Procedia Computer Science

Volume 80, 2016, Pages 587–598

ICCS 2016. The International Conference on Computational Science

In Situ Data Infrastructure for Scientific Unit Testing Platform*

Zhuo Yao,[a] Yulu Jia,[a] Dali Wang [†,b] Chad Steed,[c] Scott Atchley[c]

[a]*Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37966, USA*

[b]*Climate Change Science Institute, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA*

[c]*Computational Sciences and Engineering Division and National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, USA*

Email address: {zyao5, yjia}@vols.utk.edu, {wangd,steedca,atchleyes}@ornl.gov.

[†] *Corresponding author. Tel.: +1-865-266-0748; fax:+1-865-241-3685. wangd@ornl.gov*

Abstract

Testing is a significant software development process for the management of software systems and scientific code. However, as the complexity of scientific codes increases, extra checks are needed to monitor impacts to dependent models and to verify system constraints. The software complexity also impedes the efforts of module developers and software engineers to rapidly develop and extend their code. Recently, we have developed an automatic methodology and prototype platform to facilitate scientific verification of individual functions within complex scientific codes. With this system, the scientific module builders are able to track variables conveniently in one module or track variables' changes among different modules. In this paper, we present a procedure for automatic unit testing generation. For the interest of a general audience of this conference, we are emphasizing the technical details of integrating the In Situ data infrastructure into our platform. At the end of this paper, we have included an implementation of unit testing for the ACME Land Model (ALM) to demonstrate the usefulness and correctness of the platform. We have also used single- and multipoint checks to demonstrate the efficient variable tracking capability of this platform.

Keywords: Unit testing, compiler based analyzer, scientific codes, In Situ, ACME land model

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.344

1 Introduction

Software testing is an essential part of software development and is an important verification method to reveal program deficiencies. Test cases may be generated by sampling the execution space either randomly or by aiming at specific objectives that derive from the expected behavior, the program structure, or some information about software faults [1]. Many automated test generation techniques are proposed and used to improve testing effectiveness and efficiency [2-7].

In most development conditions, software developers and testers are responsible for ensuring the code is functionally correct. However, for large and complex software, there is an extra need to check the active impact that submodules have on other dependent modules of the source code and the system constraints to verify the test procedures.

Large, complex systems have configurations that require multiple, time-consuming test procedures. Test tools must take into account the testing and configuration time as well as any substantial overhead generated by the automatic platform that forces software engineers to switch back to the former objective.

After experimenting with several test tools in scientific code, we repeatedly had issues with scalability and usability. Based on former experiences, we wanted to build a test platform that would

- be widely used by developers and module builders to fix problems and collect variables,
- integrate smoothly into the existing workflow, and
- produce test suites automatically and provide validation.

Besides, large-scale simulations have been choosing in situ methods as their primary analysis mechanism [12, 13]. The primary reason is that I/O bandwidth to storage has not kept pace with numerical computing data generation [14], and it is a relatively expensive resource to scale-up compared to compute cycles [15]. While it is possible to achieve performance with post-processing [16], the relative cost to achieve the same level of performance using in situ analysis will be much higher. Also, it is problematic to store large-scale data for later analysis on parallel file systems, as data sets are typically flushed to slower archival systems to make room for other users, which decreases post-processing performance [19]. Tikhonova et al. [17] and Ahrens et al. [18] have been working on lower hardware and storage requirements of in situ processing.

For the past 2 years, we have developed the methodology and procedures to generate scientific function unit tests for scientific codes [8,9]. The two papers present a method to create direct linkage between site measurements and the process-based Community Land Model (CLM), and then analyze using post-processing workflow. But the platform in first paper only realized a few functional module testing while the later paper realizes all functional modules testing with only PFT level variables. In this paper, we present our new effort to integrate an In Situ infrastructure into our scientific unit testing platform. To our best knowledge, it's the first work on an In Situ infrastructure. Specifically, we first describe the overview procedure of scientific unit testing, and then we focus on the significance and technical details of integrating an In Situ data infrastructure into our testing platform. The contributions of this paper include the following:

- A set of procedures that have resulted in an automatic test case generation platform and In Situ infrastructure (Sect. 2).
- Implementation of the unit testing platform and In Situ infrastructure. This platform builds a program-testing ecosystem through workflow integration, supports every single module testing, and monitors variables under one specific module or through different modules using In Situ infrastructure (Sect. 3).
- An In Situ communication experiment result based on the platform (Sect. 4).

