Facilitating Test-Driven Development via Domain-Specific Languages in Computational Science Software Engineering

Sven Gundlach (sgu@informatik.uni-kiel.de)
Kiel University
Reiner Jung
Kiel University
Wilhelm Hasselbring
Kiel University

Research Article

Keywords: Test-Driven Development, Domain-Specific Language, Research Software Engineering

Posted Date: December 20th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3753364/v1

Additional Declarations: No competing interests reported.

Facilitating Test-Driven Development via Domain-Specific Languages in Computational Science Software Engineering

Sven Gundlach[1*], Reiner Jung[1] and Wilhelm Hasselbring[1]

[1*]Software Engineering Group, Department of Computer Science, Kiel University, Christian-Albrechts-Platz 4, Kiel, 24118, Germany.

*Corresponding author(s). E-mail(s): sgu@informatik.uni-kiel.de;
Contributing authors: reiner.jung@stk.landsh.de;
hasselbring@email.uni-kiel.de;

Abstract

Research software plays a critical role in computational science. Therefore, the software should be of high quality. Producing such software involves efforts not only in initial development, but also during maintenance. One concept for improving the quality of software and reducing maintenance efforts is Test-Driven Development (TDD), which involves implementing tests, e.g., unit tests, as a specification for development. The effects are high test coverage and better overall software quality, but also additional effort in implementing tests. By separation of concerns, such additional efforts can be facilitated through the use of tailored notations such as domain-specific languages (DSLs). DSLs provide different levels of abstraction, for example, to separate the definition of tests from other concerns in software development. Not only does this incentivize research software engineers (RSEs) to write tests, but it also helps reduce the time required to write tests.

The DSL TDD-DSL addresses this to facilitate the implementation of TDD using the Fortran Unit Testing Framework (UTF) pFUnit 4.0. TDD-DSL is focused on unit test definition and source code integration. It is implemented in Python for system under tests (SUTs) written in Fortran 90 or later. To reduce the effort of defining tests, TDD-DSL separates the test definition from the implementation and setup. Especially the latter can cause additional effort in practice, which can be mitigated by using code generators provided with the DSL tooling.

The DSL allows describing test conditions as assertions and test cases as collections of assertions. TDD-DSL provides content assist within code editors for the underlying UTF, reducing the effort required in its deployment. It also supports

parsing the SUT to expand the content assist with source code information and high-level error messaging for the test description. The content assist includes publicly available variables, types, functions and routines. TDD-DSL further supports the definition of unit values such as SI units to help Scientific Modelers check unit values in the software at runtime. For this purpose, the DSL tooling enables the generation of code templates for new implementations in the source code according to the defined tests.
To simplify extending the DSL and improve overall maintainability, the DSL tooling employs the language server protocol (LSP) and also public available tools such as the ANTLR4 grammar. Using LSP also enables a multi-editor support to integrate the DSL tooling into different editors and IDEs.
In this paper we report on the design and implementation of the DSL and its tooling as well as the evaluation with Scientific Modelers and RSEs.

Keywords: Test-Driven Development, Domain-Specific Language, Research Software Engineering

1 Introduction

In the domain of computational science, research relies heavily on research software such as Earth System Models (ESMs). ESMs are used to simulate Earth's climate and to understand its effects and aspects, such as on oceans, agriculture, and habitats. Specific aspects of the Earth system, such as the ocean, are simulated using different models. Further, models are composed of specialized sub-models. E.g., ocean models are composed of sub-models for transport, sediments, and biological processes (Alexander & Easterbrook, 2015; Collins et al., 2005; Warner, Perlin, & Skyllingstad, 2008).

Successful ESMs as well as their sub-models are long-living, complex software systems. They often start as small projects and grow over years in size and community (Aumont, Ethé, Tagliabue, Bopp, & Gehlen, 2015; Weaver et al., 2001). Due to continuous development and changes made over the years, these models face the typical issues of long-lived software, similar to software from other domains. These issues include, among others, code complexity, legacy code, and architecture erosion (Goltz et al., 2015). As a result, these issues increase maintenance costs for development and application, reducing the time and resources available for research.

Given that the development of these models is dominated by researchers that usually come from other areas of expertise rather than software engineering, the development of such models can lack best practices and therefore limit software quality (T.L. Clune & Rood, 2011). Such practices also require additional resources and effort, even when research software engineers (RSEs) are aware of them and have the expertise to apply them. Due to the lack of incentives for best practices, such as in publications, such practices are often not implemented or are only implemented with minimal effort. Also, research software such as ESMs are often seen as a by-product of scientific research rather than a distinct contribution by itself (Storer, 2017). However, since researchers rely on these software models, software quality is an important goal

in many research facilities (Haupt, Schlauch, & Meinel, 2018). High-quality software requires not only efforts in development but also maintenance.

One approach for improving the software quality and reducing maintenance is Test-Driven Development (TDD), which uses tests as predetermined software requirements (Beck, 2003). Unlike the traditional Test-Last Development (TLD) approach, in which source code is first developed and then tested for correctness, in TDD tests are written first, and subsequently functionality is implemented and directly testable. The result is better software quality (Nanthaamornphong & Carver, 2017; Staegemann et al., 2022). TDD, however, requires additional effort at the beginning compared to TLD for writing tests at the expense of the application code (Bissi, Neto, & Emer, 2016). This additional effort only gains an advantage in later development, maintenance and debugging. TDD also requires a restructuring of the development process, as there are major challenges associated with such concepts, such as the technical complexity of applying TDD and the lack of suitable software tools to create tests (Nanthaamornphong & Carver, 2017; Staegemann et al., 2022).

An approach to facilitate such software engineering methods, in this case unit testing, is separation of concerns through the use of tailor-made notations such as domain-specific languages (DSLs). DSLs provide different levels of abstraction, for example, to separate the definition of tests from other concerns of a software system, such as specific requirements for a test framework. Not only does this incentivize RSEs to write tests, but it also helps to reduce the time required to implement tests.

This paper presents the DSL TDD-DSL and its tooling to facilitate the implementation of TDD in research software using the unit test framework for Fortran pFUnit 4.0 (T. Clune, 2019; pFUnit, 2023; Rilee & Clune, 2014). TDD-DSL focuses on test definition with source code integration. TDD-DSL and its tooling are implemented in Python for system under tests (SUTs) written in Fortran 90 or later.

Software testing generally uses test conditions, which are aspects of a component or system that can be verified (IEEE, 2013) Test conditions are verified using test cases, which consist of input values, preconditions, expected results, and postconditions. Test cases can subsequently be further organized as test procedures or test suites to test software. In short, a test procedure is a sequence of test cases to be run consecutively, including the setup of common preconditions (IEEE, 2013). In contrast, a test suite specifies only the test cases that need to be executed without necessarily being cohesive (ISTQB, 2017).

The TDD-DSL approach to facilitating test execution is:

- To separate test definitions from a specific Unit Testing Framework (UTF).
- Provide a unified specification to define all test-related information.
- Reduce the effort required for the test implementation with content assist and code generation and code templates.

The test cases should be organized according to the associated modules. However, this is not further enforced to allow RSEs to organize test cases in custom order. To reduce the effort of defining tests, TDD-DSL provides an abstract test definition that separates the test definition from the implementation details and setup. Test conditions are combined with test data and expected results in a single, consistent specification.

Thus making the description independent of framework-specific requirements, and thus simplifying the implementation and setup of tests. TDD-DSL tooling supports parsing the SUT to provide content assist and high-level error messaging for the test description. The generation of UTF-specific files and source code templates further reduces the effort required for the test implementation.

Figure 1 depicts the application of TDD-DSL included in a TDD process. First, a test is implemented with TDD-DSL. In a second step, the TDD-DSL tooling generates the required files, including those required for the UTF and a template for the specific test implementation in the source code. The automatically generated templates facilitate the implementation of the specific test and allow RSE to focus on the actual test implementation without having to implement UTF-specific files, such as CMake scripts. Subsequently, the test can be executed by the UTF, e.g., via generated driver scripts. Executing the test separately reduces the risk of the DSL and its tools becoming obsolete, as the generated tests can still be used. If the test fails, the SUT can be refactored again. In case the test is successful, however, a new test can be implemented or the SUT can be released.

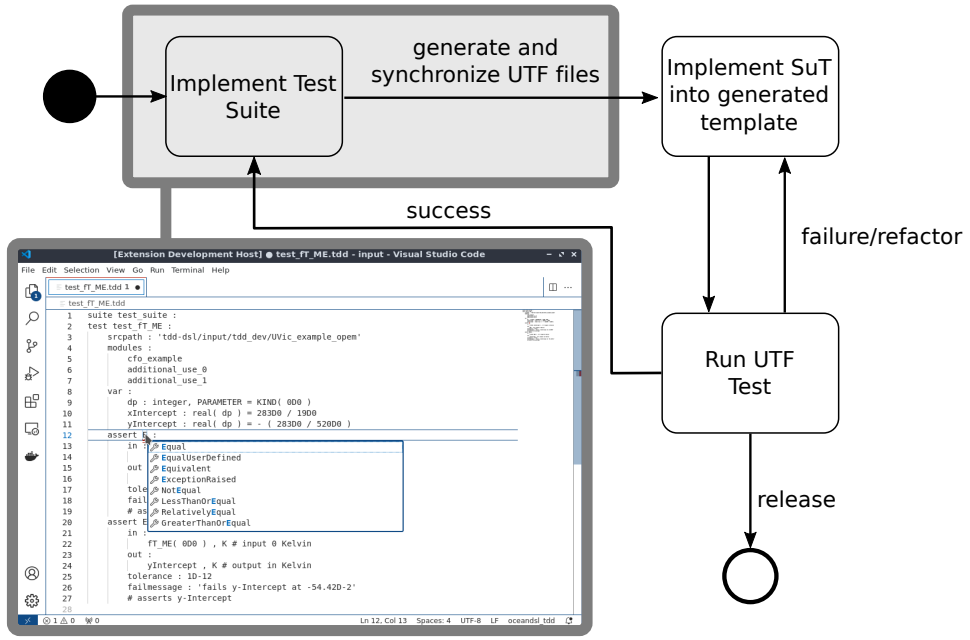

Fig. 1 TDD process detailing the DSL. A test is written before implementing the software (SUT) that complies with the test. The DSL tooling provides content assist for implementing a test, synchronizes the required UTF files, and optionally generates templates for the SUT. The test is executed by the UTF and can subsequently be refactored, further developed or released.

