CMaize: Simplifying inter-package modularity from the build up

Zachery Crandall,[1,2] Theresa L. Windus,[1,2] and Ryan M. Richard[1,2,a)]

[1)] *Chemical and Biological Sciences, Ames National Laboratory, Ames, IA, 50011, USA.*

[2)] *Department of Chemistry, Iowa State University, Ames, IA, 50011, USA.*

(Dated: 19 February 2024)

There is a growing desire for inter-package modularity within the chemistry software community, to reuse encapsulated code units across a variety of software packages. Most comprehensive efforts at achieving inter-package modularity will quickly run afoul of a very practical problem, being able to cohesively build the modules. Writing and maintaining build systems has long been an issue for many scientific software packages that rely on compiled languages such as C/C++. The push to inter-package modularity compounds this issue by additionally requiring binary artifacts from disparate developers to interoperate at a binary level. Thankfully, the *de facto* build tool for C/C++, CMake, is more than capable of supporting the myriad of edge cases that complicate writing robust build systems. Unfortunately, writing and maintaining a robust CMake build system can be a laborious endeavor because CMake provides few abstractions to aid the developer. The need to significantly simplify the process of writing robust CMake-based build systems, especially in inter-package builds, motivated us to write CMaize. In addition to describing the architecture and design of CMaize, the article also demonstrates how CMaize is used in a production-level software.

[a)] Electronic mail: rrichard@ameslab.gov

I. INTRODUCTION

Developing scientific software, including computational chemistry software[1–4], has long been fraught with challenges. Developers often need some level of domain knowledge before they can even begin development. As a corollary, this means scientific software is often developed by domain experts, not software engineers, and the resulting software is often rife with technical debt, i.e., the cost (usually measured in developer time) of future refactoring efforts necessitated by developers choosing to "get something working" instead of "doing it right". While some technical debt is the result of "not knowing better", technical debt also arises because development of scientific software is often driven by funding agency priorities. This leads to developers prioritizing quantity of results over quality of software. Another difficulty with scientific software is that it targets phenomena which can only be accurately simulated via computationally demanding algorithms. When these algorithms are poorly optimized then, at best, the time-to-solution is much longer than necessary or, at worst, the simulation is outright impractical. Finally, computer science, computer engineering, and computer hardware are all rapidly changing. Thus, over time, all software has a tendency to become obsolete, i.e., unmaintained software tends to loose applicability, performance, and potentially even correctness. However, owing to many of the aforementioned difficulties, scientific software tends to become obsolete faster than traditional software.

There is an increasing realization that many of these problems can be partially alleviated by developing more modular chemistry software[3–7]. This is because when a software's architecture is comprised of discrete, encapsulated units, developers are provided with a separation of concerns. In turn, developers can (ideally) treat the modules as black-boxes, freeing up time to develop downstream code. When difficulties do arise, the difficulties are localized to the module, making it easier to address problems in a piecemeal manner. As an added benefit, solutions can be readily adopted by downstream consumers too. While most scientific software packages have embraced some level of **intra-package** modularization[3,7] (i.e., reuse of functions, libraries, and/or plugins within a single package), it is **inter-package** modularization (i.e., reuse of code units across packages) which has the potential to shift the paradigm of scientific software development.

Fundamentally modularity works by encapsulating algorithms and data. The primary difficulty pertaining to inter-package modularity is thus in ensuring compatibility between

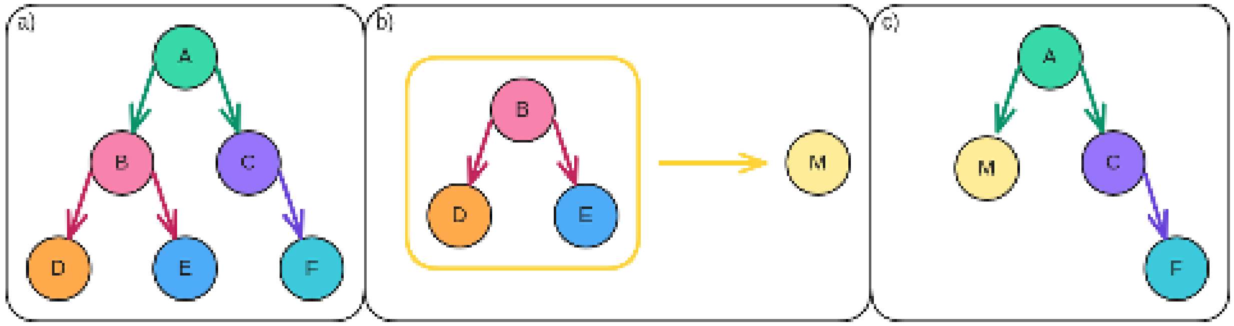

FIG. 1. **a** Schematic illustration of a program's call graph represented by a tree graph. In this graph, A is the root node, whereas D, E, and F are leaf nodes. **b** Inter-package modularization typically occurs by factoring out a leaf node, or as shown, an entire sub-tree. **c** The same call graph from **a**, except that it leverages the module "M" defined in **b**. Note that control flow is depicted top-down, left-to-right.

the module and all of the packages which may use it. Here compatibility has both a software design component and a build process component (*vide infra*). From the software design perspective, the consuming package must be structured in a manner that isolates the modularized algorithms. To better make this point consider Figure 1. Figure 1a shows the program's call graph represented as a tree. Most of the prominent, reusable, chemical-physics modules — e.g., Libint[8], LibXC[9], GauXC[10–13], and PCMSolver[14] — are designed to either replace "leaf nodes" of the call tree (i.e., "D", "E", and "F" in Figure 1a) or an entire sub-tree. Modularizing an entire sub-tree is shown in Figure 1b, where the sub-tree with root "B" is modularized into a new node "M". Figure 1c shows the same call graph from Figure 1a, but using module "M".

As a slight aside, we want to point out that in computational chemistry, modularity efforts stand to make largest impact, not by modularizing leaf nodes, but by modularizing internal nodes (i.e., those which are not roots or leaves). This is because it is often internal nodes which represent the most time-consuming operations. For example, the bottleneck step in a traditional four-center direct self-consistent field (SCF) calculation is forming **J** and **K** by contracting the density matrix with the electron-repulsion integrals. Assuming the formation of the electron-repulsion integrals is a leaf-node, this means that forming **J** and **K** will occur in internal nodes of the call graph. One of the easiest, and most flexible, ways to support modularizing internal nodes is to have the internal node obtain its child

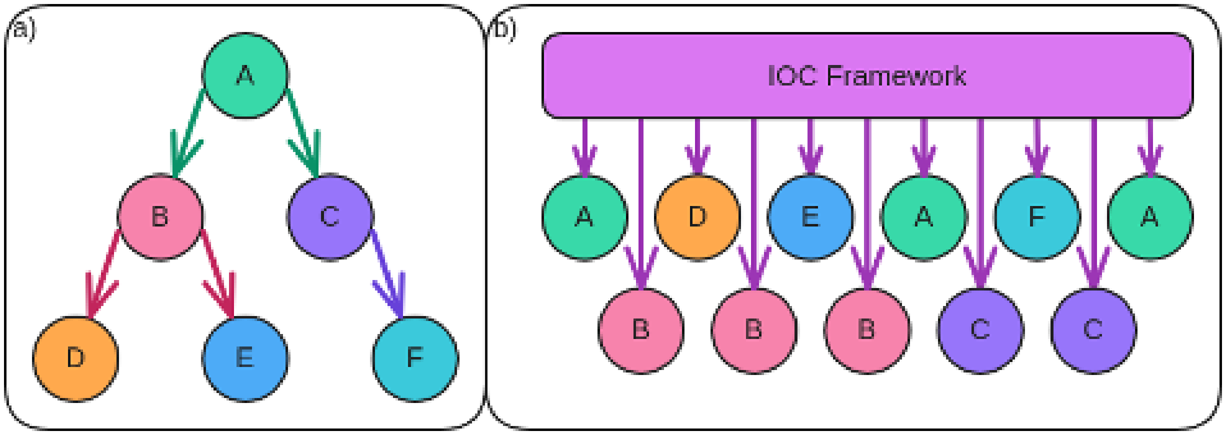

FIG. 2. **a** Schematic illustration of a traditional program's call graph. Each code unit decides the next code unit to call. **b** The same call graph as **a**, except that control dispatch is now managed by an inversion-of-control (IOC) framework, not the individual code units. Note that control flow is depicted top-down, left-to-right in each panel.

nodes from a dynamically populated list of nodes, instead of hard-coding the dispatch logic directly into the internal node. This technique is known as inversion-of-control (IOC) and is the mechanism underlying IOC frameworks such as PluginPlay[7]. The NWChemEx team is currently exploring the use of PluginPlay to create a fully modular SCF implementation.