2 Scientific unit testing procedure

In this part, we describe our methods to integrate an In Situ infrastructure into our scientific unit testing platform. Two general steps are needed for the unit testing procedure: (1) automated unit test case generation and (2) Common Communication Interface (CCI) verification. Sections 2.1 and 2.2 explain these two steps in detail.

2.1 Automated unit testing case generation

Building unit test modules are not merely a means to transform source code into executable and other artifacts. However, building separated unit test modules are used more widely in practice because they capture an important architectural dependency structure between modules and external libraries. Therefore, to make modules run independently, the most important steps are to preserve the behavior of the program and update the dependencies of the original modules to depend on only the needed constituent submodules and external libraries.

Libraries dependency

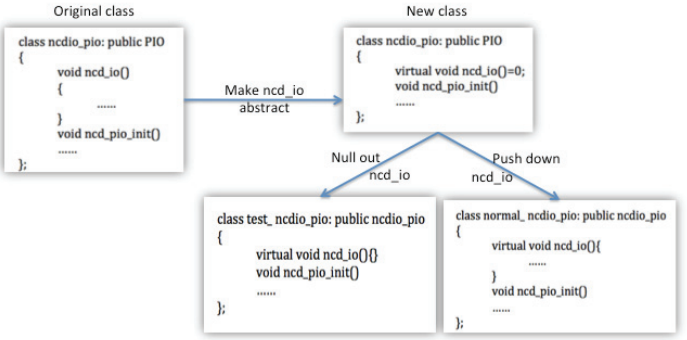

Figure 1: Push down method

Many scientific codes are combined with low-level MPI or parallel IO calls to external libraries, which can improve the efficiency of computer programming and model development. But the cost of testing MPI or parallel IO methods can be high, because they are used in distributed memory systems and the testing purpose is to validate the mathematical correctness of each module. One of the primary objectives for designing the scientific testing framework is to modulate subroutines and let them run independently in a shared memory environment. Furthermore, we seek to make our framework platform portable and easy to use, so we need to limit the number of dependent libraries. Generally, we have two options: the push down dependency method and the overwrite method. If the dependencies are pervasive, the push down method would be the best choice because it helps testers to split problematic dependencies, function *ncd_io* in Figure 1, from most of the classes, making unit testing less burdensome. When using the push down dependency method, we first make the target function *ncd_io* abstract. Then we create a subclass that is *normal_ncdio_io* and push down all problematic dependencies into that class. Last, we create a test case that is *test_ncdio_pio* subclass original class to test all its methods [10]. But at times, the dependencies in unit testing are rather localized that we only have one single method that we need to replace. In this case, the second option would be the best choice among object-oriented programs, since it can prevent side effects in our platform and track values that passed through this call. When using the overwrite method, we find all dependencies in the target unit module, check all dependency libraries' functions, create a new class with these same purpose functions and make sure all these new functions are easy to build, add this

new class in the target module and replace dependency libraries' functions with new classes' functions, and adapt the corresponding parameters. It is an ideal way to break dependencies on static methods. In this paper, we combine the two methods together. First we apply push down method to null out the *ncd_io* function, and then replace *ncd_pio* with simple IO functions to deal with data.

Data dependency and data flow generation

There are many data transformations between modules and numerous data structure initializations in the beginning of the legacy code. To eliminate the data dependence problems and solve data initialization problems, we adopt a recursive compiler-based code analyzer, which analyzes the unit module's data flow and also processes submodels' initialization problems to make the single unit runnable. Based on the access method of data, the analyzer divides data into three groups: the *write_only* group, in which data will only be written; the *read_only* group, in which data will only be read; and the *mod_only* group that is written and read by the same module. Figure 2 shows the flowchart of the dataflow analyzer.

The algorithm of dataflow generation contains four steps. The first step includes parsing the source code of the targeted unit to obtain and record all callee modules. A call graph is generated where the targeted unit is parent and callees are children. Next we analyze data flow of the targeted unit and categorize its input and output data into three data groups (*read_only*, *write_only*, and *mod_only*). The initialization function of the unit is also recorded. If a callee is detected by Step 1,go back to the first step and apply the same algorithm on every callee. If no callee is discovered, go to the next step. Lastly, we collect all initialization functions and the three data groups including all the callees. They are the required initialization functions and data to drive the targeted unit.