The design and implementation of TDD-DSL face various challenges and requirements:

1. The application of a DSL in general introduces additional tools that need to be maintained.
2. The DSL should be compatible and easy to integrate with existing development and work processes used by RSEs.
3. As research software is extended and modified over time, the DSL should be modifiable.

The first and third challenges are addressed through the use of maintained and publicly available generic tools such as the ANTLR4 parser generator (Parr, 2014) to implement TDD-DSL. The TDD-DSL is further implemented in Python, which is an established programming language in the domain. The use of maintained and generic tools, as well as programming languages established in the domain, simplifies the modification of the DSL and its general application in the domain. To reduce the complexity of TDD-DSL, SUTs are parsed with the Fortran parser fxtran (Marguinaud, 2023), and testing is performed by the underlying UTF, e.g., the maintained UTF pFUnit 4.0. For the second challenge, the tooling of TDD-DSL supports the language server protocol (LSP) (Microsoft, 2016) for editor and IDE integration. This allows to apply TDD-DSL in multiple editors used in established development processes.

Although software models such as ESMs in some cases use older Fortran versions, such as Fortran 77, most development takes place in Fortran 90, which uses modern programming paradigms such as modules. The use of TDD-DSL to integrate test support for cross-project tools further provides incentives for the use of such programming paradigms. Still, modules imply that TDD-DSL is limited to SUTs employing at least Fortran 90. The TDD-DSL tooling also supports only preprocessed code, which implies that source code with preprocessor declarations must be preprocessed first.

Given that the DSL is intended to be both intuitive to use and maintainable, the source code is made available in a publicly accessible repository on GitHub. In addition, the DSL is also provided as a Python package, which is available via PyPI.

TDD-DSL was evaluated on the ESM UVic (Weaver et al., 2001). The evaluation was done by extending UVic with tests to experimentally analyze the feasibility of the DSL and empirically analyze its usability. For feasibility, generated test setups were validated, and test participants (domain experts) were asked to describe and implement test cases. Subsequently, semi-structured interviews were conducted with these domain experts to analyze the usability. The results were analyzed using the thematic analysis (TA) approach (Braun & Clarke, 2006).

Before describing TDD-DSL, related work is discussed in Section 2. Findings from the UVic case study and a previous domain analysis (Jung, Gundlach, & Hasselbring, 2022a) to identify language and tooling requirements are presented in Section 3. The design of the DSL and its tooling are presented in Section 4. The implementation choices are explained in Section 5. Section 6 reports on the empirical evaluation. Finally, a summary and outlook are given in Section 7.

2 Related Work

In the following, related testing frameworks and DSLs are discussed.

2.1 DSLs for Unit Testing

In contrast to computational science, in the software development industry, DSLs are already used for software testing on a daily basis (Micallef & Colombo, 2015). While the specific implementations are not directly designed for computational science, mainstream technologies such as Cucumber or Gherkin (Wynne, Hellesoy, & Tooke, 2017) can be used to define test suites and test cases to validate high-level aspects (Mischke, Schaffert, Schneider, & Weinert, 2022).

- Gherkin is a DSL used in the software development industry that focuses on specifying behavior in a human-readable Given-When-Then notation consisting of a series of steps (Wynne et al., 2017). These steps are either a given precondition for the test case, a condition when a test case occurs, or a postcondition that can then be verified after the test case is executed. Gherkin is supported by Cucumber, a Behavior-Driven Development (BDD) testing framework for writing automated tests based on the desired behavior of the software. Cucumber can be used to define test scenarios and be integrated with programming languages such as Java, Ruby and JavaScript to execute the tests. But Cucumber is not integrated with Fortran.
- Micallef and Colombo (2015) designed multiple DSLs based on Gherkin for five case studies found in the software development industry. In addition to Gherkin, the Xtext language framework (Bettini, 2016; Efftinge et al., 2012) was used to provide an editor integration based on ANTLR3, as opposed to TDD-DSL based on ANTLR4. To address the different requirements of the case studies, a layered, hierarchical approach, as in SPRAT, was used to separate the concerns of domain experts, Scientific modelers, and RSEs. In order to capture key test concepts, such as test suites similar to TDD-DSL, a generic root DSL was implemented. Derived from the root DSL, more specific DSLs were implemented using various testing frameworks, such as Sikuli (Yeh, Chang, & Miller, 2009).

Gherkin specifies the behavior of a program using BDD, which allows complementing test coverage by validating high-level aspects of a program, such as the execution and results of complete test workflows. In computational science, such workflows often involve a process whose behavior is the actual research question or part of it, and consequently, its results are unknown in the beginning. Therefore, DSLs such as Gherkin can be used for validation in conjunction with low-level functional tests such as unit tests and integration tests. Still, Python is often used in computer science, which is not supported by Xtext. Also, ANTLR3 requires more complicated grammar constructs since it is based on LL(*), while ANTLR4 supports, e.g., left recursion using Adaptive LL(*) (Parr, Harwell, & Fisher, 2014).

2.2 UTFs in computational science

While without specific use of DSLs, UTFs for testing software like Cucumber also exist in computational science.

- Yao, Wang, Riccuito, Yuan, and Fang (2019); Yao, Wang, Sun, and Zhong (2017) designed a UTF to generate unit tests. It is intended to make understanding undocumented source code easier and be suitable for large-scale software. The UTF uses

Vampir (Nagel, Arnold, Weber, Hoppe, & Solchenbach, 1996) to visualize the program behavior. Further, it supports global variables that are analyzed based on the data flow of the code through instrumented subroutines. The subroutines are divided into three categories based on the methods used to access the global variables. To be practical for large-scale scientific codes, it applies MPI-based parallelization, I/O behavior optimization, and profiling results to increase performance.

- The functional unit testing (FUT) platform (Wang et al., 2014) was developed for scientific function module generation and verification with captured data from reference simulations. The purpose of the platform is to allow direct comparison between simulation results and real-world measurements at the functional element level. It consists of four main components. A unit test driver, the SUT, data capture and verification modules. The SUT is instrumented using code injection to extract I/O parameters from the contained subroutines. Next, like in TDD-DSL, a unit test driver is used to initialize the parameters, execute the SUT independently of the model containing it, and store the output. The unit test driver allows the FUT to generate test cases and execute specific test cases of interest.
- SPEL is a software tool for porting the Exascale Energy Earth System Model (E3SM) Land Model to GPUs using OpenACC (Schwartz, Wang, Yuan, & Thornton, 2022). It is a collection of Python scripts designed to automatically generate GPU-ready test modules for ELM functions and break the ELM into stand-alone unit test programs. SPEL uses the FUT concept and applies compiler directives to port the scientific code. Since it specifically targets porting the E3SM ELM to GPUs, SPEL modifies the ELM main function and uses deepcopy for data handling. In addition, SPEL removes modules that cannot run on GPUs or are undesirable. It also uses static code analysis on subroutines to make modifications required for GPU compilation.

These frameworks provide comprehensive test environments and partially address the increased time required to develop tests and the difficulty of writing tests. Their focus is on performance and comparing inputs to real-world measurements. They also require a TLD approach to extract information from the SUT. Separation of concerns can complement these efforts by reducing the effort involved in applying unit tests.

2.3 General UTFs for Fortran

While UTFs are not commonly used in Fortran like JUnit for Java or PyTest for Python, multiple UTFs have been developed for Fortran (T.L. Clune & Rood, 2011). Some frameworks are no longer actively maintained or have been superseded. Still, active frameworks are the parallel Fortran Unit testing framework (pFUnit), FLIBS, Objexx Fortran ToolKit (ObjexxFTK), Veggies and Zofu.

- pFUnit is a hosted on GitHub and enables JUnit-like testing with MPI extensions (pFUnit, 2023). It is being developed by NASA and NGC TASC and is based on the Fortran unit testing framework (FUnit) (FUnit, 2001), which is not actively maintained anymore. pFUnit uses Fortran 2003, object-oriented design techniques and a TDD methodology. Since pFUnit is intended for TDD within the research

domain, it supports single/double precision testing with optional tolerance. Single/-double precision is particularly useful for tests to check results for subtle numerical problems such as inaccuracies. pFUnit uses preprocessor input files written in standard Fortran, and like FUT, it uses a test driver to run the SUTs independently and build them into executable test suites.

