Agile Acceleration of LLVM Flang Support for Fortran 2018 Parallel Programming

Katherine Rasmussen, Damian Rouson,
Dan Bonachea, Brian Friesen, Hussain Kadhem
Lawrence Berkeley National Laboratory, USA
{krasmussen,rouson,dobonachea,bfriesen,hmk}@lbl.gov

Najé George
Computational Science Research Center
San Diego State University, USA
ngeorge8641@sdsu.edu

***Abstract*—The LLVM Flang compiler (”Flang”) is currently Fortran 95 compliant, and the frontend can parse Fortran 2018. However, Flang does not have a comprehensive 2018 test suite and does not fully implement the static semantics of the 2018 standard. We are investigating whether agile software development techniques, such as pair programming and test-driven development (TDD), can help Flang to rapidly progress to Fortran 2018 compliance. Because of the paramount importance of parallelism in high-performance computing, we are focusing on Fortran's parallel features, commonly denoted "Coarray Fortran". We are developing what we believe are the first comprehensive, open-source tests for the static semantics of Fortran 2018 parallel features, and contributing them to the LLVM project. A related effort involves writing runtime tests for parallel 2018 features and supporting those tests by developing a new parallel runtime library: the CoArray Fortran Framework of Efficient Interfaces to Network Environments (Caffeine).**

***Index Terms*—HPC, Exascale Computing, Compiler Testing**

I. Introduction

The LLVM Compiler Infrastructure project [1] provides open-source frontends for several languages, including C and C++. Because of the essential role that Fortran plays in high-performance computing, the U.S. Department of Energy (DOE) has led a multi-year effort, through the Exascale Computing Project (ECP), to develop a Fortran frontend for LLVM [2] named "Flang". At the time of writing, over 150 developers have authored 4895 commits to the Flang subdirectory on the main branch of the LLVM repository [1] (hereafter "upstream"); that work has brought Flang up to Fortran 95 [3] compliance. In addition to full support for Fortran 95, Flang can parse Fortran 2018 [4]. However, the Flang test suite does not exhaustively check Fortran standards-compliance, and the authors are unaware of a publicly available, comprehensive Fortran 2018 compliance test suite. Moreover, Fortran's evolution through three subsequent standards, commonly denoted Fortran 2003, 2008, and 2018 [4–6], implies significant remaining work to reach up-to-date compliance. This situation makes it attractive to explore ways to accelerate Flang's progression through Fortran language standards.

Berkeley Lab's ECP Flang project [7] focuses on writing static semantic tests for Fortran's parallel features, which are commonly referred to as "Coarray Fortran." We contribute these tests to the main branch of the upstream LLVM repository. We are also writing a comprehensive runtime test suite for parallel features. Fortran 2008 introduced parallel features but Flang lacks complete 2008 support, so we are also developing the Caffeine library to support runtime test execution.

Agile software development [8] encourages early release of working software and subsequent iteration toward complete solutions. To the extent that the concept of agility is associated with deliberate speed, agile practices might provide helpful avenues along which to attempt rapid development. This abstract and our poster focus primarily on the outcomes of approximately one person-year of effort, during which we reached 56% completion of static semantics test coverage. Although a direct comparison of speed with other development approaches would require a great deal more information and analysis, our subjective experience indicates that the agile practices described herein save us development time through frequent, rich developer interactions and through the unambiguous specification of feature requirements in the form of unit tests.

