest.html

[9] R. Osherove, *The Art of Unit Testing: With Examples in C#*. Shelter Island, NY, USA: Manning, 2013.

[10] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," *ACM SIGPLAN notices*, vol. 46, no. 4, pp. 53–64, 2011.

[11] U. Kanewala and J. M. Bieman, "Testing scientific software: A systematic literature review," *Information and Software Technology*, vol. 56, no. 10, 2014.

[12] D. Heaton and J. C. Carver, "Claims About the Use of Software Engineering Practices in Science: A Systematic Literature Review," *Information and Software Technology*, vol. 67, Nov. 2015.

[13] D. Lecomber and P. Wohlschlegel, "Debugging at Scale with Allinea DDT," in *Tools for High Performance Computing 2012*. Springer, 2013.

[14] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch, "MARMOT: An MPI analysis and checking tool," *Advances in Parallel Computing*, vol. 13, 2004.

[15] T. Clune and R. Rood, "Software Testing and Verification in Climate Model Development," *IEEE Software*, vol. 28, no. 6, 2011.

[16] M. Rilee and T. Clune, "Towards Test Driven Development for Computational Science with pFUnit," in *Proc. of the 2nd Int. Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, 2014.

[17] J. C. Adams, W. S. Brainerd, R. A. Hendrickson, R. E. Maine, J. T. Martin, and B. T. Smith, *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*. Springer, 2014.

[18] I. B. Tavis Rudd, Mike Orr, "Cheetah: The Python-Powered Template Engine," in *10th Int. Python Conference*, February 2002.

[19] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An Overview of the HDF5 Technology Suite and Its Applications," in *Proc. of the EDBT/ICDT 2011 Workshop on Array Databases (AD'11)*, New York, NY, USA, 2011.

[20] T. Xie, "Cooperative Testing and Analysis: Human-Tool, Tool-Tool and Human-Human Cooperations to Get Work Done," in *12th Int. Working Conf. on Source Code Analysis and Manipulation (SCAM 2012)*, Sept 2012.

[21] J. Kunkel, "HDTrace – A Tracing and Simulation Environment of Application and System Interaction," Univ. Hamburg, Germany, Tech. Rep., 2011.

[22] K. M. Chandy and C. V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," *IEEE Transactions on Computers*, vol. C-21, no. 6, June 1972.

[23] Y.-J. Lee and M. Hall, *A Code Isolator: Isolating Code Fragments from Large Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[24] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, "CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization," *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 1, Apr. 2015.

[25] Y. Kim, J. Dennis, C. Kerr, R. R. P. Kumar, A. Simha, A. Baker, and S. Mickelson, "KGEN: A Python Tool for Automated Fortran Kernel Generation and Verification," in *Proc. of the Int. Conf. on Computational Science (ICCS 2016)*, 2016.

[26] T. Clune, H. Finkel, and M. Rilee, "Testing and Debugging Exascale Applications by Mocking MPI," in *Proc. of the 3rd Int. Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, 2015.

[27] A. Orso and B. Kennedy, "Selective Capture and Replay of Program Executions," *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, May 2005.

[28] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic Test Factoring for Java," in *Proc. of the 20th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'05)*, 2005.

[29] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and Replaying Differential Unit Test Cases from System Test Cases," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, Jan 2009.

[30] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, 2009.

[31] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker, "MPIWiz: Subgroup Reproducible Replay of MPI Applications," *ACM Sigplan Notices*, vol. 44, no. 4, 2009.

[32] C.-E. Hong, B.-S. Lee, G.-W. On, and D.-H. Chi, "Replay for debugging MPI parallel programs," in *Proc. of the 2nd MPI Developer's Conference*. IEEE, 1996.

[33] J. Kunkel, "Using Simulation to Validate Performance of MPI(-IO) Implementations," in *Proc. of the 28th Int. Supercomputing Conference, (ISC)*, 2013.

[34] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely, "PSINS: An open source event tracer and execution simulator for MPI applications," in *European Conf. on Parallel Processing*, 2009.

[35] G. Zängl, D. Reinert, P. Rípodas, and M. Baldauf, "The ICON (ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core," *Quarterly Journal of the Royal Meteorological Society*, vol. 141, no. 687, 2015.