- FLIBS is a collection of Fortran modules and is integrated, e.g., in the Ftnunit framework (Arjen Markus, 2010). It relies on parts of FUnit and provides routines for checking assertions as well as routines for running the tests. Unit testing modules in FLIBS cannot be detected automatically and have to be executed by general run routines.
- ObjexxFTK is a commercial UTF consisting of a set of Fortran packages (Objexx Engineering, 2023). It is developed by Objexx Engineering and supports Fortran up through Fortran 2018. It further provides array utility predicates as well as string functions. An executable test driver is generated via Python scripts by collecting existing test cases.
- Veggies is a unit testing framework that is being developed in parallel with Garden (Richardson, 2022a, 2022b). Both are replacements for the unit testing framework Vegetables. Veggies is hosted on GitLab and requires the Fortran Package Manager (fpm) and other prerequisites managed via fpm. It is written using functional programming principles and can be used to test parallel code. It also provides a readable test/code specification. Furthermore, Veggies and Garden provide input generators for simple types that can be overwritten by the user. Both Veggies and Garden are intended for BDD-style test specifications of Fortran 90 code. To generate test suites and executable drivers, the authors provide the cart tool and use a Given-When-Then notation like Gherkin. Alternatively, users can also write their own Fortran function using the Given-When-Then notation to define test suites.
- Zofu is hosted on GitHub and was designed in the context of the Waiwera geothermal flow simulator (Croucher et al., 2019). It is intended to replace the FORTRAN Unit Test Framework (FRUIT) (Chen & David, 2003) for Waiwera. Like pFUnit, Zofu is written using object-oriented functions from Fortran 2003 and also uses the Fortran 90 module concept. Zofu supports MPI and tools such as CTest or Meson to execute test drivers. Depending on the tool, tests can be executed individually per module or as a whole test suite. The execution of a whole test suite has the disadvantage that the run of the test suite stops at the first failed test case.

While these UTFs can be used to test research software written in Fortran, they do not focus on separation of concerns as can be done with DSLs. A DSL approach such as TDD-DSL can therefore make use of these UTFs.

2.4 DSLs in computational science

DSLs are already used to improve software engineering for computational science. These DSLs are primarily focused on parallelization, refactoring, and numerical modeling. Respectively, they do not have built-in features or explicit support for TDD practices. Still, they provide abstractions for scientific code, thus enabling software engineering practices with less effort.

- PSyclone (Parallelisation System code generator) is an internal DSL embedded into Fortran as a host language based on in-place code transformation specifically designed for high-performance computing and parallelism in scientific codes (Adams et al., 2019; Sivalingam, Ashworth, Porter, & Ford, 2018). It is developed by the UK Science and Technology Facilities Council's Hartree Centre in collaboration with the Met Office UK and the Australian Bureau of Meteorology. PSyclone targets Fortran and enables the automatic generation of parallel code using domain-specific abstractions. The code transformation allows to both write code in PSyclone and optimize existing code that uses appropriate code structures using PSyclone as a compiler infrastructure. PSyclone provides a set of extensions to Fortran syntax that are replaced by standard Fortran code via transformation. The code transformation is mainly implemented in Python.
- SPRAT is a hierarchical DSL designed for end-to-end ecosystem modeling (Johansson & Hasselbring, 2017; Johanson, Oschlies, Hasselbring, & Worm, 2017). It uses a multi-layer scientific modeling approach with four layers covering deployment, simulation parameterization, ecosystem modeling, and FEM-PDE solver setup. Like TDD-DSL, SPRAT provides features such as content assistance for programmers and high-level error messages. It also supports advanced checks for unit types and correct value conversion. The layers represent separate concerns supported by three specific DSLs. As the first DSL, the Sprat PDE solver DSL is an internal DSL embedded into C++ designed to facilitate the implementation of numerical algorithms with a focus on finite-element methods for partial differential equations (PDEs). The second DSL is the Sprat Ecosystem DSL, an external DSL based on Xtext (Bettini, 2016; Efftinge et al., 2012), which is based on the Eclipse Modeling Framework (Steinberg, Budinsky, Paternostro, & Merks, 2009). Finally, the third is the Sprat Deployment DSL, which describes how a simulation can be mapped to a specific execution environment.
- DUNE (Distributed and Unified Numerics Environment) is a modular framework for solving PDEs (Bastian et al., 2021). While it is not a DSL itself, it provides a set of C++ libraries and interfaces that facilitate the development of PDE solvers in a structured and reusable manner. The core module is maintained by the DUNE developers and builds an agreed-upon design. Higher functionalities, such as PDE assembler modules, are built by independent groups. Thus, modules exist, providing the equal functionality but following different design ideas.
- FEniCS is a collection of open-source software components and a DSL for solving PDEs using the finite element method (Alnæs et al., 2015). It is written in C++/Python and is accessed via the interface DOLFIN. FEniCS provides the Unified Form Language (UFL) (Langtangen, 2012) an internal DSL embedded into Python to define physical equations in terms of finite element variational forms. It also includes the internal FEniCS Form Compiler to translate these high-level mathematical descriptions expressed in UFL into a Unified Form-assembly Code (UFC). UFC is a C++ interface for low-level C++ functions evaluating finite element variational forms for PDEs. Thereby, FEniCS can automatically generate efficient PDE solvers.

Although these DSLs support separation of concerns, they are not designed for testing, particularly not for TDD.

3 Motivation and Rationale for TDD-DSL

TDD-DSL is designed to be applicable to research software such as ESMs that use Fortran 90. Therefore, prior to the design and implementation phases of TDD-DSL, requirements for the DSL and its tooling were identified through semi-structured interviews with domain experts from the computational science domain coupled with a TA approach. A detailed description can be found in Jung, Gundlach, and Hasselbring (2022b).

This section provides a brief introduction to the relevant TA findings based on the interviews regarding the development and work processes, testing techniques and methods, environments regarding platform, operating system, system access and programming tools, as well as applied programming languages and UTFs that were identified in the domain.

The results are based on several interviews conducted with scientists and technical staff, i.e., RSEs or scientific programmers involved in ocean model development from the GEOMAR Helmholtz Centre for Ocean Research Kiel, the German Climate Computing Center (DKRZ) and the Max Planck Institute for Meteorology (MPI).

3.1 Software Development

Actors in the computational science domain have varying programming skills and scientific backgrounds, from which their development processes derive. This includes best practices and techniques of software development, such as version/variant management, programming environments and test concepts. Based on the tasks performed by the actors, actors can be grouped into roles, of which they can perform multiple. In our interview analysis, we identified seven roles and also identified several processes and sub-processes (Jung et al., 2022a).

This paper focuses on two processes in which the DSL should be integrated. These are modeling, including developing and testing, and maintenance, as shown in Figure 2 and Figure 3. Directly involved roles are Scientific Modeler, RSE and Gatekeeper. The Scientific Modeler works with the model to conduct research. RSEs are developers who work on the model. This means, for example, that new features are either developed by RSEs for their own research or implemented by RSEs for others. Gatekeepers are RSEs with special tasks for upstream models.

3.2 Traditional Test Methods and Techniques

In the interviews, interviewees stated that features are tested manually after being implemented by individual RSEs in a TLD approach. It was also stated that tests are performed manually in simplified test setups or with scripts, whereby the model results are compared manually with the results calculated by Scientific Modelers. Some interviewees stated that automated techniques are sometimes used together with scripts. The Scientific Modelers presumed to have a good comprehension of what the code

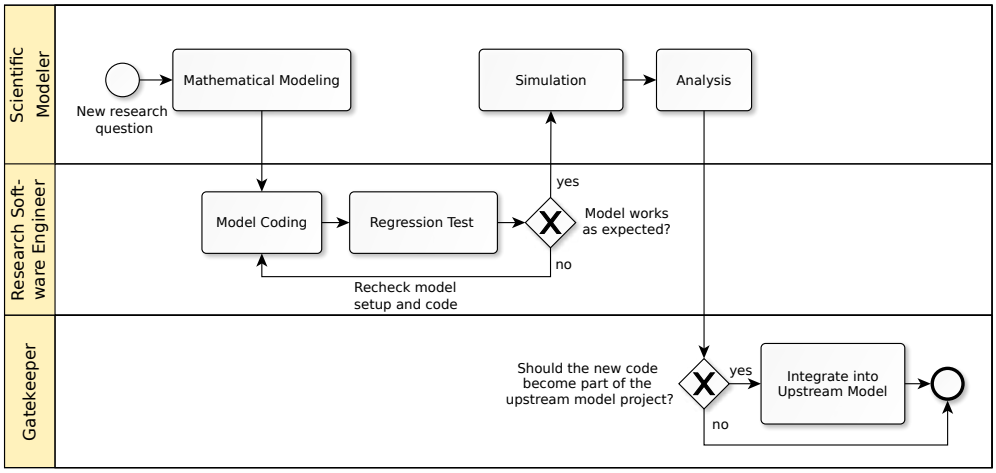

Fig. 2 Extract of the ocean modeling and simulation process, depicting the modeling process with the three roles Scientific Modeler, RSE and Gatekeeper. The process is specified with the standardized Business Process Modeling Notation (BPMN) (Chinosi & Trombetta, 2012)

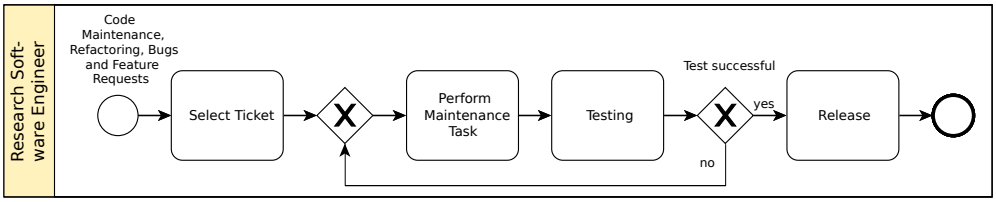

Fig. 3 Extract of the ocean maintenance process depicting the maintenance process done by RSEs. The process is specified with the standardized Business Process Modeling Notation (BPMN) (Chinosi & Trombetta, 2012)

should produce. However, it was pointed out that secondary processes can nevertheless prove to be more important than assumed, become more significant, or that initial assumptions prove to be incorrect. All of which may affect the program behavior.