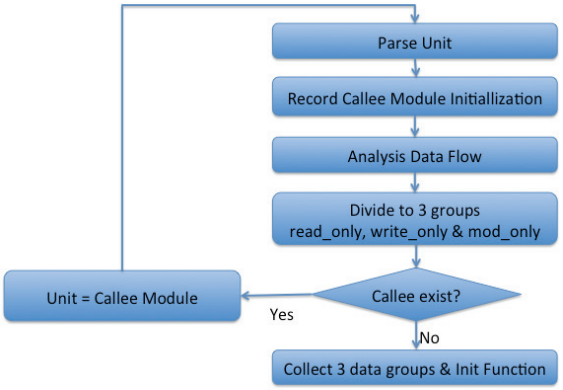

Figure 2: Flow chart of the dataflow analyzer

We apply this method to all unit modules to get input data and the submodule call list to cut the dependency.

2.2 Verification via In Situ infrastructure

Verification is an essential part of scientific unit testing. Scientific codes generally are very sparse in unit testing. In our case, we capture the inflow and outflow data from a benchmark case simulation and then use them to validate the implementation of our unit testing cases. Technically, we build a dataflow "inspector" at the beginning and end position on the module of interest within the scientific code. The dataflow "inspector" will inspect all variables from the *read_only*, *write_only*, and *mod_only* lists, created through the process of "data flow generation" (Sect. 2.1). Then we build a data movement service, which uses In Situ data communication. The goal is to be able to inspect variable

values inside the simulation code while the simulation code is running without I/O needs. In the following subsections, we present technical details of our system's In Situ communication scheme, data packaging format, and external data exchange format.

Communication scheme

Based on an In Situ data infrastructure, such as CCI [11], a communication service can be developed by running an analysis code as a separate application from the simulation application, which conducted a single computer. The data communication design is illustrated in Figure 3. The analysis code interacts with the simulation code, and the user specifies the variable names and simulation time window from the analysis code. The analysis code sends the request to the simulation code. After each simulation module is finished, the simulation code checks if the data generated in this module are requested by the analysis code. If the data packages are requested, they are packed into a memory buffer and sent to the analysis code over the network. The analysis code listens on its own shared data endpoint. If data arrives, the analysis code unpacks the data and stores the data packages in separate memory buffers or external formats.

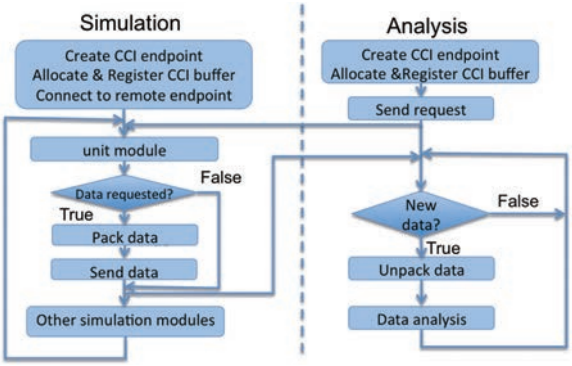

Figure 3: In Situ data communication format

Data packaging format

To transfer data to the remote analysis code, the sender first packs data into a buffer so the data packages are contiguous in memory. The data buffer has the size of the data stored at the beginning as a size *t* value. Variables are stored one after another. For each variable, the metadata such as name, data type, element size, dimensions, and total length of the variable are stored, followed by the actual data values.

Exchangeable common data form

One important aspect of the In Situ infrastructure is the capability to save the data into permanent disk space. Although many domain-specific data formats are available, in our research, we adapted any self-describing, machine-independent data formats, such as Networked Common Data Format (netCDF), briefly described here. NetCDF supports the creation, access, and sharing of array-oriented scientific data. With these features, many groups have adopted netCDF as a standard for representing some forms of scientific data. The self-describing capability is achieved by its own description language, called network Common data form Description Language (CDL). A CDL consists of three parts: dimensions, variables, and data. A dimension has a name and a length. A variable has a name, a data type, and a shape described by its list of dimensions. The optional data section of a CDL is where

netCDF variables may be initialized. More information on netCDF and CDL can be found online at http://www.unidata.ucar.edu/software/netcdf/docs/.