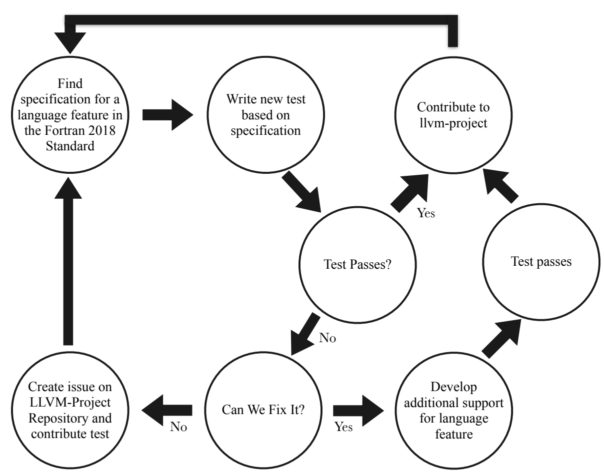

Fig. 1. Agile Test-Driven Development

II. Methodology

A. Agile Practices

The agile technique that we believe to be novel in its application to an open-source Fortran compiler is test-driven development (TDD) [9]. Our approach is illustrated in Fig. 1.

TDD starts with writing software tests in lieu of other forms of requirements documents and specifications. Given the resulting tests, developers add features to the subject software specifically to, and only to, support the tests.

Another agile practice we employ is pair programming: interactive sessions in which coders receive immediate feedback from a live observer. Pair programming keeps team members abreast of work progression in real time and enables synchronous communication of useful input. Our experience indicates that two minds are better than one. For example, we find and fix mistakes more quickly when one member focuses on writing the test while the other focuses on critiquing and contextualizing it within the broader aims of the project.

Once a new test is ready for wider dissemination and additional feedback, we post it to the LLVM community's code-review tool, Phabricator, which also records the developer dialogue and any resulting code updates. Throughout this process, we leverage an agile practice embraced by the wider LLVM community: continuous integration (CI) testing. LLVM's CI infrastructure uses automation to verify that contributed changes do not break pre-existing code.

B. Static Semantic Tests

We register a static semantic test for each parallel feature in a GitHub issue associated with a GitHub project [10] that captures the test's status. For each feature, such as the intrinsic function `num_images`, we write a test program containing a comprehensive set of standard-conforming and non-conforming statements exercising the given feature. Constructing each program requires consulting the formal definition of the feature in the Fortran 2018 standard [4]. The list of conforming invocations of `num_images`, for example, covers each of the function signatures defined in the 2018 standard: `num_images()`, `num_images(team)`, and `num_images(team_number)`, where `team` is of intrinsic type `team_type` and `team_number` is an integer. For each allowable form of invocation, different lines test keyword and non-keyword arguments; different keyword argument ordering; and variable, constant, or literal-constant arguments. We also test each allowable argument type. For example, the collective subroutine `co_sum` accepts a first argument of any numeric type, so we test each of `integer`, `real` and `complex` arguments in separate invocations.

The non-conforming statements include various disallowed statement forms and procedure invocations with arguments rendered invalid by their type, type parameters, array dimensions (rank), (im)mutability, or other attributes. For example, if the standard requires a dummy argument to have the `intent(out)` attribute, we intentionally violate the argument's intent specification by passing a literal constant. We also test procedure invocations with incorrect numbers of arguments, invalid or repeated keyword names, and other static semantics violations. We check constraints the standard requires compilers to report as errors as well as statically verifiable non-conformance that the standard specifies but does not require compilers to diagnose.

We use LLVM's `llvm-lit` testing tool [11], where tests pass if and only if all lines of code marked by an `ERROR` directive generate the error message provided in the directive and no unmarked lines generate an error. The `XFAIL` directive is used to mark a test that is expected to fail.

Our static semantics test suite reveals Flang's current level of parallel feature support. Many of our tests for intrinsic procedures, for example, have revealed that the compiler interpreted some intrinsic procedure references as user-defined procedures with missing interfaces. In such cases, we mark the corresponding tests with `XFAIL`, thus directing attention to features for which additional work will be needed for the test to pass. In other cases, the interface for the intrinsic procedure is already available, but our tests have revealed potential programmer mistakes that can be caught at compile time but are not currently caught. Where feasible, our team has added either the interface for the intrinsic procedures or the additional static semantic analysis.

III. Runtime Work

Because Flang cannot yet produce executable programs from Fortran 2018 source code, we develop runtime tests in a separate repository: Caffeine [12], a compiler-agnostic Fortran 2018 parallel runtime library that executes atop the GASNet-EX [13, 14] networking middleware – see Fig. 2. We are also developing Caffeine itself. We report elsewhere [15] on the design and implementation of Caffeine.