Based on the interviews, the code is usually published without test cases. Tests are often either discarded or stored locally. It was remarked that in some cases, tests for physical units are implemented in the source code and checked at runtime. As models are in most cases not modularized in such a way that the components can be run independently, testing can only be carried out on the target platform. Interviewees conducted simple tests without long models locally.

In the case of migration, interviewees stated that plausibility checks were carried out on the target platform on which the model runs. In plausibility checks, selective findings such as negative concentrations or mass balance checks are reviewed in the global model. Interviewees specified that they carry out tests in private environments but do not publish the tests. However, it was highlighted that new models such as NEMO 4.0 (Madec et al., 2023) and MITgcm (Artale et al., 2010) come with publicly available test cases, such as preconfigured model variants that serve as regression tests for the model. Still, test procedures and unit tests are reportedly not established in the

domain. Therefore, integration tests for global models are carried out manually and represent a considerable time effort for RSEs, which, according to the interviewees, can take several years.

For integration with upstream models, it was reported that regression tests are performed to compare the results with previous model runs, using specific test scenarios and a scientifically guided interpretation of the results. No specific quality standards were applied by the interviewees. Tasks for upstream models involve Gatekeepers and are used, i.e., for global releases.

Quality measures such as coding styles or standards are applied in some models. Styles include, for example, schemes for naming variables or using or not using typing and procedures. Interviewees did not use tool support to enforce such measures.

3.2.1 Intended Test Methods and Techniques

In addition to the applied methods and techniques, it was reported that further methods and techniques are planned but have not yet been implemented. Automatic tests using, e.g., GitLab-CI to test specific units are considered and should avoid errors such as semantic errors at an early stage, but were not realized. Continuous integration was also intended to make inline configuration automatically available in the model. Consideration was also given to including testing procedures in student training to raise awareness of testing issues. Further, attempts to implement serialization frameworks that serialize the inputs and outputs of a model to enable more independent model components were reported. Independent components should thereby be able to be tested individually in order to verify their correct behavior.

3.3 Environments and Platform

Development and runtime platforms were diverse among the interviewees and depended on the community model used and personal preferences. Scientific Modelers and RSEs reported to use workstations PCs or laptops with various operating systems, computing clusters and HPC infrastructure. Most hardware platforms run some sort of Linux distribution, and some workstations run macOS. According to the interviewees, the separation between development and runtime platforms is unclear, as development platforms can also be runtime platforms.

Larger models are reportedly often developed, tested and maintained directly on the runtime platform. Interviewees accessed remote environments such as clusters via SSH scripts. Vi and derivatives such as Vim were most commonly used by interviewees, especially in use cases where they switched between local and remote development. The interviewees pointed out that outdated tools and documents are a constant problem, as relevant sources are not always available.

3.4 Programming Languages

Models that interviewees were familiar with are mainly written in Fortran 77 and Fortran 90. C and C++ code is also used for specific tasks, particularly for infrastructure. To some extent, the interviewees employed Matlab and Octave for smaller models, prototyping and analyses.

Interviewees generally highlighted that R and especially Python are used for post-processing and analysis. Interviewees also pointed out that Python is increasingly being used in other roles, such as a host language for embedded DSLs. Other roles for Python are libraries written via internal DSLs or in other languages such as C and C++, which are accessed via Python. In addition, the interviewees reported that YAML formatting has become established.

Models that use the Fortran 90 module concept follow a modularization scheme using the Fortran *module* statement. Others, such as MITgcm, have their own similar package concept but do not use the Fortran *module*. Models such as the UVic ESCM (Weaver et al., 2001) started with older versions such as Fortran 77 and are grouped into multiple source directories related to submodels, general functions and library support. Models are further structured using *subroutines* or *functions*.

For the build and setup of models, several configuration and parameterization schemes were identified. Often, these schemes are combined into specialized build setups using parameters, input files, and preprocessor directives. For example, UVic uses a two-step setup scheme where features and parameters are configured at compile-time or runtime using two input files, `mk.in` and `control.in`. Further, UVic uses conditional compilation using C preprocessor directives such as `#ifdef` and `#if defined` and combines source files into one compilation unit using directives such as `#include`.

3.5 Unit Testing Frameworks

Interviewees did not used a UTF for Fortran. Still, for other programming languages, the domain experts used UTFs, such as PyTest for Python, which uses an XUnit-like testing approach.

4 Design of TDD-DSL

The following section explains the design decisions for the TDD-DSL (Section 4.1), followed by a presentation of the abstract metamodel (Section 4.2) and the abstract megamodel (Section 4.3) as well as the design decisions for the code generator (Section 4.4).

The design decisions for TDD-DSL derive from the identified requirements presented in Section 3 and aim to solve the challenge described in Section 1. The requirements include established programming languages such as Python and Fortran, tool support to reduce additional workload, support of physical units for parameters, support of established editors, and usability on test and target platforms. Using the DSL to facilitate test development, the development process should be consistent with well-known testing frameworks of other applied programming languages such as PyTest for Python and make it simpler for RSEs to get started.

4.1 General Design Decisions

The syntax should match other languages with which Scientific Modelers and RSEs are familiar, especially Python and Fortran. Using familiar languages helps to incorporate the DSL into existing development and work processes (cf. Challenge 2 in Section 1).

To avoid nested braces, which are not a common concept in Fortran or Python, the DSL follows syntaxes present in established configuration formats, namely YAML and to some extent Python. However, in contrast to Python, indentation is not part of the syntax of TDD-DSL, as grammars formatted with indentation are difficult to maintain. The decision to use YAML and Python is based on early evaluation feedback from RSEs.

Content assist helps RSEs significantly with implementation. For this purpose, the DSL tooling combines and provides information from the SUT and the test specification, such as existing variables or routines. Providing assistance such as accessible variables is a priority, as complex SUTs are difficult for RSEs to keep track of.

To reduce the maintenance effort of the DSL parts of the tooling are generated automatically from the grammar via code generators such as parser generators. Further, the use of a generated visitor allows for simple modification of the DSL tooling application code (cf. Challenge 3 in Section 1).

The support of LSP allows to integrate the DSL in different editors and IDEs that support LSP. This support makes the application of the DSL more flexible and enables to use the DSL in existing workflows (cf. Challenge 2 in Section 1). As LSP is under active development, the DSL tooling employs a generic language server to reduce the maintenance impact of the ongoing LSP development. High-level error messages can indicate inconsistencies in the test definition and the test implementation. By deriving and combining information from both sources, such warnings are generated and forwarded via LSP. To support physical units for better documentation of the intended applications, TDD-DSL allows to express physical units as expressions of SI units (Newell & Tiesinga, 2019; "Système International d'Unités", 1960) for parameter values. Unit information is also incorporated into the code generation for runtime checks.

The test implementation provides information about the requirements of the SUT such as declarations and code references. To reduce the task of matching test specifications and SUT implementations, the DSL tooling supports the generation of templates for code structures such as functions, routines and modules. These templates can be further implemented by the RSE. In addition, setup files for the underlying UTF are generated automatically to reduce the effort for the RSE to run tests. This allows RSEs to focus on the tests themselves without having to maintain additional files (cf. Challenge 1 in Section 1).

To facilitate the compartmentalization of SUTs and improve software quality and maintainability, the DSL tooling uses the Fortran 90 module concept. As such, the Fortran *module* statement, which is supported by at least Fortran 90, is required by the DSL tooling. This restricts the use of the DSL to Fortran versions starting from Fortran 90 but encourages the use of modern code versions for new SUTs.

4.2 Abstract Metamodel

To combine the test case information described by the TDD-DSL and source code information from a SUT, a generic metamodel for supported symbols is used. The metamodel describes the abstract syntax of the TDD-DSL that is used to model which information belongs to a particular symbol. The purpose of the abstract syntax presented here is to compactly represent what is *supported* in the DSL. The metamodel is represented in the standardized Unified Modeling Notation UML (Object Management Group, 2017).

The metamodel (Figure 4) uses a Symbol as a root from which subsequent supported symbols are derived. The two directly derived classes are TypedSymbol and ScopedSymbol.

Typed symbols represent symbols that contain values such as units or types. As such, variable symbols may contain physical units, specific types, or both. TDD-DSL supports primitive types such as integer, real, float, double and string. However, Fortran types such as real may refer to floats or doubles depending on the system environment.

Scoped symbols describe all symbols that contain other symbols. These are test case symbols, modules of the SUTs and routines, e.g., Fortran functions or subroutines. Symbol tables that contain all symbols for a test suite are also scoped symbols.