Since the IOC framework makes the call on the caller's behalf, the use of an IOC framework in essence converts every node into a leaf node. This process is schematically shown in Figure 2. Figure 2b shows how the traditional call graph shown in Figure 2a is actually executed using an IOC framework. In Figure 2b control first enters the program through the IOC framework. The IOC framework somehow (usually through user input) determines that it needs to run code unit "A" first. "A" then runs until "A" needs to call another code unit; at this point, "A" returns control to the IOC framework, which then runs "B". Via similar logic, "B" will determine that it too needs a subcall and will also return control back to the IOC framework. The framework then runs "D". Since "D" does not require subcalls, when "D" returns, the IOC forwards the results of "D" back to "B" and "B" continues running. This continues until the entire call graph depicted in Figure 2a has been run.

As already mentioned the other aspect to module/package compatibility lies in the build process, i.e., the process of turning the source code of a project into a usable package. Packages relying on reusable modules will be most performant if data communication happens in native formats, i.e., communication should avoid replicating distributed data, converting data, serializing objects, writing to disk, etc. This requires application programming interface (API)-compatibility among modules and in turn places significant onus on the build system (i.e., the set of build tools and code needed to actually carry out the build process for a software package) to find/build a set of modules which satisfy the version and configuration needs of the entire stack. This problem is exacerbated by the fact that the more modules a software package relies on, the more compatibility criteria there are to satisfy. Making matters even worse, for performance reasons, many of the modules will be written in compiled languages — e.g., Fortran, C, or C++— which requires ensuring the resulting packages are not only API-compatible, but also application binary interface (ABI)-compatible (e.g., compiled for the same architecture and operating system). The point is, even even though package management and build processes have historically been an afterthought, if the chemical-physics community wants to increasingly rely on reusable components, there is a need for robust and reliable build systems.

The impact of the build process on inter-package modularity efforts should not be understated; at least one previous effort, the Common Component Architecture (CCA) project[15], failed in part because of the complexities associated with building software. Thankfully, since the CCA's efforts, a number of aspects of building scientific software have advanced. First, scientific software developers have largely consolidated around C/C++ for writing new performance critical source code , and Python otherwise, which limits the programming languages to prioritize. Second, thanks to libraries like pybind11[16] and Cppyy[17] language interoperability between C/C++ and Python is fairly simple. Third, CMake[18] has emerged as the *de facto* build tool for developing C++ build systems. In practice, CMake is an incredibly powerful, cross-platform build tool that is capable of handling the intricacies of building scientific software; however, CMake's verboseness, rapidly evolving language features, and lack of user-friendly abstractions result in a steep learning curve for new users. Fourth, online code repositories make obtaining and contributing to source code easy. Finally, the power and storage capacity of most modern computers means that recompiling is a relatively quick process, and multiple versions of a package occupy a relatively trivial amount of space. Together the fourth and fifth points mean that build tools, like CMake, and package managers (i.e., software which facilitates finding, building, and tracking packages and their dependencies) can automatically obtain dependencies for users and ensure they are built in a manner that is compatible with the project.

The present paper is predicated on the assumption that **in order for inter-package modularity to become widespread we must simplify the build process** of (compiled) scientific software. To this end, we have developed CMaize (`https://github.com/CMakePP/CMaize`). CMaize is a CMake module (code written in the CMake language meant to serve as extensions to the CMake language) which provides succinct, user-friendly APIs for declaring dependencies, building targets, and defining packages. Since CMaize is written purely in CMake, users of CMaize have the full CMake ecosystem at their disposal. The remainder of the paper is organized as follows: in Section II we motivate the need for CMaize and briefly review other relevant options, in Section III we explain the architecture and design rationale underlying CMaize, in Section IV as a proof-of-concept we show how CMaize has simplified the build process for PluginPlay[7] (and many other packages in the NWChemEx Community). Finally, in Section V we end with a summary and future outlook for CMaize.

II. CMAIZE STATEMENT OF NEED

As discussed in Section I, scientific software developers are at present overwhelmingly developing new software in C/C++ and Python. Thanks to standardized build processes[19,20], and package managers like pip[21] (and the underlying package repository[22]), building Python packages is comparatively easy — and not considered further here. Unfortunately, there are no standardized build processes for C/C++ projects and package manager efforts[23–27] are relatively new (most package management projects are less than 10 years old). Generally speaking, the build processes for C and C++ are similar and for brevity we no longer distinguish between the two programming languages opting to exclusively use C++. Fortunately, CMake has emerged as the *de facto* build tool[18,28,29] for managing the build process of C++ projects.

The CMake software package is comprised of a series of executables, of which we are only primarily interested in the `cmake` executable . The remaining executables facilitate additional DevOps processes such as testing, analyzing test results, and creating installers. Users of CMake write scripts in the CMake programming language, detailing the build process of their project's packages. When run, the `cmake` executable generates a build

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

FIG. 3. Contents of the C++ source file `hello_world.cpp`. The source for this executable will be used to illustrate the verboseness of a CMake-based build system.

system from the user-provided scripts and then executes the generated build system. While this means that CMake is officially a build system generator, in practice, build system generation happens transparently to the user and thus, for all intents and purposes, CMake (when combined with the scripts detailing the build process) can be seen as a build system.

As a build system, CMake is incredibly feature-rich and flexible. In practice, between CMake's feature repertoire and the CMake programming language, CMake can support almost any use case a build system developer may encounter. However, with CMake's power comes an extremely steep learning curve, especially when the goal is to develop reliable, cross-platform, and sustainable build systems. As scientific organizations begin to manage more projects —which will be a natural progression as focus shifts towards inter-package modularity) the sustainability of the build systems will become more important. If developers fail to afford build systems (and other infrastructure) the same considerations as traditional source code, the result will be additional technical debt and lost developer productivity. The following subsections detail some of the hurdles encountered when striving to write sustainable build systems with CMake.

A. Verboseness

Since C++ has no standardized build process, CMake has been designed with flexibility in mind. More specifically, most CMake commands make little to no assumptions about the project being built. The result is that CMake-based build systems are very verbose. To illustrate this point consider arguably the simplest C++ project possible, a single source executable with no dependencies (aside from the standard library). The source code complexity of the executable is irrelevant, so as a concrete minimum working example we opt for the "hello world" executable shown in Figure 3. We presently stipulate that the simplest CMake-based build system for this "hello world" project should:

1. build the executable,
2. install the executable,
3. be configurable (i.e., users should be able to control where the executable is installed, which compilers are used, etc.),
4. be usable by other CMake-based build systems both in source and installed forms, and
5. build warning free.

Some readers may reasonably argue that a minimum build system need only satisfy a subset of these stipulations; however, we chose these stipulations to align with our overall motivation, namely inter-package modularity. In particular, the build system of our project should adhere to current CMake best practices in order to ensure it can seamlessly be incorporated into other CMake-based build systems. Subject to these stipulations we need to write two files for our build system: Figure 4 shows the `CMakeLists.txt` detailing the build process, and Figure 5 shows a template for the configuration file. Unlike the source file for the C++ executable, these files require a bit of explanation.