3 Case study

In this part, we present an implementation of the In Situ infrastructure for a scientific code.

3.1 A scientific code

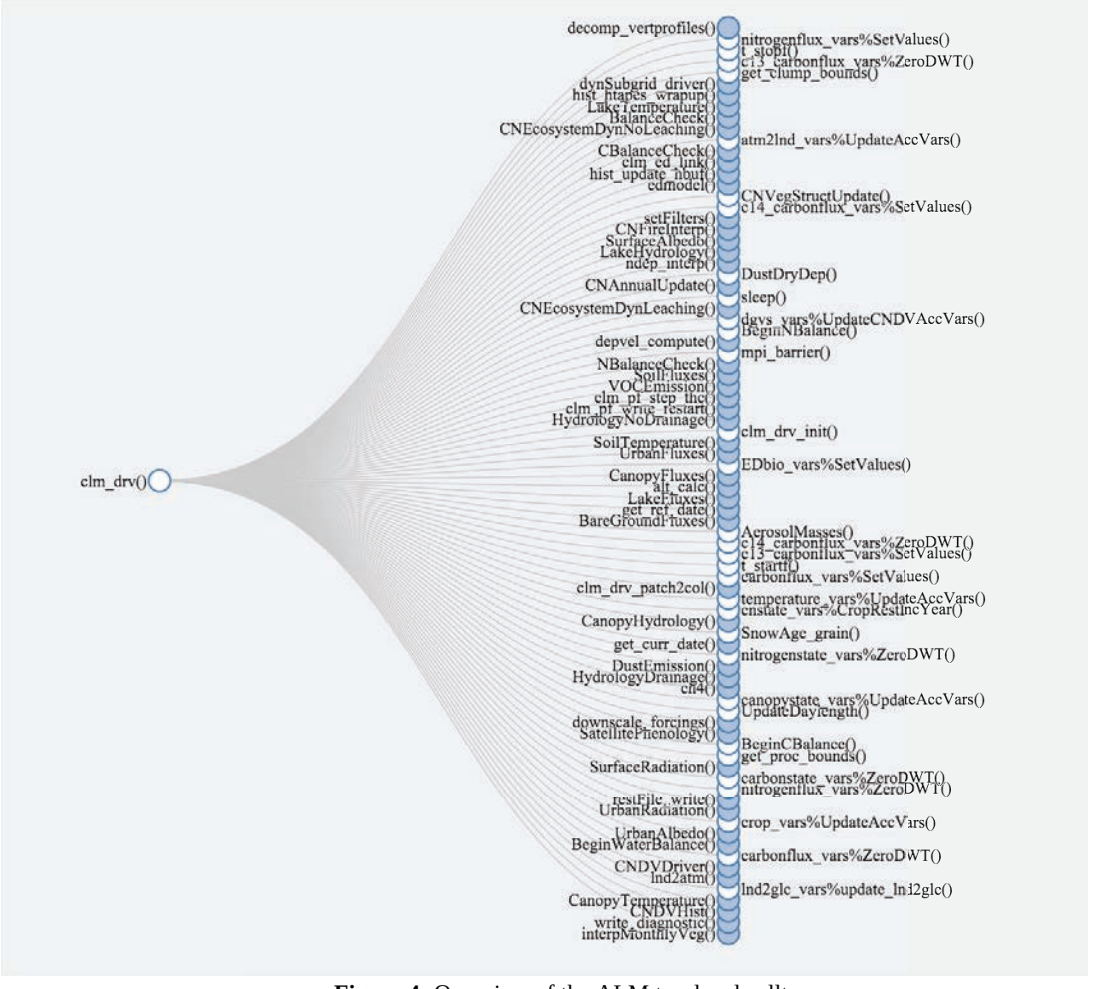

Figure 4: Overview of the ALM top-level calltree

Over the past several decades, researchers have made significant progress in developing high fidelity earth system models to advance our understanding of the earth system in the United States and to improve our capability to better project future scenarios. The Community Earth System Model (CESM, http://www2.cesm.ucar.edu) is one of the leading earth system models. Since 2014, the US Department of Energy (DOE) has branched off the CESM main repository and created its own model, under a nationwide project, called Accelerated Climate Modeling for Energy (ACME). Within ACME,

the ACME Land Model (ALM), originally branched from the Community Land Model (CLM), is the active component to simulate surface energy, water, carbon, and nitrogen fluxes and state variables for both vegetated and non-vegetated land surfaces.