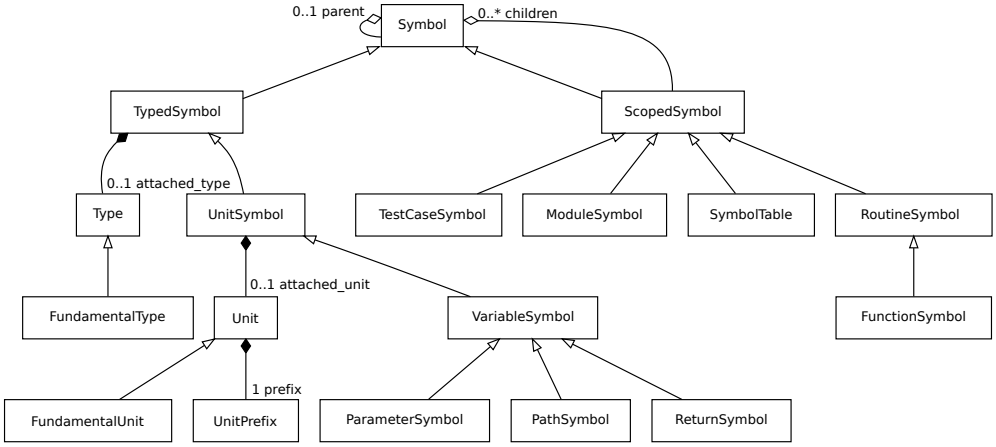

Fig. 4 Metamodel representing *supported* symbols. Symbols are organized in a parent-child tree structure. Scoped symbols describe all symbols containing other symbols as child symbols. Typed symbols represent symbols that contain values such as units or types and are grouped under a parent symbol. As such, variable symbols may contain physical units, specific types, or both.

4.3 Megamodel for Code Generation

Figure 5 shows a so-called *megamodel* that visualizes the relationships and transformations for a test setup. The notation follows Favre, Lämmel, and Varanovich (2012) and Jung (2016). Boxes represent models, arrows with closed tips and solid lines illustrate the direction of data flow. Asterisks represent a multiplicity of files. The test

suite specification, which corresponds to the TDD-DSL grammar, is transformed via a parser into a concrete-syntax tree (CST) and subsequently into the symbol metamodel. In parallel, the Fortran SUT is parsed into an AST, which is also further transformed into the symbol metamodel by the DSL tooling. Finally, a code generation produces the test artifact, which can be scripts, Fortran modules, or other test files.

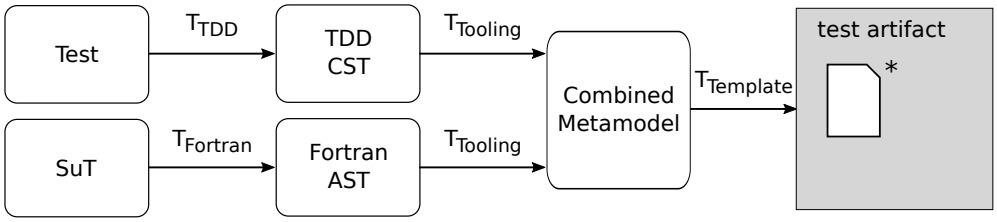

Fig. 5 The megamodel of the abstract process depicts the model transformation from the test specification and the SUT to a test artifact. The test specification is transformed into a CST. In parallel, the Fortran SUT is parsed into an AST. Both models are transformed into a combined metamodel that is used for the final generation of the test artifacts. Boxes represent models, arrows with solid lines and closed tips illustrate the direction of data flow. Asterisks represent multiplicity.

4.4 Code Generators

The design of the code generators is based on the visitor pattern, transforming rules with template engines into Fortran templates that match the requirements of the UTF. Using a template Engine allows to define plain text templates with macros and variables to generate UTF-specific files. The transformation further uses the symbol metamodel to combine the information from the test case specification with the information from the SUT. The combination allows to supplement the generated templates to match both the test case and the SUT. The plain text implementation of the templates also allows RSEs to define and adapt templates without changing the DSL implementation. This decouples the test specification from the file generator.

The code generation is also designed to allow code to be merged into existing SUTs and update UTF files accordingly. The usage of the DSL tooling with existing SUTs allows to refactor SUTs and also use the DSL to write tests for existing code bases.

5 Implementation of TDD-DSL

The following chapter presents the implementation infrastructure of TDD-DSL (Section 5.1), the concrete grammar (Section 5.2), code parsing (Section 5.3), and the implementation of the code generators (Section 5.4). The implementation specifies the abstract megamodel with concrete models, as shown in Figure 6.

The design of the TDD-DSL allows to modify the code generator to support different UTFs. Still, the current version of TDD-DSL tooling implements pFUnit as the underlying UTF, since pFUnit is intended for TDD in scientific applications and thus supports existing development processes well, which complements the approach

of the DSL. pFUnit also allows to define separate tests in preprocessor input files and run the tests independently with scripts, which allows to separate the UTF from the source code.

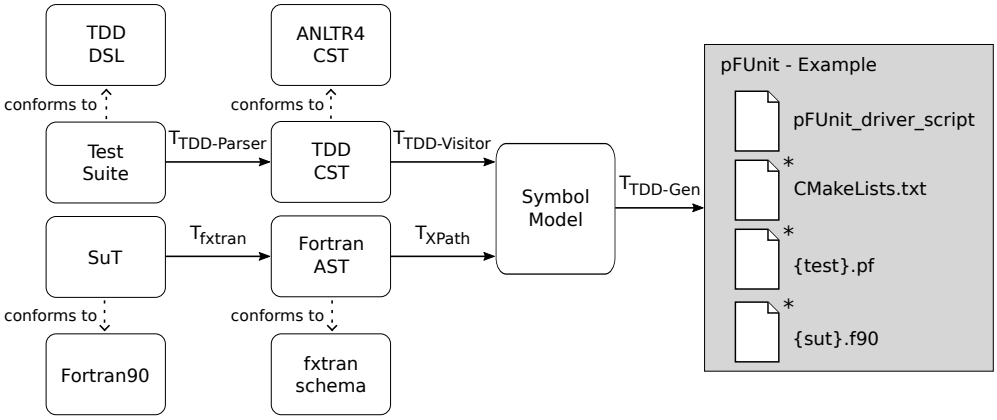

Fig. 6 The megamodel of the concrete process depicts the model transformation from the test suite and the SUT to a concrete pFUnit example. The example consists of a driver script, CMake setup files, a preprocessor and the adapted SUT. The test suite specification, corresponding to the TDD-DSL grammar, is transformed into the CST and subsequently into the metamodel (Figure 4). In parallel, the SUT, which conforms to Fortran 90, is parsed into an AST, which is further transformed into the metamodel. Finally, the DSL tooling generates the pFUnit files. Boxes represent models, arrows with solid lines and closed tips illustrate the direction of data flow, arrows with open tips and dashed lines indicate a model conformity with a metamodel. Asterisks represent multiplicity.

5.1 Infrastructure Implementation

For the implementation infrastructure, the DSL tooling uses ANTLR4 (Parr, 2014), Jinja2 (Ronacher, 2008) and pygls (Open Law Library, 2019). ANTLR4 is a parser generator that allows to implement DSLs based on grammar specifications using rules and lexer tokens. It features an adaptive LL(*) parser, also known as ALL(*), direct link recursion, and decoupling of application code via listener and visitor patterns. The ANTLR4 parser provides a parse tree, also known as CST, and also supports Python code generation. TDD-DSL uses ANTLR4 to generate the parser and lexer for the TDD-DSL grammar. The visitor pattern is also used to decouple the grammar from the code generation.

pygls is a generic language server "skeleton" that implements LSP (Microsoft, 2016). The feature support of pygls enables simple adaption and extension of the LSP support for the DSL. pygls supports asynchronous listening for incoming messages and corresponding responses to registered commands and functions. The concrete behavior of the DSL tooling is specified via Python functions with decorators from pygls.

Jinja2 is a general-purpose templating language that supports Python. The engine allows to define templates with macros and variables to generate input files for the

UTF in text format. The flexible design makes it simple for RSEs to write and modify generator templates. Using Jinja2 allows to write generators without requiring a specific template DSL that would otherwise be needed to support different generators.

Further, the TDD-DSL tooling uses the Fortran parser fxtran (Marguinaud, 2023). fxtran constructs ASTs in the form of XML documents that can be used with Document Object or XPath Models.

The tooling was chosen to reduce the maintenance effort for the DSL (cf. Challenge 1 in Section 1) and also to reduce the effort required to extend it if necessary (cf. Challenge 3 in Section 1). Supporting LSP reduces the effort required to integrate the DSL with existing IDEs and editors (cf. Challenge 2 in Section 1).

While the use of the DSL and its tooling introduces new dependencies and thus risk, the risk is mitigated as the generated code can be used by the UTF without using the DSL. Also, the generated artifacts are still readable even if the DSL becomes deprecated.

5.2 Concrete Grammar

To illustrate the grammar, Listing 1 shows a textual representation of an example from UVic. The test suite consists of one test case that tests the x-intercept and y-intercept of a simple linear temperature function. The function should be implemented in the module `cfo_example`. The parameters of the function are defined as expressions in variables and are used to assert the intercepts of the function. The variable definition and typing are written in Fortran, while the formatting follows YAML syntax. In addition, the test case specifies that existing test setup files are overwritten. Fortran files are retained.

In general, the grammar allows each test file to describe a test suite of multiple test cases with a unique name. Each test case is specified with a name that is used to identify the test in the setup files. In addition to the name, each test case contains test variables, assertions and information about where the SUT is located. Modules and variables are marked as groups with `modules` and `var` as shown in lines five and seven. Each test case can contain multiple modules or variables in `modules` or `var`, respectively. Assertions start with the keyword `assert` (cf. lines 11 and 19).