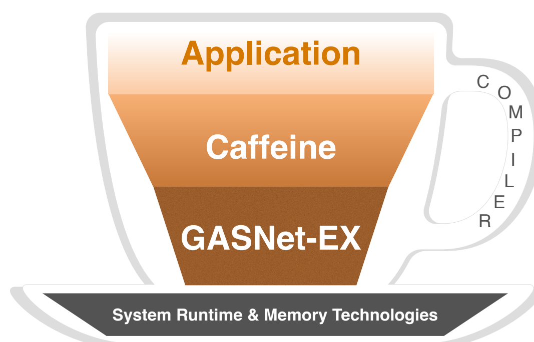

Fig. 2. Caffeine system stack

IV. Outcomes

We have deployed static semantics tests for 32 of 41 parallel statements and intrinsic procedures, including tests for intrinsic functions that support coarrays, collective subroutines, synchronization statements, and event statements. All 32 tests have been pushed to the main branch of the upstream LLVM Compiler Infrastructure project. Additional tests remain in-development or in-review. Based on missing features identified by our tests, the authors have contributed upstream additional static semantic analysis and error checking for 11 parallel features. We have also developed 44 runtime tests that we exercise by developing Caffeine.

Acknowledgment

This research was supported in part by the Sustainable Research Pathways for High Performance Computing (SRP-HPC), a project of the Sustainable Horizons Institute. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] *LLVM Compiler Infrastructure project*, https://github.com/llvm/llvm-project.

[2] *Flang Project in the Exascale Computing Project*, https://www.exascaleproject.org/research-project/flang/.

[3] Fortran Standards Committee JTC1/SC22/WG5, *Information technology — Programming languages — Fortran, ISO/IEC 1539-1:1997*. International Organization for Standardization (ISO), Dec 1997, https://www.iso.org/standard/26933.html.

[4] Fortran Standards Committee JTC1/SC22/WG5, *Information technology — Programming languages — Fortran, ISO/IEC 1539-1:2018*. International Organization for Standardization (ISO), Nov 2018, https://www.iso.org/standard/72320.html.

[5] Fortran Standards Committee JTC1/SC22/WG5, *Information technology — Programming languages — Fortran, ISO/IEC 1539-1:2004*. International Organization for Standardization (ISO), Nov 2004, https://www.iso.org/standard/39691.html.

[6] Fortran Standards Committee JTC1/SC22/WG5, *Information technology — Programming languages — Fortran, ISO/IEC 1539-1:2010*. International Organization for Standardization (ISO), Oct 2010, https://www.iso.org/standard/50459.html.

[7] Lawrence Berkeley National Lab, *Flang Testing project*, https://go.lbl.gov/flang-testing.

[8] M. Beedle and et al., *Manifesto for Agile Software Development*, https://agilemanifesto.org/.

[9] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[10] Lawrence Berkeley National Lab, *Semantics Tests for Parallel Features in LLVM Flang*, https://github.com/BerkeleyLab/flang-testing-project/projects/1.

[11] *lit - LLVM Integrated Tester*, https://llvm.org/docs/CommandGuide/lit.html.

[12] *Caffeine: CoArray Fortran Framework of Efficient Interfaces to Network Environments*, https://go.lbl.gov/caffeine.

[13] D. Bonachea and P. H. Hargrove, "GASNet-EX: A High-Performance, Portable Communication Library for Exascale," in *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18)*, ser. LNCS, vol. 11882. Springer, October 2018, doi:10.25344/S4QP4W.

[14] *GASNet*, https://gasnet.lbl.gov.

[15] D. Rouson and D. Bonachea, "Caffeine: CoArray Fortran Framework of Efficient Interfaces to Network Environments," in *Proceedings of the Eighth Annual Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC2022)*, November 2022, doi:10.25344/S4459B.