ALM contains more than 1,800 source files (over 350,000 lines of source code) and heavily relies on external libraries for advanced numeric calculations, data communication, and parallel IO, etc. Figure 4 shows the top-level call-tree of ALM.

3.2 Platform implementation

The ALM source code runs on the ORNL Institutional Cluster (OIC), a multiprogrammatic heterogeneous federated cluster on the Red Hat Enterprise Linux (RHEL) 6 64-bit system. The unit testing platform, conducts a single processor, runs on a laptop with four processors (2.67 GHz Intel i7) and 4 GB of memory, running 32-bit Ubuntu GNU/Linux 12.04. The ALM unit testing platform over an In Situ infrastructure (as shown in Figure 5) has five stages: file, communication, preprocess, unit testing, and verify.

At the file stage, there is a dataflow analyzer; its goal is to split the data dependency between modules. It analyzes and stores dataflow using a dataflow generation algorithm. In addition, it can access any content of the original legacy code (see the algorithm listed in 2.1). When this analyzer, run on a unit testing server, receives original source module code, it first collects all submodules, and data groups go through it and its submodules based on a depth-first traversal algorithm. Then it inserts all data declaration and output function to the beginning and ending of the original module to be an "inspector," as explained in Sect. 2.2. Due to the ALM source code's complexity and less standard forms, we need to pay extra efforts to deal with specific modules and variables.

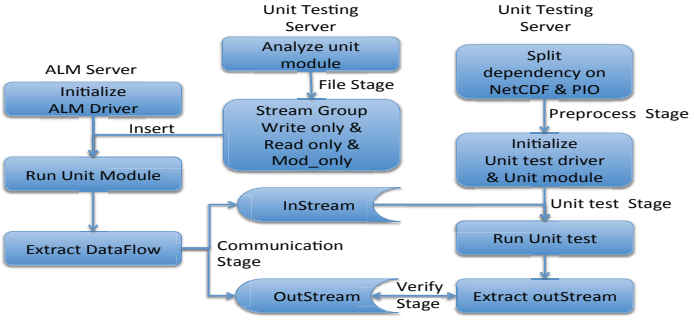

Figure 5: Overview of the In Situ infrastructure for the ALM unit testing platform

Take the *CanopyFluxes* module as an example: this unit calls 18 subroutines, and four of them have expanded subroutines based on Figure 6. To make this unit runnable without any prerequisite variable initialization problems, the compiler-based analyzer will find and analyze all subroutines' initialization methods and extract three data groups (*read_only*, *write_only*, *mod_only*). The analyzer combines and divides *CanopyFlux*'s data groups with its subroutines' data groups into two parts based on their roles: the input data part including the *read_only*, the *write_only*, and the *mod_only* groups; and the output data part including the *write_only* and the *mod_only* groups. After grouping, the analyzer inserts the data output methods to the *CanopyFluxes* module.

The second stage is the communication stage. ALM source code is written in Fortran, while CCI only provides a C interface. Therefore, in our unit testing server, we first implemented a C function to package the data required for unit testing. Then, we implemented a Fortran wrapper that runs in the ALM server to call the C function using Fortran/C interoperability defined by the Fortran 2003 standard. Specifically, an International Organization for Standardization (ISO) C BINDING module

enables declaration of C-compatible variables in Fortran. The attribute tells the compiler that a function is bound to a C function. In these two steps, the instrumented and augmented ALM code and the unit testing platform communicates serially based on the scheme described in Sect. 2.2.

The main task at the preprocess stage is to separate dependency on the parallel IO library (PIO), to make the unit test platform more portable. Technically, the following steps are implemented in our effort: (1) identify the dependency needs to be separated, because the cost in the unit test period with PIO is too high; and (2) add a new class that includes easy IO operation and replace all PIO functions with new IO functions. The result is a platform that makes it easy to deal with data collection from the original ALM simulation code.

At the unit testing stage, the unit test driver simulates a module's working background to run all unit modules. The behaviors of the test driver include: initializes global parameters and constants, defines variables used for subroutines and sets values based on original ALM code configuration, connects ALM simulation and loads submodules and input data collected from the communication stage using override methods described previously, and runs unit test and creates a new output file.