System file paths are used to identify existing and new modules. If the module does not exist, a new module is created. If the module exists, it is searched for the routine, which is optionally added to the module if the routine is not found. Existing routines are not modified.

Test variables contain variable declarations and optional expressions. Expressions can be functions, references or values. Assertions contain input and output parameters, which can also be functions, references or values. Optional assertions can have physical units and user-defined comments.

5.3 Code Parsing

The megamodel shown in Figure 6 depicts the general steps involved in parsing. Initially, all relevant source files are parsed and transformed into in-memory representations. After this, the relevant modules, variables and routines are collected in

```
suite test_suite :
test test_fT_ME :
    overwrite : pf , cmake
    srcpath : 'tdd-dsl/input/tdd_dev/UVic_example_opem'
    modules :
        cfo_example
    var :
        dp : integer, PARAMETER = KIND( 0D0 )
        xIntercept : real( dp ) = 283D0 / 19D0
        yIntercept : real( dp ) = - ( 283D0 / 520D0 )
    assert Equal :
        in :
            fT_ME( xIntercept ) , K # custom comment when
                input is 14 ,89473684 Kelvin
        out :
            0D0 , K # output should equal 0 Kelvin
        tolerance : 1D-12
        failmessage : 'fails x-Intercept at 14.89D0'
         # comment for pFUnit file : asserts x - Intercept
    assert Equal :
        in :
            fT_ME( 0D0 ) , K # custom comment when input is
                0 Kelvin
        out :
            yIntercept , K # output should equal -0
                ,5442307692 Kelvin
        tolerance : 1D-12
        failmessage : 'fails y-Intercept at -54.42D-2'
         # more comment for pFUnit file : asserts y -
            Intercept
```

Listing 1 DSL file that generates a pFUnit example for a linear function that tests intercepts.

the metamodel (Figure 4) and provided to the RSEs as needed. To ensure the correct symbol structure, each symbol is added under a specific scope, from which it can be retrieved accordingly.

For the code of SUTs, the scope is determined by scope-changing elements. These are elements for which a scope can be determined. Such elements are collected dynamically and stored on a scope stack to ensure the correct order. As test cases can only access public elements, a scope filter is used to restrict inaccessible parts of the SUT and reduce the elements to the scopes included by the test case.

For the test case, scoping is provided via the ANTLR4 visitor pattern. Each element in a test case is determined via the CST structure. This structure includes so-called *contexts* that describe the environment of each element in the CST. By using the *contexts* as scoping identifiers, the metamodel can be updated using the CST.

5.4 Code Generation

As shown in the DSL megamodel in Figure 6, the code generation is based on the metamodel (Figure 4). Initially, to determine the generated test cases, the CST is processed via the ANTLR4 visitor pattern in order to leverage the CST structure for code generation. After this, for each test case found in the test suite, a TestCaseSymbol is retrieved from the metamodel. The symbol is used to populate variables required by the corresponding Jinja2 template implemented for the underlying UTF. The correct template is identified by the unique rule index in the CST, which is mapped to the rule name supplied by the ANTLR4 parser.

The generated code templates are finally included into existing modules or written directly as new modules if the modules do not yet exist. In this way, a specific generator is implemented for each required file type.

6 Evaluation

The evaluation is based on structured interviews with five domain experts using replications of existing routines from UVic.

The DSL was found to be practical and useful to ease the test implementation process, especially the generation of UTF-specific files, which was considered helpful (cf. Q6 in Figure 7).

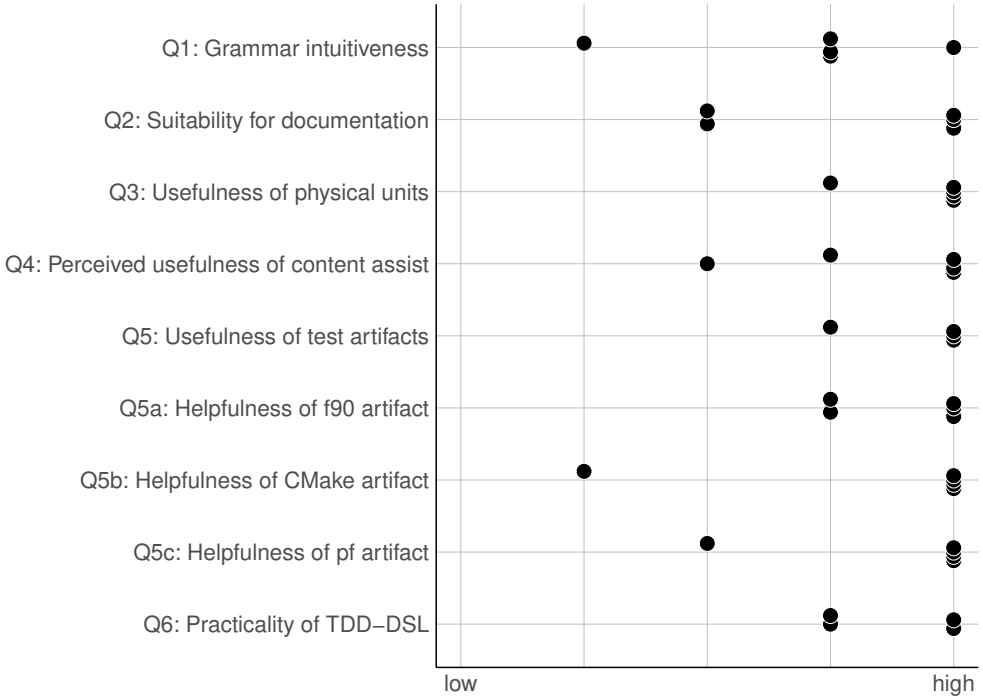

Fig. 7 Qualitative feedback from domain experts on their perception of TDD-DSL.

6.1 Demographics

The domain experts considered themselves all Scientific Modelers and partially RSEs and Gatekeepers, and were familiar with UVic. The editors used are Emacs, Vim and VS Code, all of which support LSP. Besides UVic the interviewees were familiar with other ESMs such as NEMO. Interviewed experts considering themselves RSEs were familiar with TDD, while others were not familiar with TDD. In some cases, efforts were already being made to incorporate TDD into the ongoing workflow. Still, specific UTFs such as pFUnit were not known or not used.

As a common practice, the lack of software documentation is known. Nevertheless, the documentation is considered necessary for the correct application. Therefore, RSEs document the developed code to the best of their abilities.

Build tools such as CMake are known by some domain experts, but not used due to the initial effort required and the complexity of such tools. Instead, make scripts are used to handle initial tasks and are expanded to the point where script maintenance is the main test-writing task. Therefore, the tool support in test generation was considered by the domain experts to be a significant help and was expected to increase productivity in code development. Such code generation was found to be a strong incentive for using tools such as TDD-DSL.

Open licensing and the availability of the source code of the DSL and its tooling were indicated as required, and GitHub was considered suitable.

6.2 Grammar Evaluation

The grammar was found to be intuitive and self-explanatory (cf. Q1 in Figure 7). Still, a more strict syntax that domain experts are used to, such as YAML lists, would be preferred, even if it would make the definition of the test more verbose. In addition, more descriptive names for flags, e.g., for overwriting, were noted. Furthermore, the option to add README notes on top was considered helpful.

Inconsistencies, such as different input sequences between the generated UTF code and the grammar, were found to be negligible.

6.3 Feature Evaluation

Variable definitions were found to be important for defining not only specific test inputs, but also general constants such as zero degrees Celsius in Kelvin.

The tests were perceived as suitable supplements to documentation (cf. Q2 in Figure 7) and, in some cases, as more helpful than the documentation alone. The option to easily implement stand-alone tests for new functions was also seen as useful.

Physical unit definitions were considered useful for testing in terms of *documentation as code* and as useful examples for documentation in general (cf. Q3 in Figure 7). Still, they were found to be unnecessary for unit tests in which RSEs know the intent of the SUT. RSEs would intend to test physical units within integration tests if SUTs required physical units. With regard to integration, some experts considered simple comments without code integration to be sufficient. Some SUTs do not require physical

units. Issues were noted with the complexity of definitions of physical units, e.g., vector units such as directed velocities as opposed to scalars used in elementary routines. The use of arrays for multiple units was found to be intuitive.

The content support offered by the DSL tooling was considered helpful and as expected (cf. Q4 in Figure 7). Expected contents were existing modules, functions and variables. For variables, parameters were considered sufficient instead of mutable public variables from SUTs. Issues were noted with uninitialized variables, since in Fortran only constants need to be initialized.

The generation of routine templates that can be used by RSEs when implementing newly defined code was considered useful and important to ease the effort required for test implementation (cf. Q5 in Figure 7). In particular, the automatic generation and handling of UTF-specific files such as CMake, which are separate from the source code, was considered especially important (cf. Q5a,b,c in Figure 7). Source references included in the templates were considered helpful to assist RSEs in linking tests and SUTs.

The ability to use the DSL tooling as a Command-Line Interface (CLI)-based tool, only generating UTF-specific files from existing TDD-DSL test specifications, was found to be particularly useful for long-term application and employment of the DSL.

6.4 Suggested Improvements

To improve unit integration, the automatic conversion of physical units or the integration of unit libraries was highlighted as a possible useful function. Furthermore, additional support for generic test functions for assignable arrays that test indexed fields was raised as potentially helpful.