First focusing on `CMakeLists.txt`, lines 1 and 2 are needed to ensure the build is warning free. Line 4 is needed to actually build the executable. Lines 6 through 12 install the executable. Lines 14 through 55 generate and install the packaging files needed so that other CMake-based build systems can locate installed versions of our executable via CMake's `find_package` command. Hence, the bulk of Figure 4 is concerned with packaging the executable. The source code for packaging our executable is adapted from CMake's official documentation[30]; thus, if there is a more succinct way to generate the configuration files, CMake is not advertising it. By comparison to the `CMakeLists.txt` file, the `hello_world-config.cmake.in` file is much simpler. Line 1 is boilerplate that CMake will replace in the generated file. Line 5 imports the package's targets (the executable). Finally, line 7 ensures that the targets were loaded correctly. Although it is a bit premature at

```
make_minimum_required(VERSION 3.5)
project(hello_world VERSION 1.0.0)

add_executable(hello_world hello_world.cpp)

include(GNUInstallDirs)
set(target_name hello_world)
install(
    TARGETS ${target_name}
    EXPORT ${PROJECT_NAME}Targets
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)

include(CMakePackageConfigHelpers)

set(install_cmake_dir ${CMAKE_INSTALL_LIBDIR}/cmake)
install(
    EXPORT ${PROJECT_NAME}Targets
    FILE ${PROJECT_NAME}-targets.cmake
    NAMESPACE ${PROJECT_NAME}::
    DESTINATION ${install_cmake_dir}/${PROJECT_NAME}
)

set(version_file ${PROJECT_NAME}-config-version.cmake)
set_property(
    TARGET ${target_name}
    PROPERTY VERSION ${${PROJECT_NAME}_VERSION}
)
set_property(
    TARGET ${target_name}
    PROPERTY
        INTERFACE_${target_name}_MAJOR_VERSION
        ${${PROJECT_NAME}_VERSION_MAJOR}
)
set_property(
    TARGET ${target_name}
    APPEND PROPERTY COMPATIBLE_INTERFACE_STRING ${target_name}_MAJOR_VERSION
)
write_basic_package_version_file(
    "${CMAKE_CURRENT_BINARY_DIR}/${version_file}"
    VERSION ${${PROJECT_NAME}_VERSION}
    COMPATIBILITY SameMajorVersion
)

set(config_file ${PROJECT_NAME}-config.cmake)
configure_package_config_file(
    ${CMAKE_CURRENT_SOURCE_DIR}/${config_file}.in
    "${CMAKE_CURRENT_BINARY_DIR}/${config_file}"
    INSTALL_DESTINATION ${install_cmake_dir}/${PROJECT_NAME}
)
install(
    FILES ${CMAKE_CURRENT_BINARY_DIR}/${config_file}
          ${CMAKE_CURRENT_BINARY_DIR}/${version_file}
    DESTINATION ${install_cmake_dir}/${PROJECT_NAME}
)
```

FIG. 4. Contents of the `CMakeLists.txt` file detailing the build process for the C++ source code shown in Figure 3.

```
@PACKAGE_INIT@

set(_hw_package_name hello_world)
set(_hw_target_file ${_hw_package_name}-targets.cmake)
include(${CMAKE_CURRENT_LIST_DIR}/${_hw_target_file})

check_required_components(${_hw_package_name})
```

FIG. 5. Contents of the `hello_world-config.cmake.in` template file needed by the build system shown in Figure 4.

```
cmake_minimum_required(VERSION 3.5)
project(hello_world VERSION 1.0.0)

include(cmaize/cmaize)

cmaize_add_executable(hello_world SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR})
cmaize_add_package(hello_world)
```

FIG. 6. Contents of the `CMakeLists.txt` when the build system shown in Figures 4 and 5 is instead written using CMaize.

this point, for comparison, Figure 6 showcases what the same build system looks like using CMaize (more details regarding the CMaize build system can be found in Section IV).

As can be seen, even a minimal CMake-based build system requires a fairly substantial amount of code, much of which is boilerplate (a point illustrated by factoring out the origin of information instead of hard-coding the values). As evidenced by tutorials prominently showcasing boilerplate[28,31,32] and a growing number of template repositories[33–40] (essentially version-controlled copy/paste), the community seems to have accepted that boilerplate is an intrinsic aspect of writing a CMake-based build system. Relying on heavy use of boilerplate is an anti-pattern that hinders sustainability because it runs afoul of the "Don't Repeat Yourself" (DRY) principle[41]. Proponents of the DRY principle argue that by adhering to DRY the resulting source code is more sustainable because then the information only exists in a single place, the code is easier to read, and changes propagate automatically , while copy/paste solutions require changing every pasted version manually. Ultimately, succinct build systems are easier to sustain, and thus the verboseness of CMake must be addressed.

B. Build System Instability

CMake version 3.0 was released in 2014[42]. As of December 2023, CMake is now at version 3.28[43]. CMake developers adhere to semantic versioning meaning each of the 28 minor updates have not broken the API while also adding at least one new feature. Unfortunately, even though CMake developers guarantee backwards *language* compatibility (build systems written with older versions of the CMake language can still be parsed by newer versions of CMake) they do not guarantee backwards *build system* compatibility (build systems written with older versions of CMake can still be leveraged by build systems written with newer versions of CMake).

To better illustrate this problem, note that when CMake 3.0 was released, finding dependencies relied heavily on the use of built-in or user-defined `FindXXX.cmake` files. Here `XXX` is a placeholder for the name of the dependency, so for example `FindBoost.cmake` would be used to find the Boost C++ libraries[44]. The `FindXXX.cmake` files were responsible for finding the dependencies and then setting variables such as `XXX_INCLUDE_DIRS` (the location where public header files reside). Starting with about CMake 3.5[45] in 2016, the `FindXXX.cmake` files shipping with CMake started also providing targets. CMake 3.5 is also the start of what is termed "Modern CMake"[29]; one of the central tenets of which is that build systems should be target-based. In turn, with CMake 3.5 developers needed to update their user-defined `FindXXX.cmake` files so that they also produced targets. Around the same time there was also a push[29] to do away with `FindXXX.cmake` files entirely and instead rely on configuration files provided by the dependency (see for example Figure 5). However, until a dependency's build system was updated to generate configuration files, consumers of the package needed to continue maintaining their own `FindXXX.cmake` files. This in turn pressured the dependency's build-system developers to update the build system to generate configuration files. Unfortunately, once a dependency switched over to generating configuration files, consumers no longer needed their `FindXXX.cmake` files and had to yet again modify their build systems.

Another example comes from building dependencies. Up until CMake 3.11[46], if a build system maintainer wanted to automatically build dependencies for the user, best practice was to rely on superbuilds[47] (essentially containerized, nested builds). CMake 3.11 introduced the `FetchContent` module. With `FetchContent`, instead of nested build systems, there is a single dynamically discovered build system, i.e., the build system of the package is the union of its build system plus the build systems of its dependencies. In practice, many CMake-based build systems written pre-CMake 3.11 could not immediately be used with `FetchContent`. For example, they may have relied on targets with conflicting names, overridden global state, or made any other myriad of assumptions which may have worked within the isolation of a superbuild, but now invalidated assumptions made by another build system.

The point of the above examples was to show that because of CMake's viral nature (the build system of a package leverages the build system of its dependencies) CMake-based build systems have historically been somewhat unstable. In practice, this can partially be attributable to the verboseness of CMake (see Section II A). Without succinct APIs encapsulating the details of how the build system works, the evolution of best practices has required developers to rewrite their build systems in terms of the new features. However, keep in mind that refactoring was not required because the package suddenly obtained/used the dependency in a different way, but because best practices for registering the package/dependency relationship with CMake changed. Put another way, if CMake possessed a stable API for succinctly describing the package/dependency relationship, these changes could have been encapsulated as implementation details.