At the verify stage, we compare the result from the unit test with the results from the original code simulation to verify if the unit modules ran correctly. Take the *CanopyFluxes* module as an example: this unit has one data output based on the communication stage and one data output from the unit testing stage. To make sure the platform is reliable and the unit module generated from the last stage is useful and correct, the Python verification tool will compare every single variable in these two files.

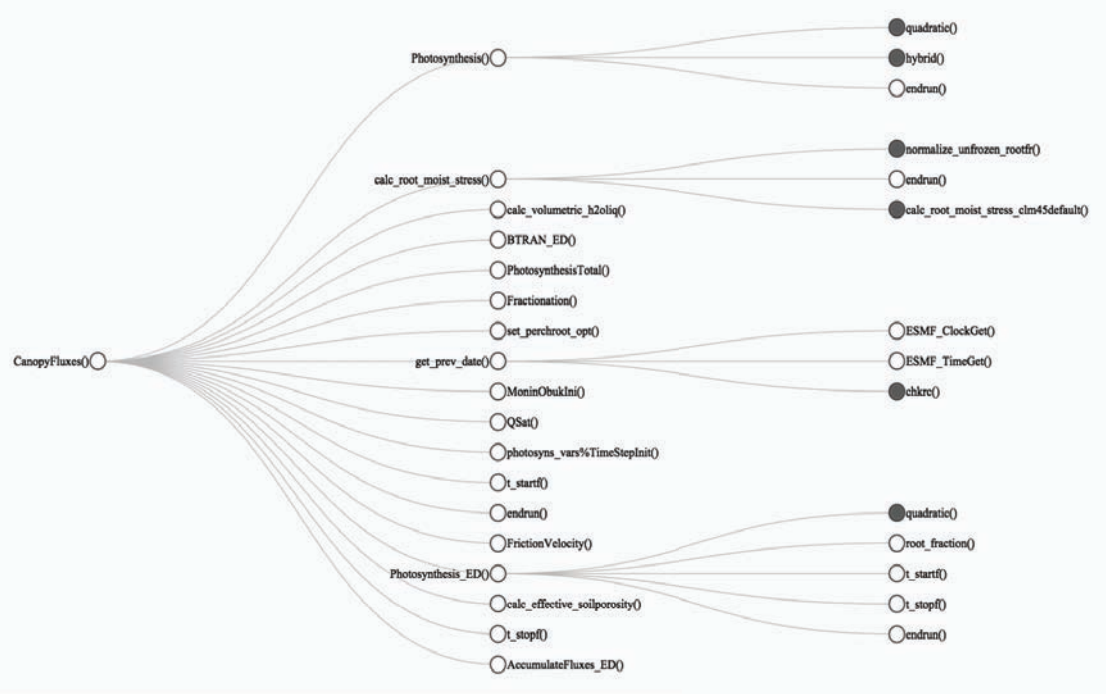

Figure 6: Call-tree of unit *CanopyFluxes*

4 Results

In this experiment, we try to track variables inside the scientific code while the scientific code is running to do single and multipoint checks. Even though it is easy for testers to track variables, it is

difficult for module builders to interact with certain modules through debuggers. As shown below, we use the unit test platform and one unit test module to generate a data bulk declaration and insert all these declarations into the scientific code to prepare for data sending. Then we use another data analysis application to request and analyze the data (see Figure 7).

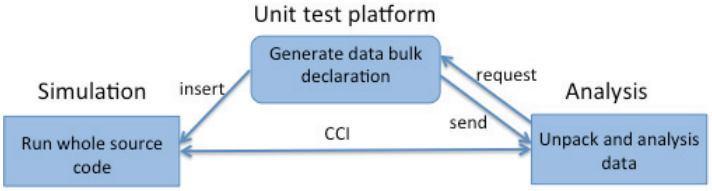

Figure 7: Experiment setup

Based on the former description about the CCI software, the scientific code (simulation side) and the analysis code (analysis part) can send each other messages and share large amounts of data with each other. So our experiment has two goals: single point check, which monitors all input and output variables in one certain function and multipoint check, which tracks one specific variable among different modules based on the scientific code's data flow. All data resulted from the simulation side can be managed in two ways: analyzed directly while in memory or saved on a disk for further data analysis. In this paper, we chose the second method to analyze data.