To expose private functions for testing, experts suggested integrating wrappers into the source code that indicate a test use for private functions, and considered the inclusion of such routines in models to be suitable. Support for custom assertions by including Fortran test implementations in the test definition was also considered helpful.

Automatically resolvable links were noted as potential features.

Finally, the experts suggested supporting teardown and launch routines in the DSL to increase the reusability of tests.

6.5 Feedback on Limitations

Limitations such as Fortran 90 were noted and seen as a drawback for models using older versions such as Fortran 77, which is used in production code even though it is known to be complicated to work with. Nevertheless, it was not considered the task of the DSL to support such versions, as development should move to more modern approaches to software development. This was seen not only to support the use of tools such as the DSL but also to improve general code comprehension.

The use of module structures for new developments was also considered necessary, independent of the DSL. Mocking support was not viewed as necessary because the number of use cases was considered too small. Adding CMake as a prerequisite for test execution was considered reasonable for models as long as model execution remained unaffected.

6.6 Threats to Validity

Construct Validity

The questions on which the interviews are based can overlook relevant aspects. Semi-structured interviews were therefore used to adapt the interview questions when new aspects emerged.

Internal Validity

The experience level of the participants or different knowledge among the participants could influence the evaluation. To mitigate this threat, participants were divided according to their knowledge of Fortran programming, testing, UVic, and DSLs. All participants had prior experience in ESMs and computational science. It can be argued that the number of participants in the experiment was low. However, the participants are technical staff and scientists working in the domain of computational sciences on real research projects with research software and are directly or indirectly involved in the development of ESMs. Therefore, they are able to conduct the experiment and provide informed feedback on the application of the DSL and its integration into existing work processes.

External Validity

Simplicity of the example used in the experiment, as only two configurations of simple functions and modules were used. To mitigate this threat, the tasks and examples were taken from a real ESM.

Conclusion Validity

Conclusions from individual statements may not be reliable. To avoid this, the results were analyzed with TA to identify concepts or patterns.

7 Conclusions and Future Work

This paper presents the domain-specific language (DSL) TDD-DSL to facilitate Test-Driven Development (TDD) in computational science software engineering. The DSL and its tooling have been evaluated by domain experts from computational science who are familiar with the UVic ESCM. The key concepts of the DSL tooling, e.g., template generation, content assist, code integration, and unit support with code integration, are presented.

TDD-DSL separates the test definition from the source code and combines all test-relevant information into a unified specification, from which its tooling provides code generation for the required files.

The paper explains the general architecture of the code generation and system under test (SUT) integration and illustrates this with a case study based on the Earth System Model (ESM) UVic. The evaluation included domain experts working as Scientific Modelers, research software engineers (RSEs) and Gatekeepers. The discussion of the major requirements of the DSL itself is based on a previous domain analysis

through semi-structured interviews with domain experts and a thematic analysis (TA) of the interview results, as documented in Jung et al. (2022a, 2022b).

In its current form, the level of abstraction of TDD-DSL was found to be helpful by the domain experts, and the DSL was found to be intuitive, suitable, useful, and practical to reduce test implementation efforts. The file generation of Unit Testing Framework (UTF)-specific files separately from the source code from a single file was mentioned as particularly useful for employing unfamiliar or unknown UTFs. Provided that the user understands the underlying concept of TDD, domain experts found it to be intuitive and self-explanatory. Experts also found it helpful in the process of familiarization with new models and features. The content assist and physical unit integration were considered helpful by the domain experts to ease the implementation process and avoid mistakes.

A key concern for scientists is ease of use, limiting dependencies, and integration into their workflows. TDD-DSL and its tooling address these issues through an intuitive DSL, support for the language server protocol (LSP) without additional dependencies other than Python, and human-readable artifacts. The integration of the ANTLR4 parser generator and the Jinja2 templating language allows to target changes in the underlying UTF and simple modification of the DSL. LSP support allows the integration of the DSL into a variety of editors used among RSEs such as Vim and Emacs. The wide support is particularly helpful in a scientific context where contributors use different editors and environments. The support of the LSP via the generic pygls language server allows simple adaptation of the DSL tooling to changes in LSP. The use of Fortran Modules restricts the use to at least Fortran 90, but also encourages the application of modern Fortran versions.

As this is an ongoing research project, we aim to further extend and improve TDD-DSL in close contact with Scientific Modelers and RSEs. We initially developed the DSL for a representative subset of UVic with pFUnit and are currently evaluating it with further ESMs. We defined specific setups for UVic based on the model used at GEOMAR, as available domain experts were familiar with the ESM and employed it in real research projects. Thus, supporting the replication of our experiments.

However, the current support for physical units raised a major point of comment from the domain experts, as the physical unit definitions were seen as useful but complex, especially when converting and therefore should be used in conjunction with unit libraries. A concern was raised that the LSP support of the DSL tooling could become deprecated, and therefore the use of code generation via the Command-Line Interface (CLI) option was highlighted as important.

It was also noted that features for generic test functions, the use of automatically resolvable links, and the support of teardown and launch routines would increase the usability of the DSL. We will address these insights in the next revision of the DSL, including a thorough investigation of more generic approaches.

Our current efforts also focus on improving the content assist, code generation and semantics of TDD-DSL and its tooling. In particular, the current parameter-override semantic follows a straightforward solution where existing files can be overwritten or updated. Since this allows the accidental definition of routines, we will evaluate several language solutions in the future that prevent this. For instance, the possibility to

tag not-further implemented routine templates as obsolete. We will further evaluate TDD-DSL versions with domain experts from our partner institutions.

Supplementary information. The implementation of TDD-DSL is available as open source software under https://github.com/cau-se/python-oceandsls.

The complete results of the previously conducted interviews (Section 3) are available in the form of an online graph at https://oceandsl.uni-kiel.de/graph.

Acknowledgments. Funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. HA 2038/8-1 – 425916241.

Statements and Declarations

Funding

Funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. HA 2038/8-1 – 425916241.

Competing Interests

The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Ethics approval

Not applicable.

Consent to participate

All participants in the evaluation consented to participate in the evaluations.

Consent for publication

All participants in the evaluation consented to the publication of the evaluations.

Availability of data and materials

The complete results of the previously conducted interviews (Section 3) are available in the form of an online graph at https://oceandsl.uni-kiel.de/graph.

Code availability

The implementation of TDD-DSL is available as open source software under https://github.com/cau-se/python-oceandsls.

Authors' contributions

Sven Gundlach wrote the main manuscript text, created and finalized figures 1-7 and conducted the final interviews. Reiner Jung prepared figures 2 and 3. Wilhelm Hasselbring contributed to the conception and design. All authors reviewed the manuscript.

References

Adams, S.V., Ford, R.W., Hambley, M., Hobson, J.M., Kavčič, I., Maynard, C.M., ... Wong, R. (2019, October). LFRic: Meeting the challenges of scalability and performance portability in weather and climate models. *Journal of Parallel and Distributed Computing*, *132*, 383–396, https://doi.org/10.1016/j.jpdc.2019.02.007

Alexander, K., & Easterbrook, S.M. (2015, April). The software architecture of climate models: a graphical comparison of CMIP5 and EMICAR5 configurations. *Geoscientific Model Development*, *8*(4), 1221–1232, https://doi.org/10.5194/gmd-8-1221-2015

Alnæs, M., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., ... Wells, G.N. (2015). The FEniCS project version 1.5. *Archive of Numerical Software*, *3*, 9–23, https://doi.org/10.11588/ANS.2015.100.20553

Arjen Markus, M.B. (2010). *FLIBS - A collection of Fortran modules.* https://flibs.sourceforge.net.

Artale, V., Calmanti, S., Carillo, A., Dell'Aquila, A., Herrmann, M., Pisacane, G., ... Rauscher, S. (2010, October). An atmosphere–ocean regional climate model for the mediterranean area: assessment of a present climate simulation. *Climate Dynamics*, *35*(5), 721–740, https://doi.org/10.1007/s00382-009-0691-8

Aumont, O., Ethé, C., Tagliabue, A., Bopp, L., Gehlen, M. (2015, August). PISCES-v2: an ocean biogeochemical model for carbon and ecosystem studies. *Geoscientific Model Development*, *8*(8), 2465–2513, https://doi.org/10.5194/gmd-8-2465-2015

Bastian, P., Blatt, M., Dedner, A., Dreier, N.-A., Engwer, C., Fritze, R., ... Sander, O. (2021, January). The dune framework: Basic concepts and recent developments. *Computers & Mathematics with Applications*, *81*, 75–112, https://doi.org/10.1016/j.camwa.2020.06.007

Beck, K. (2003). *Test-driven development: by example.* Addison-Wesley Professional.

Bettini, L. (2016). *Implementing domain specific languages with xtext and xtend - second edition* (2nd ed.). Packt Publishing Ltd.

Bissi, W., Neto, A.G.S.S., Emer, M.C.F.P. (2016, June). The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, *74*, 45–54, https://doi.org/10.1016/j.infsof.2016.02.004

Braun, V., & Clarke, V. (2006, January). Using thematic analysis in psychology. *Qualitative Research in Psychology*, *3*(2), 77–101, https://doi.org/10.1191/1478088706qp063oa

Chen, A.H., & David, P. (2003). *Fortran unit test framework (FRUIT).* https://sourceforge.net/projects/fortranxunit.