C. Poor Dependency Management

The previous subsection, Section II B, provided some examples of how CMake's strategy for dependency management has changed over the years. At this point, CMake's package management solution amounts to the `find_package` function[48] and the `FetchContent` module[49]. The `find_package` function is responsible for locating a requested package and the `FetchContent` module can be used to build the package if it is not found. While conceptually this is a complete solution, the reality is that there is a reason that software developers often describe the process of dependency management as "dependency hell."[50]

Developers have a tendency to think of a dependency requirement as "my project depends on Python", but in many cases the requirement is actually more like "my project depends on the the stable ABI static libraries found in the development version of CPython version

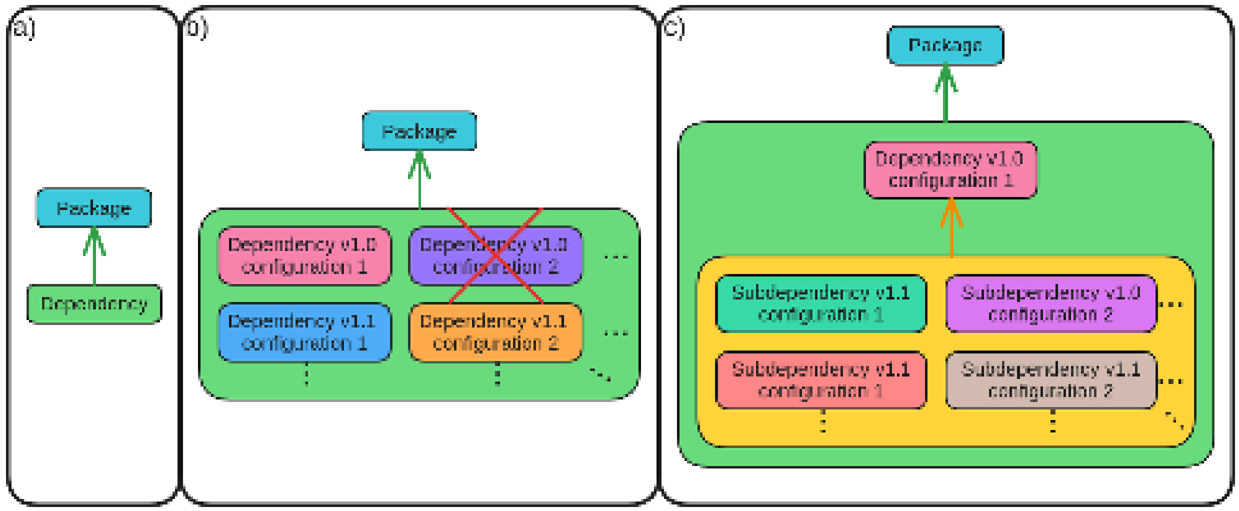

FIG. 7. **a** Often dependencies are thought of as being a single entity. **b** In practice, most dependencies have many variants. Packages may only work with a subset of the possible variants. **c** The dependencies of the dependency, i.e. subdependencies, also have variants. The total number of variants for a particular dependency depends not only on the number of variants of the dependency, but also on the number of subdependency variants.

3.11.4 or later." The point is that most dependencies have many "variants" (i.e., the different versions, configurations, and builds) and the dependent software may work with all variants, a subset of the variants, a single variant, or even none of the variants. Making matters worse, for a dependency D, the number of variants of D depends not only on the number of combinations for configuring and building D, but also on the number of variants for each of D's dependencies. While designing D with good software engineering practices can often encapsulate the impact subvariants have on the package, the reality is that there exists many dependencies which leak their dependency details. The above is summarized by Figure 7.

Proper dependency management requires being able to: describe the variants, determine what is the variant of an installed package, compare a variant to a dependency requirement, and finally install a variant of a dependency when a suitable variant can not be located. Given the sheer number of variants, managing dependencies correctly and efficiently is a hard task. As with other aspects of the build process, to avoid alienating potential use cases, CMake's `find_package` and `FetchContent` modules require substantial user input in order to operate. Furthermore, both `find_package` and `FetchContent` effectively defer the entire process of describing and comparing variants to the user (though CMake can facilitate describing and comparing some basic parameters such as versions). That said, there is a very reasonable argument[51,52] to be made that dependency management should not be done by CMake, but by a package manager (i.e., tools such as Aptitude[53], YUM[54], DNF[55], Homebrew[56], Spack[26], pip[21], Conan[23], etc.) dedicated to resolving dependency graphs. Nonetheless, fully separating CMake from dependency management is not practical as many package managers are language-specific and the build system is thus needed to handle cross-language dependencies. Thus, there is a need for better CMake package manager support. As a disclaimer, package manager support was fairly recently added to CMake in version 3.24[57] (released in late 2022); however, most build systems do not yet appear to be leveraging this feature, and, as noted in Section II B, it is likely that as best practices develop build systems will need to be refactored.

D. Difficult to Extend

Scientific software packages routinely involve multiple programming languages. Ideally, the package should provide a comprehensive build system which builds the entire package, not just the piece of it written in a particular programming language. While CMake is primarily targeted at C++, CMake has supported Fortran since version 2.2[58] released in 2005. Feature releases have grown the number of languages CMake supports and the level of support. For example, version 3.8[59] added native support for Compute Unified Device Architecture (CUDA) extensions and version 3.12[60] greatly improved CMake's ability to support Python. Nonetheless, there are many other potential programming languages one may reasonably encounter while building scientific software, e.g., Julia, Rust, or R. Thankfully many of the newer programming languages often have a single toolchain. Thus newer programming languages often have more standardized build systems when compared to C++. In turn, many software developers may find it easier to absorb the build process of other programming languages into a CMake-based build system, rather than the other way around. However, expecting the CMake developers to natively support every programming language is an unreasonable request. Instead it would be ideal if it was easier for users to extend CMake to additional programming languages and notably the package managers and build tools associated with those programming languages.

E. Other Potential Solutions

The previous subsections described the high-level complications with writing CMake-based build systems. However, CMake is the *de facto* build system for C++ projects. Consequently, many developer tools support CMake-based build systems[61] (e.g., CLion[62], and Vim[63]). So while there are alternative tools[64–67] for writing C++ build systems, at present, it seems unlikely that they will usurp CMake. Thus, we designed CMaize as a build tool which:

1. is CMake-based,
2. limits boilerplate and redundancy,
3. has stable APIs for writing build systems,
4. leverages existing package/dependency managers instead of "reinventing the wheel,"
5. works with package/dependency managers to ensure ABI compatibility, and
6. is easily extended to new programming languages, package managers, and build tools.

CMaize is not the community's first attempt at addressing the above needs. Generally speaking, we can categorize previous solutions as being: generators, CMake extensions, or CMake-based package managers. Within the generator approach, the build tool usually takes as input the basic specifications of the project and then generates a CMake-based build system. CMake extensions are CMake modules which provide simplified interfaces for performing common build system tasks. The implementation of the module then maps the user's input to CMake's native functionality. CMake-based package managers either integrate with the `find_package` function and the `FetchContent` module or are used as alternative interfaces. As a disclaimer, the following descriptions come from documentation and the projects' source code; thus the accuracy of the following descriptions depends on how accurately/completely each project has portrayed itself.

Autocmake[68] and cmake-init[69] have attempted to address the community's needs via the generator approach. While generators can conceivably address all of the community's needs (after all, CMake works by generating other build systems), unless a generator is a complete build system solution, the user ends up needing to modify or patch the generated

CMake-based build system. The net result is that the user only relies on the generator for creating a starting point for their project's build system and the generator amounts to a sophisticated mechanism for copy/paste-ing the CMake-based build system. The other problem with generators is that they usually do not integrate seamlessly into existing CMake-based workflows (though the generated build systems do). As far as we are aware, both Autocmake and cmake-init suffer from both of these problems, as they are not complete build system solutions and require additional steps outside the traditional CMake workflow.

Build, Link, and Test (BLT)[70], CMake++[71], Izzy's eXtension Modules (IXM)[72], and project_options[73] strive to simplify writing CMake-base build systems via the CMake extension approach. BLT is a simplified, CMake-based build system for high-performance computing (HPC) applications. Users of BLT can stitch together their build system from BLT functions; however, these functions presuppose a relatively small set of dependencies and target platforms. While BLT is likely a boon for projects with the same software stack and use cases as the BLT developers, it is unclear whether BLT can be extended without directly modifying BLT. CMake++ extends the CMake language to include support for object-oriented programming. CMake++ leverages the object-oriented extension to implement a number of additional CMake features meant to facilitate writing CMake-based build systems. Unfortunately, CMake++ was a single-developer project that is now inactive. Furthermore, the lack of low-level API and developer documentation makes CMake++ hard to use and extend. Similarly, the IXM project seemed promising, but is now inactive. IXM aimed to simplify CMake-based build systems by relying on a concept called "blueprints"; though exactly how this would have worked is hard to glean from the sparsely available documentation. Like BLT, project_options provides a streamlined, CMake-based build system, but the resulting build system is heavily tied to the preferences of the project_options developers. While each of these CMake extensions have merit and interesting features, either they are now inactive or they lack the generality needed to support the multidisciplinary dependencies encountered in scientific software.