4.1 Single point check

We chose the *CNAllocation* module, designed to allocate key chemical elements (such as carbon [C], nitrogen [N], and most recently phosphorus [P]) in the terrestrial ecosystems, as our point to simulate. In this module, there are 160 variables: 65 in the read only group, 38 in the write only group, and the rest in the modify group. The ACME has 32 plant functional type (PFT) levels, which are intended to capture the biogeophysical and biogeochemical differences between broad categories of plants in terms of their functional characteristics. In this case, we set PFT = 10 to track *deadstemc_patch* in the carbon state data structure. The simulation code simulated from 1/1/2000 to 1/1/2001 and recorded data every 30 minutes; therefore, there are 17,520 total time slots. Figure 8 shows how the *deadstemc_patch* changes during 1 year through the *CNAllocation* function.

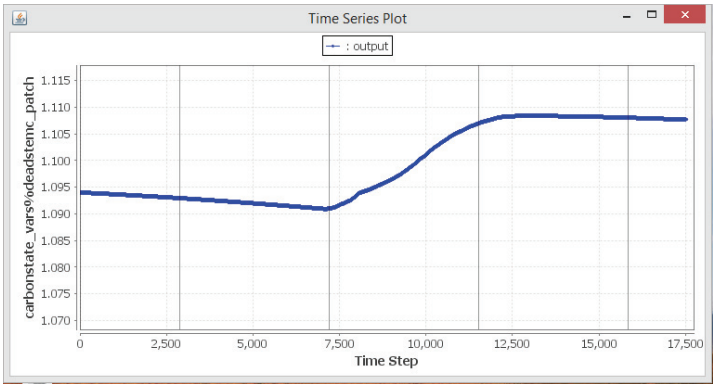

Figure 8: Status of the variable *deadstemc_patch* in the *CNAllocation* function

4.2 Multipoint check

Because there are many functions/subroutines and variables incorporated within this scientific system, it is important for model developers to track and record some key variables through the entire scientific procedure over temporal or spatial domains of interest. For example, *leafc_patch*, a variable that stores the leaf on ecosystem patch, is read by six functions (e.g., function determines harvest mortality routine for coupled carbon-nitrogen, function determines the flux of C and N from displayed pools to litter, function calculates fire area, function determines gap-phase mortality routine for coupled carbon-nitrogen, function determines the flux of C and N from displayed pools to litter, and allocation function) and can be modified by six functions (e.g., *CStateUpdate1*, *CStateUpdate3*, *CStateUpdate2*, *CStateUpdate2h*, *CNPrecisionControl*, *dyn_cnbal_patch*). Once a user decides to track those key variables, the unit test platform can automatically generate the data declaration and insert them on the scientific code side. After the simulation is finished, the data analysis side receives the extracted data and stores the data for further research. Figure 9 shows the status of variable *leafc_patch* for 2.5 days in the 11th PFT level. The colors blue, red, green, and purple represent *CStateUpdate1*, *CStateUpdate2*, *CStateUpdate2h*, and *CStateUpdate3* functions, respectively. The drop during the function *CStateUpdate2* means leaf storage is less and less in September.