Chinosi, M., & Trombetta, A. (2012, January). BPMN: An introduction to the standard. *Computer Standards & Interfaces*, *34*(1), 124–134, https://doi.org/10.1016/j.csi.2011.06.002

Clune, T. (2019, April). *Testing fortran software with pFUnit* (Tech. Rep.).

Clune, T.L., & Rood, R.B. (2011, November). Software testing and verification in climate model development. *IEEE Software*, *28*(6), 49–55, https://doi.org/10.1109/ms.2011.117

Collins, N., Theurich, G., DeLuca, C., Suarez, M., Trayanov, A., Balaji, V., ... da Silva, A. (2005, August). Design and implementation of components in the earth system modeling framework. *The International Journal of High Performance Computing Applications*, *19*(3), 341–350, https://doi.org/10.1177/1094342005056120

Croucher, A., O'Sullivan, M.J., O'Sullivan, J., Yeh, A., Burnell, J., Kissling, W. (2019). An update on the waiwera geothermal flow simulator: development and applications. *Proceedings 41st new zealand geothermal workshop* (Vol. 25, p. 27).

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M. (2012, September). Xbase. *ACM SIGPLAN Notices*, *48*(3), 112–121, https://doi.org/10.1145/2480361.2371419

Favre, J.-M., Lämmel, R., Varanovich, A. (2012). Modeling the linguistic architecture of software products. *Model driven engineering languages and systems* (pp. 151–167). Springer. https://doi.org/10.1007/978-3-642-33666-9_11

FUnit (2001). http://nasarb.rubyforge.org/funit.

Goltz, U., Reussner, R.H., Goedicke, M., Hasselbring, W., Märtin, L., Vogel-Heuser, B. (2015). Design for future: managed software evolution. *Computer Science - Research and Development*, *30*(3-4), 321–331, https://doi.org/10.1007/s00450-014-0273-9

Haupt, C., Schlauch, T., Meinel, M. (2018, June). The software engineering initiative of DLR. *Proceedings of the international workshop on software engineering for science.* ACM. https://doi.org/10.1145/3194747.3194753

IEEE (2013). Software and systems engineering Software testing Part 1: Concepts and definitions. (pp. 1–64). https://doi.org/10.1109/ieeestd.2013.6588537

ISTQB (2017, May). *ISTQB®/GTB Standard Glossary of Terms Used in Software Testing* (Tech. Rep.).

Johanson, A.N., & Hasselbring, W. (2017, August). Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. *Empirical Software Engineering*, *22*(4), 2206–2236, https://doi.org/10.1007/s10664-016-9483-z

Johanson, A.N., Oschlies, A., Hasselbring, W., Worm, B. (2017, April). SPRAT: A spatially-explicit marine ecosystem model based on population balance equations. *Ecological Modelling*, *349*, 11–25, https://doi.org/10.1016/j.ecolmodel.2017.01.020

Jung, R. (2016). *Generator-composition for aspect-oriented domain-specific languages* (Doctoral thesis/PhD). Faculty of Engineering, Kiel University.

Jung, R., Gundlach, S., Hasselbring, W. (2022a, April). Software development processes in ocean system modeling. *International Journal of Modeling, Simulation, and Scientific Computing*, *13*(02), , https://doi.org/10.1142/s1793962322300023

Jung, R., Gundlach, S., Hasselbring, W. (2022b, April). Thematic domain analysis for ocean modeling. *Environmental Modelling & Software*, *150*, 105323, https://doi.org/10.1016/j.envsoft.2022.105323

Langtangen, H.P. (2012). A FEniCS tutorial. *Automated solution of differential equations by the finite element method* (pp. 1–73). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-23099-8_1

Madec, G., Bell, M., Blaker, A., Bricaud, C., Bruciaferri, D., Castrillo, M., ... Wilson, C. (2023). NEMO Ocean Engine Reference Manual. https://doi.org/10.5281/ZENODO.1464816

Marguinaud, P. (2023). *fxtran.* https://github.com/pmarguinaud/fxtran.

Micallef, M., & Colombo, C. (2015, April). Lessons learnt from using DSLs for automated software testing. *2015 IEEE eighth international conference on software testing, verification and validation workshops (ICSTW).* IEEE. https://doi.org/10.1109/icstw.2015.7107472

Microsoft (2016). *Language Server Protocol.* https://microsoft.github.io/language-server-protocol.

Mischke, R., Schaffert, K., Schneider, D., Weinert, A. (2022). Automated and manual testing in the development of the research software RCE. *Computational science – ICCS 2022* (pp. 531–544). Springer International Publishing. https://doi.org/10.1007/978-3-031-08760-8_44

Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.-C., Solchenbach, K. (1996). *VAMPIR: Visualization and analysis of MPI resources.* https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-26639.

Nanthaamornphong, A., & Carver, J.C. (2017). Test-driven development in scientific software: a survey. *Software Quality Journal*, *25*(2), 343–372, https://doi.org/10.1007/s11219-015-9292-4

Newell, D.B., & Tiesinga, E. (2019). *The international system of units (SI):: 2019 edition.* https://doi.org/10.6028/nist.sp.330-2019

Object Management Group (2017, December). *OMG Unified Modeling Language – Version 2.5.1.* https://www.omg.org/spec/UML/2.5.1.

Objexx Engineering (2023). *ObjexxFTK: Objexx Fortran ToolKit.* https://objexx.com/ObjexxFTK.html.

Open Law Library (2019). *pygls, a pythonic generic language server.* https://pypi.org/project/pygls.

Parr, T. (2014). *The definitive ANTLR 4 reference* (2nd ed.). The Pragmatic Bookshelf.

Parr, T., Harwell, S., Fisher, K. (2014, October). Adaptive LL(*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, *49*(10), 579–598, https://doi.org/10.1145/2714064.2660202

pFUnit (2023). https://github.com/Goddard-Fortran-Ecosystem/pFUnit.

Richardson, B. (2022a). *Garden.* https://gitlab.com/everythingfunctional/garden.

Richardson, B. (2022b). *Veggies.* https://gitlab.com/everythingfunctional/veggies.

Rilee, M., & Clune, T. (2014, November). Towards test driven development for computational science with pFUnit. *2014 second international workshop on software engineering for high performance computing in computational science and engineering.* IEEE. https://doi.org/10.1109/se-hpccse.2014.5

Ronacher, A. (2008). *Jinja2 documentation.* https://jinja.palletsprojects.com.

Schwartz, P., Wang, D., Yuan, F., Thornton, P. (2022, November). SPEL: Software tool for Porting E3SM Land Model with OpenACC in a Function Unit Test Framework. *2022 workshop on accelerator programming using directives (WACCPD).* IEEE. https://doi.org/10.1109/waccpd56842.2022.00010

Sivalingam, K., Ashworth, M., Porter, A., Ford, R. (2018, September). PSyclone: a code generation and optimisation system for finite element and finite difference codes. *NCAR MultiCore*, *6*, ,

Staegemann, D., Volk, M., Perera, M., Haertel, C., Pohl, M., Daase, C., Turowski, K. (2022, July). A literature review on the challenges of applying test-driven development in software engineering. *Complex Systems Informatics and Modeling Quarterly*(31), 18–28, https://doi.org/10.7250/csimq.2022-31.02

Steinberg, D., Budinsky, F., Paternostro, M., Merks, E. (2009). *EMF: Eclipse Modeling Framework* (2nd ed.). Pearson Education.

Storer, T. (2017, August). Bridging the chasm. *ACM Computing Surveys*, *50*(4), 1–32, https://doi.org/10.1145/3084225

Système international d'unités. (1960). *Proceedings of the 11th CGPM* (p. 87). Bureau International des Poids et Mesures. https://doi.org/10.59161/cgpm1960res12e

Wang, D., Xu, Y., Thornton, P., King, A., Steed, C., Gu, L., Schuchart, J. (2014, May). A functional test platform for the community land model. *Environmental Modelling & Software*, *55*, 25–31, https://doi.org/10.1016/j.envsoft.2014.01.015

Warner, J.C., Perlin, N., Skyllingstad, E.D. (2008, October). Using the model coupling toolkit to couple earth system models. *Environmental Modelling & Software*,

23(10-11), 1240–1249, https://doi.org/10.1016/j.envsoft.2008.03.002

Weaver, A.J., Eby, M., Wiebe, E.C., Bitz, C.M., Duffy, P.B., Ewen, T.L., ... Yoshimori, M. (2001, December). The UVic earth system climate model: Model description, climatology, and applications to past, present and future climates. *Atmosphere-Ocean*, *39*(4), 361–428, https://doi.org/10.1080/07055900.2001.9649686

Wynne, M., Hellesoy, A., Tooke, S. (2017). *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf.

Yao, Z., Wang, D., Riccuito, D., Yuan, F., Fang, C. (2019). Parallel computing for module-based computational experiment. *Lecture notes in computer science* (pp. 377–388). Springer International Publishing. https://doi.org/10.1007/978-3-030-22741-8_27

Yao, Z., Wang, D., Sun, J., Zhong, D. (2017, December). A unit testing framework for scientific legacy code. *2017 international conference on computational science and computational intelligence (CSCI)*. IEEE. https://doi.org/10.1109/csci.2017.163

Yeh, T., Chang, T.-H., Miller, R.C. (2009, October). Sikuli. *Proceedings of the 22nd annual ACM symposium on user interface software and technology*. ACM. https://doi.org/10.1145/1622176.1622213