CPM[27], CPM.cmake[74], and Hunter[25] are CMake-based package managers. CPM is a now inactive, CMake package manager targeted at static linking of dependencies (which in turn allows full encapsulation of the dependencies). CPM.cmake is an independently developed alternative project to CPM with more general support for finding and building dependencies. Unfortunately CPM.cmake is not a full build system tool and is exclusively focused on facilitating the finding and building of dependencies. Hunter is an alternative package manager which is implemented by requiring all dependencies to live in a central repository. In turn, developers must "Hunter-ize" every dependency before they can use Hunter. Making matters worse, Hunter's central repository seems to be largely maintained by a single developer, which is a fairly precarious design. As detailed in Section III, CMaize is designed to be extendable to external package managers and thus these other projects could conceivably be integrated with CMaize to provide users additional package management options.

While they lie somewhat outside the categories of other solutions, two other efforts, i.e., Cinch[75] and cmake-get[76], bear mentioning. Cinch is a tool for automatically discovering and building the source files, unit tests, and documentation of a package. Cinch is able to automatically build a package's dependencies, but only if the dependencies also use Cinch. cmake-get[76] is a project which was designed as an alternative to superbuilds. Both Cinch and cmake-get are now inactive projects. Finally, for completeness, we mention CMakeBuild[77]. CMakeBuild was started by the NWChemEx team and ultimately served as an initial design for CMaize. At this point the authors' development focus lies entirely with CMaize. Nonetheless, CMakeBuild is still used by a few projects.

To summarize, CMake is the *de facto* build system for C++ projects. The ubiquitous nature of CMake-based build systems means that diverging from standard CMake workflows would break compatibility with many existing developer tools and workflows. Nonetheless, CMake is a verbose, rapidly evolving programming language, with support largely targeted at compiling C++ software, i.e., CMake lacks support for many build tools and programming languages. Thus, there is a need for extensions to the CMake language which make it easier to write CMake-based build systems and extend it to new use cases. While there have been a number of previous efforts in these directions, we are unaware of any active project addressing all of these needs. It is for these reasons that we have opted to write CMaize.

III. CMAIZE: ARCHITECTURE AND DESIGN

This section motivates and describes the high-level design of CMaize starting with the overall architecture. For brevity we only summarize the design aspects necessary for the reader to conceptually understand how CMaize works. More design details are available in

CMaize's developer documentation[78].

CMaize is founded on needing software which addresses the following considerations:

1. **honors established CMake workflows**. CMake is the *de facto* build system for C++ projects and it is unlikely that any solution we develop will change this in the foreseeable future. Therefore, our solution must integrate seamlessly into existing CMake-based workflows.

2. **CMake based**. Conceivably one could write a Python (or similar programming language) library, with CMake bindings, to manage the build system. However, this would inevitably require rewriting much of the existing feature set of CMake. Instead, we stipulate that the solution should be written in CMake so that it can leverage as much of CMake's feature set (present and future) as possible. Being CMake-based also makes it easier to satisfy the **honors established CMake workflows** consideration.

3. **minimize verboseness**. In our experience, much of the maintenance headache of managing CMake-based build systems come from their verboseness (specifically their redundancy). An ideal solution takes only the inputs necessary to specify the build system. The mechanism for mapping those inputs to the resulting build system are encapsulated from the user.

4. **target support**. Modern CMake is target based. To integrate into existing CMake-based workflows it is thus essential that the build system be target-based. This also facilitates letting the user fine-tune the build system by dropping down to CMake's native target APIs.

5. **package manager support**. Dependency management is an incredibly complicated task. Traditional CMake places much of the work on the build system maintainer. In many cases, package managers can alleviate much of this burden. It is therefore essential that CMaize be able to leverage existing package managers (particularly those for other programming languages).

6. **recursive**. CMake-based build systems are recursive, meaning the CMake-based build system of a package leverages the build system of its dependencies. CMaize must be capable of functioning in a recursive environment.

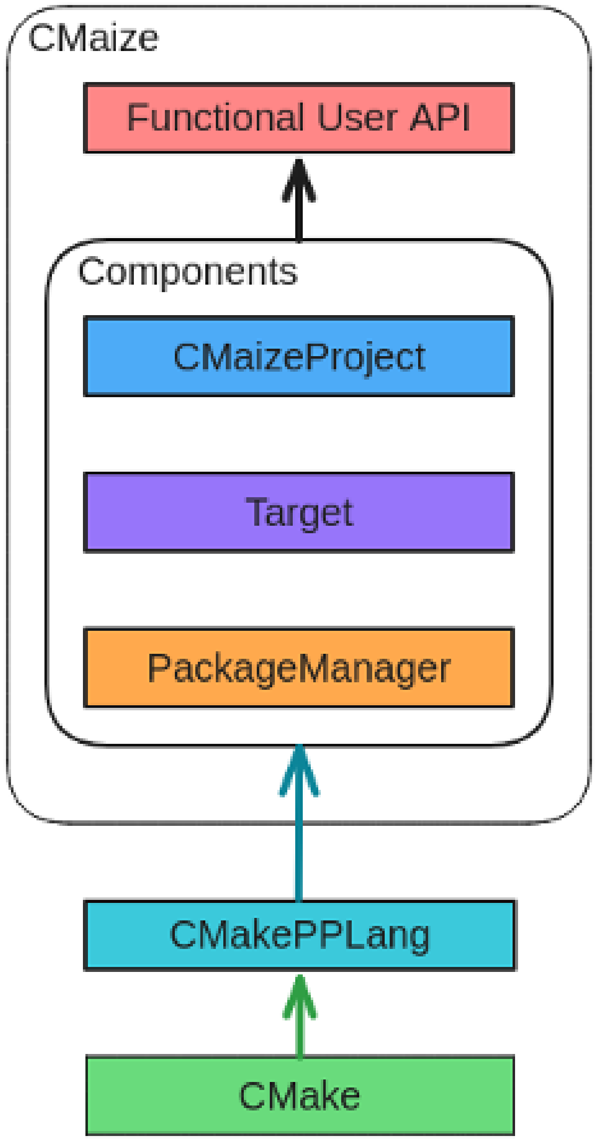

FIG. 8. Overall architecture of CMaize. Code units within the box labeled "CMaize" are pieces of CMaize; code units outside the box are dependencies.

7. **object-oriented**. While not strictly necessary to address the above considerations, many modern developers prefer object-oriented languages to functional languages. For our particular needs, we feel that object-oriented paradigms greatly facilitate extending CMaize to new programming languages and build tools by encapsulating each in their own class.

Based on the previous considerations, we have opted to structure CMaize as shown in Figure 8. Users of CMaize interact with CMaize through the user API, which is discussed in detail in Section III A. For now, we note that we have opted for CMaize's user-facing API to be functional. This decision stems from the **honors established CMake workflows** and **CMake based** considerations. More specifically, since the CMake language is functional in nature, providing a functional API facilitates user adoption and minimizes problems from mixing programming paradigms. That said, in accordance with consideration **object-oriented**, the user API is implemented in terms of objects. CMake support for objects is provided by the CMakePPLang[79] extension. CMakePPLang defines a map data type which it then used as the basis for defining classes; this is similar to how Python works. Importantly, CMakePPLang is implemented entirely with the CMake programming language, consistent with the **CMake based** consideration. It should also be noted that CMaize will automatically obtain CMakePPLang, so its use is an implementation detail.

The core of CMaize is comprised of three object-oriented components. The first component is the project component and its primary class `CMaizeProject` (see Section III B for more details). The project component is responsible for collecting information about the overall project including its name, version, dependencies, and targets. In practice, `CMaizeProject` objects behave like namespaces, helping to keep the state of the current project isolated from that of dependency or parent projects (this in turn helps address the **recursive** consideration). The second component is the target component which primarily centers on the `CMaizeTarget` class hierarchy. As the **target support** consideration noted, modern CMake is target-based. The `CMaizeTarget` class hierarchy provides an object-oriented API for creating, inspecting, and manipulating CMake targets. Of note, the `CMakeTarget` is designed so that it can be extended to additional programming languages without leaking the details of how the implementation chooses to map the programming language to a CMake target. The final component of CMaize is the package manager component and the accompanying `PackageManager` class hierarchy. CMaize's `PackageManager` component facilitates using CMake's built-in dependency management tools (i.e., `find_package` and `FetchContent`) with existing package managers. Of note, through inheritance it is possible to extend CMaize's `PackageManager` class hierarchy to additional programming languages. Ultimately, `PackageManager` objects facilitate mapping project specifications to `CMaizeTarget` objects.