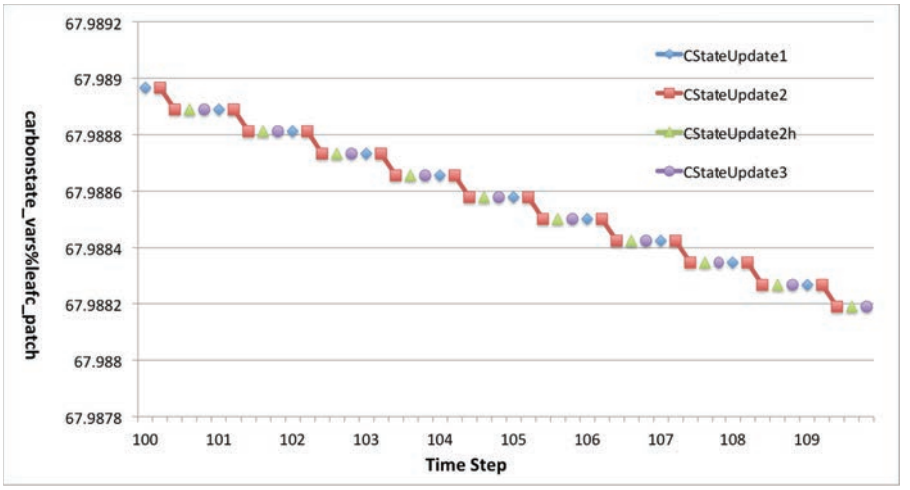

Figure 9: Status of variable *leafc_patch* in four functions

5 Conclusion

Testing is a significant software development process. However, many scientific codes have become much more complicated than before, which means there are extra needs to check that the changes positively impact dependent modules and to verify the system constraints. In this paper, we presented a compiler-based analyzer to generate unit data flow for separating modules and have developed an automatic methodology and prototype platform to facilitate scientific verification of individual functions within complex scientific codes, so that the science module builders are able to track variables conveniently in one module or track variables' changes among different modules. We used two experiments' results of monitoring variables to demonstrate the easy tracking capability of this platform. We believe the methodology and experience presented in the paper can be beneficial to

broad scientific communities with interests in In Situ model verification. In the future, we will apply a parallel method to the In Situ data analysis procedure and design a user-friendly interface.

Acknowledgements

This research was funded by the US Department of Energy, Office of Science, Biological and Environmental Research, Advanced Scientific Computing Research (ASCR), and an Oak Ridge National Laboratory (ORNL) Laboratory Director's Research and Development project (#7409). This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at ORNL, which is managed by UT-Battelle LLC for the Department of Energy.

References

[1] M. Pezze and M. Young, *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, 2007.

[2] L. C. Briand and A. Wolf, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering*, 2007, pp. 85–103.

[3] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," *SIGSOFT Software Engineering Notes*, vol. 27, no. 4, 2002, pp. 123–133.

[4] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on operating systems design and implementation*, 2008, pp. 209–224.

[5] P. Godefroid, M. Y. Levin, and D. Molnar, "Whitebox fuzzing for security testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2008, pp. 20:20–20:27.

[6] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra, "Guided test generation for web applications," in *Proceedings of the International Conference on Software Engineering*, 2013, pp. 162–171.

[7] D. Babic´, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2011, pp. 12–22.

[8] D. Wang, Y. Xu, P. Thornton, A. King, L. Gu, C. Steed, J. Schuchart, "A Functional Testing Platform for the Community Land Model," *Environmental Modeling and Software*, vol. 55, 2014, pp. 25–31. DOI: 10.1016/j.envsoft.2014.01.015.

[9] D.Wang, W. Wu, T. Janjusic, Y. Xu, C. Iversen, P. Thornton, M. Krassovisk, "Scientific Functional Testing Platform for Environmental Models: An Application to Community Land Model," International Workshop on Software Engineering for High Performance Computing in Science, 37th International Conference on Software Engineering, May 16–24, 2015, Florence, Italy. DOI: 10.1109/SE4HPCS.2015.10.

[10] M. Feathers, "Working Effectively with Legacy Code," 2012. http://www.objectmentor.com/resources/articles/WorkingEffectivelyWithLegacyCode.pdf.

[11] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca and R. Minnich, "The Common Communication Interface (CCI)" in the 19th IEEE Symposium on High Performance Interconnects (HOTI), Santa Clara, Calif., August 23–25, 2011

[12] Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee. Synergistic challenges in data-intensive science and exascale computing – DOE ASCAC data subcommittee report. Technical report, DOE Office of Science, March 2013.

[13] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron. From mesh generation to scientific visual-ization: An end-to-end approach to parallel supercomputing. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[14] A. S. Szalay, G. C. Bell, H. H. Huang, A. Terzis, and A. White. Low-power amdahl-balanced blades for data intensive computing. *SIGOPS Oper. Syst. Rev.*, 44(1):71–75, Mar. 2010.

[15] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11, April 2012.

[16] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson. Simplified parallel domain traversal. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 10:1–10:11, New York, NY, USA, 2011. ACM.

[17] A. Tikhonova, C. Correa, and K.-L. Ma. Visualization by proxy: A novel framework for deferred interaction with volume data. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6):1551–1559, 2010.

[18] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 424–434, Piscataway, NJ, USA, 2014. IEEE Press.

[19] J. Woodring, M. Petersen, A. Schmeißer, J. Patchett, J. Ahrens, and H. Hagen. In situ eddy analysis in a high-resolution ocean climate model. To appear in SciVis2015 and in IEEE Transactions on Visualization and Computer Graphics (Jan, 2016), 2015.