A. User API

The overall architecture considerations of a **CMake based** solution and needing to **minimize verboseness** gave rise to the need for CMaize's user API. Superficially, most build systems need the same information, and follow the same steps. The CMaize user API is designed to follow the common build system workflow:

1. **Project setup**. This is where the build system developer establishes high-level metadata about the project such as the name, version and programming languages. This is also the point where build tools needed by the build system are located.

2. **Build options**. Projects typically support different configurations. For example, some targets may only build on certain architectures or if a feature is enabled. The build system developer needs to enumerate these options.

3. **Find or build dependencies**. With a configuration in hand, the build system starts to search for dependencies satisfying the configuration's requirements. User friendly build systems will typically build these dependencies if they can not be located. The build system developer needs to list the dependencies and what dependency variants are needed.

4. **Build project targets**. At this stage the build system knows the details of the dependencies, but still does not know how to build the project's targets. Input needed from the developer at this stage is primarily which source files map to which target and the dependencies of each target.

5. **Test the project**. With knowledge of the project's targets, including those from dependencies, the next step is to define tests which will establish that the built artifacts work as expected. This often is a repeat of the **Build project targets** step, but with the new targets only being used for testing, i.e., the new targets should not be distributed with the resulting package.

6. **Install package**. Finally, once the correctness of the targets have been verified, the build system will assemble the final package and install it. This requires the build-system developer to state the targets to install and potentially the location for the installed executables or libraries.

CMaize's (functional) user API is designed to allow the user to configure each of these steps, with a minimal amount of information. To minimize user-required input, CMaize makes a number of assumptions (which can be overridden), e.g., the installation tree will follow best practices for the targeted operating system, users have grouped their targets into separate directories, and the user wants installed versions of the package to work with CMake's `find_package` function (and CMaize's equivalent). CMaize will also automatically propagate the information for the user, e.g., there is no need to specify install locations more than once, track names of auto-generated files, or restate the dependencies of a target. Full details of CMaize's user API can be found in CMaize's user documentation[80]. For now, note that Figure 6 already presented a minimal CMaize build system written in terms of CMaize's user API. A more complicated example is deferred to Section IV.

B. CMaizeProject

The need for CMaize's project component, specifically the `CMaizeProject` class, was motivated by the **object-oriented** and **recursive** architectural considerations. The `CMaizeProject` class must track all of the project's state including:

1. **package specification**. The build process results in a variant of the project. Fully describing the variant is the responsibility of the class `PackageSpecification`. The `PackageSpecification` class captures the project's metadata such as name, version, and build options which are often sufficient to establish API-compatibility from among different variants, but not ABI-compatibility. ABI-compatibility is better left to the package managers.

2. **programming languages**. Managing build tools for the project requires knowing what programming languages the project is written in, e.g., knowing the project contains C++, means we will need a C++ compiler, knowing the project contains CUDA means we need to find the CUDA toolkit libraries, and knowing the project contains Python means we will need a Python interpreter.

3. **package manager management**. The `CMaizeProject` component tracks the package managers the project uses.

FIG. 9. Architecture of the `CMaizeProject` class.

4. **target tracking**. The `CMaizeProject` component needs to track the CMaize targets (and by proxy, the CMake targets implementing them) associated with the resulting package including dependency targets.

Figure 9 shows the architecture of the `CMaizeProject` class. The class is primarily a data aggregate. CMake, by design, only ever has one active project, an assumption CMaize capitalizes on in order to automatically build and populate `CMaizeProject` objects. In other words, the functions comprising CMaize's user API (see Section III A) first grab the active `CMaizeProject` object (creating it if it does not exist) and then manipulate its state according to the user's input, e.g., `cmaize_find_or_build_dependency` will use the package managers associated with the project to create either an installed target (if the requested variant is found) or a build target (if the requested variant must be built) and then add it to the active `CMaizeProject` object.

Internally, the `CMaizeProject` maintains a `PackageSpecification` object for the project. The partitioning between the state in the `PackageSpecification` class and the remainder of the state in `CMaizeProject` is based off of whether the information is primarily needed after building the project (in which case it lives in `PackageSpecification`) or if the information is primarily needed for building the project (in which case it lives directly in `CMaizeProject`). We note that each `CMaizeProject` object contains its own state and does not alias the state of another `CMaizeProject`. In turn, in recursive projects it is possible for different `CMaizeProject` objects to rely on different languages, package managers, and even dependency variants (ensuring that the different variants are compatible is the responsibility of the package manager).

C. CMaizeTarget

In designing CMaize's architecture we noted the **target support** consideration, i.e., that modern CMake is target based. The repercussion of this consideration is that no matter how CMaize works, it must ultimately produce correctly initialized CMake targets. While this may seem restrictive, thanks to CMake's `add_custom_command`[81] it is actually possible to create CMake targets with fairly broad functionality. In turn it is possible for CMaize to support targets for a wide variety of programming languages and build artifacts. To manage the complexity of this task and remain consistent with the **object-oriented** consideration, CMaize introduces a class hierarchy with the common base class, `CMaizeTarget`. Creation of the current project's target objects is handled by CMaize's user API. Creation of target objects for dependencies is handled by the `PackageManager` object (see Section III D).

In designing the `CMaizeTarget` class hierarchy, we noted the following design considerations:

1. **common target needs**. All targets are ultimately nodes in a tree (see Figure 1). By virtue of this there exists states common to all targets (e.g., an identifier and a CMake target) and algorithms which are applicable to all targets (e.g., get/set target properties).

2. **built versus found**. Targets ultimately are treated different based on whether our build system needs to build the target or if it is already installed.

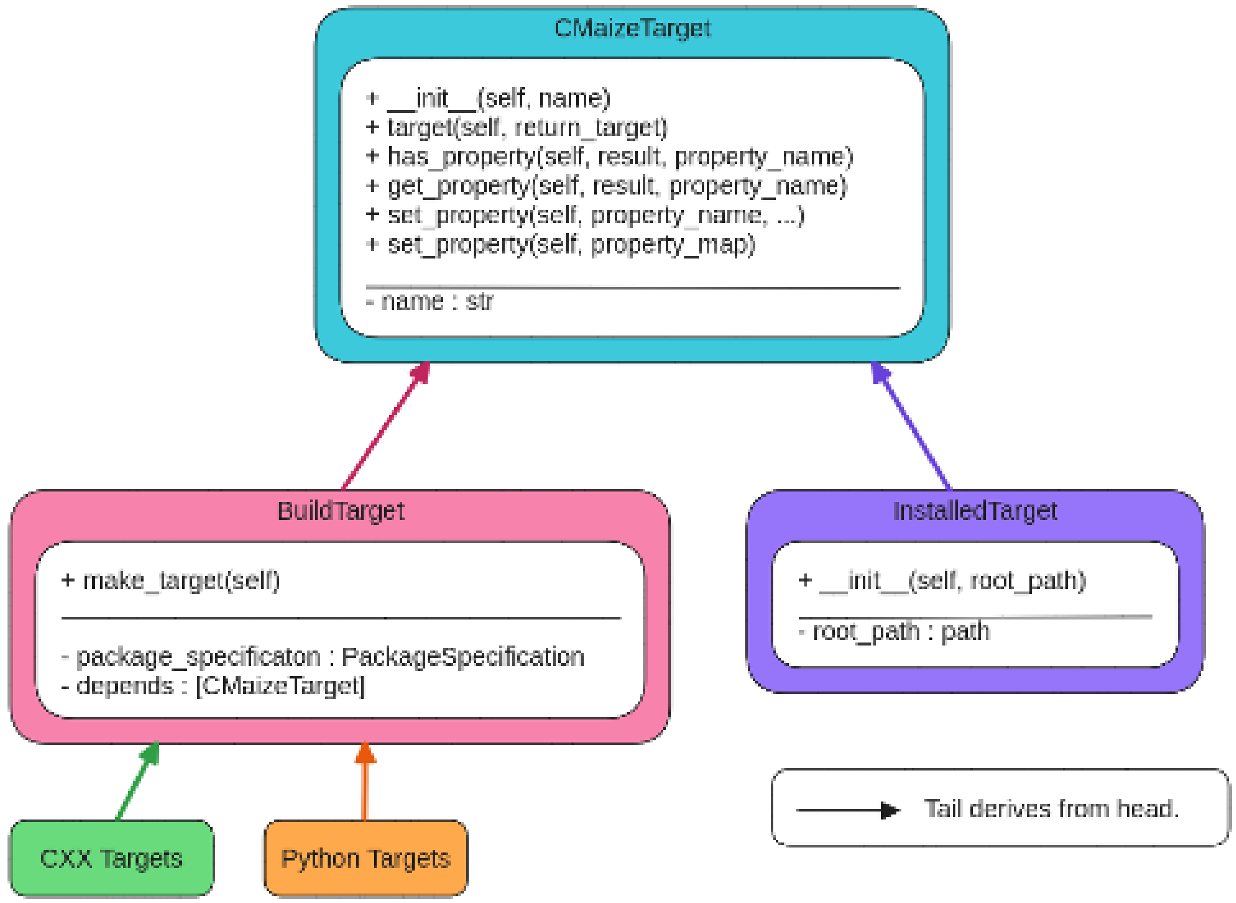

FIG. 10. Design of the language independent aspect of the `CMaizeTarget` class hierarchy. Arrows point from the derived class to the base class. Bullets denote access level of the member (i.e., "-" are private members and "+" are public). The "CXX Targets" and "Python Targets" are representative of where in the class hierarchy various languages tie in; as of this writing only C++ targets are implemented.

3. **target language**. The build process of a target depends on the programming language the target is written in.

4. **library versus executable**. While the terminology varies among programming languages, most programming languages have concepts akin to C++ libraries and executables. CMake requires slightly different interactions with library targets versus executable targets; the class hierarchy will bifurcate to implement interactions correctly.

The base of the `CMaizeTarget` class hierarchy consists of three language-independent classes. Figure 10 shows the design of these three classes. Stemming from the **common target needs** use case, we have factored common state out into the `CMaizeTarget` class. At its core, the `CMaizeTarget` class is tasked with wrapping the CMake target in a more user-friendly interface. The state of the `CMaizeTarget` class is the name of the wrapped CMake target. Methods of the `CMaizeTarget` class are dedicated to getting/setting properties of the target. To address the **built versus found** consideration, `CMaizeTarget` is subclassed by `BuildTarget` and `InstalledTarget`. When CMaize (or CMake) finds suitable variant of an installed package, the targets representing the package are already set up. In turn, the `InstalledTarget` class simply contains the install location (if pieces of the package are installed to different root directories, multiple `InstalledTarget` objects can be created). The `BuildTarget` class has two components of its state: its `PackageSpecification` and a list of targets it depends on (targets can be either `BuildTarget` or `InstalledTarget`).

The remainder of the `CMaizeTarget` class hierarchy contains specializations to specific programming languages (consideration **target language**). To illustrate this point, Figure 11 shows the APIs of the `CXXTarget`, `CXXExecutable`, `CXXLibrary`, and the `CXXInterfaceLibrary` classes. The `CXXTarget` class serves as code factorization for storing common state, such as the C++ standard (i.e., C++11, C++14, C++17, etc.) and the paths to the source files. The `CXXTarget` class also wraps the setting of the C++ target properties, e.g., the access level (whether or not the target can be linked to outside the project), the compiler options, paths to dependencies, and paths to the target's header files. Stemming from the **library versus executable** consideration, the `_create_target` method of the `CXXTarget` class abstracts away the differences in initializing different CMake targets; `CXXExecutable` and `CXXLibrary` respectively implement `_create_target` so that it creates an executable or library. Finally, `CXXInterfaceLibrary` further specializes the `CXXLibrary` class for header-only C++ libraries (which CMake terms "interface libraries"). Similar hierarchies for other programming languages can be added to CMaize.

D. PackageManager

Modern CMake-based build systems rely on CMake targets to convey package/dependency relations. Similarly, CMaize-based build systems rely on `CMaizeTarget` objects to convey package/dependency information. For the targets provided by a package, CMaize's user

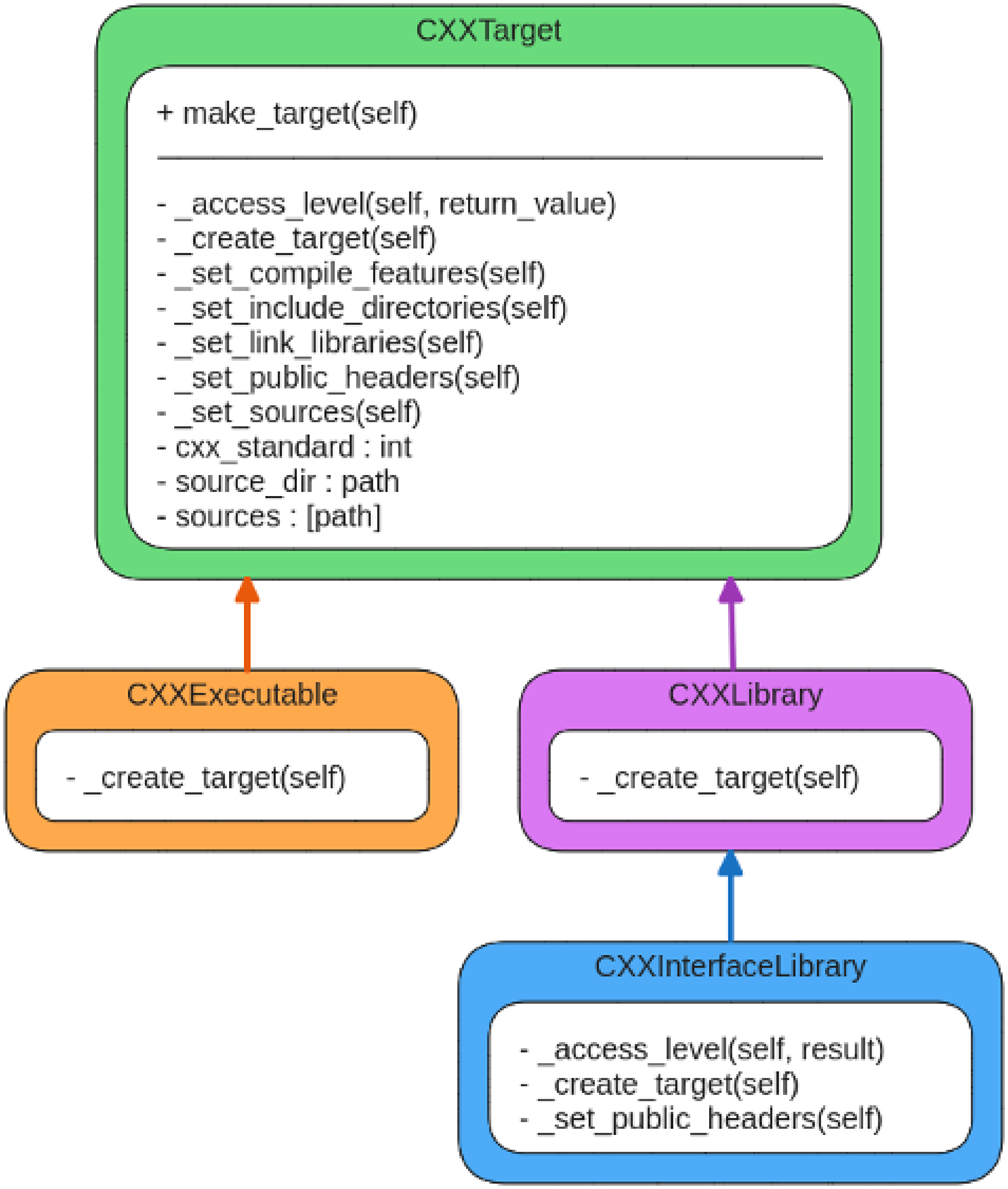

FIG. 11. Design of the classes responsible for extending the `CMaizeTarget` class hierarchy to C++. Arrows point from the derived class to the base class. Bullets denote access level of the member (i.e., "-" are private members and "+" are public). Methods in derived classes which have the same name and signature as a method in a base class override the base class method.

API (see Section III A) transparently facilitates the creation of the necessary `CMaizeTarget` objects. For targets which are provided by dependencies of the project, CMaize needs a mechanism to convert dependency specifications to `CMaizeTarget` objects. That mechanism is encapsulated in the `PackageManager` class hierarchy. The `PackageManager` class hierarchy is tasked with:

1. **finding packages**. Most packages can be installed for reuse. Assuming a suitable variant of a package already exists, the package manager should be able to locate the installed version and create a suitable `CMaizeTarget` object.

2. **building packages**. To be user-friendly, a build system should be able to build a suitable variant of a dependency, when such a variant can not be located.

3. **installing packages**. In order to be able to satisfy future requests for a variant of a package, the package manager will install packages it builds. This notably includes the package resulting from running the CMaize-based build system.

4. **CMake package management**. CMake ships with its own package management commands (i.e., `find_package` and `FetchContent`). Many existing CMake-based build systems heavily rely on CMake's package management commands and CMaize must be able to support it.

5. **extendable to other package managers**. Package management is a complicated task. CMaize does not want to "reinvent the wheel." Thus, it must be possible to extend CMaize's `PackageManager` class hierarchy to additional package managers (particularly those affiliated with other programming languages).

Figure 12 shows the design of the `PackageManager` class hierarchy. Following usual object-oriented design, the `PackageManager` class establishes the API that all package managers must implement. The methods of the common API come directly from the first three considerations (i.e., **finding packages**, **building packages**, and **installing packages**). In these methods, the input is labeled "specification" and is an object of the `PackageSpecification` type introduced in Section III B. In all cases, "target" is an object of type `CMaizeTarget`; however, `find_installed` and `get_package` return the target after

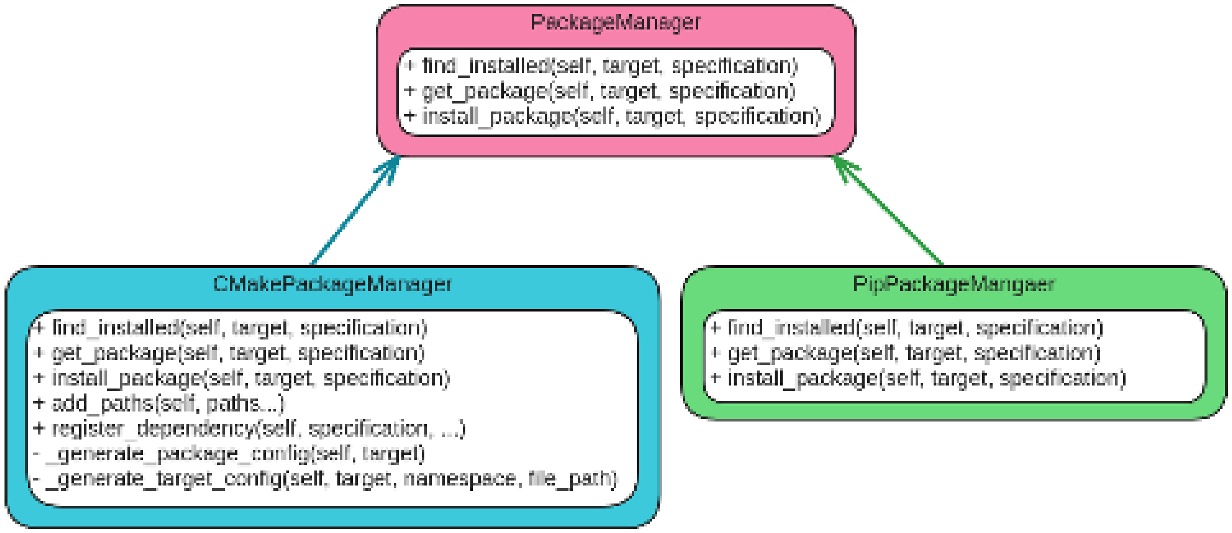

FIG. 12. Design of the `PackageManager` class hierarchy.

respectively finding and building the package, whereas `install_package` takes the target to install as an additional input.

CMaize currently ships with support for CMake's native package management functions (implemented by the `CMakePackageManager` class) and pip (implemented by the `PipPackageManager` class). `CMakePackageManager` is necessary to address the **CMake package management** consideration. The `CMakePackageManager` class implements `find_installed` and `get_package` by respectively deferring to CMake's native `find_package` function and CMake's native `FetchContent` module. The `install_package` method wraps the process of registering the target for installation (done through overloads of CMake's `install` function) and generating the resulting package configuration files. The `CMakePackageManager` class is ultimately a relatively primitive package manager. Full package manager support should come from wrapping existing C++ package managers, such as Conan[23] or Spack[26]. Nonetheless, even with support for full-fledged package managers, the `CMakePackageManager` class will be necessary for installing the project's package.

As mentioned in Section I, C++ libraries are increasingly incorporating Python support. To better support mixed C++/Python projects CMaize version 1.0.0 has limited pip support via the `PipPackageManager` class. The `PipPackageManager` class takes as input a Python interpreter and will use the pip executable associated with that interpreter. The `find_installed`/`install_package` members respectively are overridden to rely on pip's `list/install` commands. As of version 1.0.0 the `install_package` command can only install dependencies, not the package, and `get_package` is a no-op (i.e., we assume that the Python dependency does not need to be built before it can be installed). It is our intent to expand on CMaize's pip capabilities in future releases.

E. Other design considerations

The previous subsections detailed how CMaize's source code is structured in order to address the considerations raised at the beginning of this section. In designing CMaize, we also have made a number of decisions which we feel facilitate the sustainability of CMaize. First, CMaize is open source and licensed under the very permissive Apache 2.0 license[82]. In turn, users are free to modify, adapt, and redistribute CMaize as they see fit, so long as they attribute proper credit to the original authors. The source code for CMaize is version controlled with `git`[83] and hosted on GitHub (`https://github.com/CMakePP/CMaize`) as part of the CMakePP Organization[84]. The CMakePP Organization welcomes contributions and feedback from anyone in the community, as long as they sign the contributor license agreement (CLA) and adhere to the organization's code of conduct.

Even though CMaize is a build tool written in the CMake programming language, we strive to treat CMaize like any other programming project. In particular, contributions to CMaize must adhere to a pull request process relying on automated continuous integration/continuous development (CI/CD) pipelines. These pipelines automatically check that CMaize works with multiple versions of CMake, documentation builds, and that contributors have signed the CLA. Once the pull request has been reviewed and approved by a CMaize code maintainer, the pipeline automates the process of tagging the contribution (following semantic versioning[85]) and deploying updated documentation. Key to the CI/CD pipelines are CMakeTest[86] and CMinx[87] which respectively facilitate testing and documenting code written in CMake.

IV. INITIAL APPLICATIONS

A. Hello World

Figure 6 in Section II demonstrated the simplest CMaize-based build system. Recall that this is the build system for a single source file, no dependency, C++ executable. Like CMake-based build systems, CMaize-based build systems are also defined in `CMakeLists.txt` files. For concreteness, the CMaize-based build system shown in Figure 6 assumes that the `hello_world.cpp` source file (contents shown in Figure 3) resides in the same directory as the `CMakeLists.txt` file implementing the build system. For the purposes of simplifying this example, we assume that there is a CMaize installation already on the system, and that the path to that CMaize installation is included in CMake's `CMAKE_MODULE_PATH` variable. The next example will show how to avoid this assumption.

The first two lines of Figure 6 are required by CMake for warning-free execution. Line number 4 imports the CMaize module, which in turn brings CMaize's functionality into scope. Line 6 of Figure 6 declares and defines the `hello_world` executable. For this simple scenario, defining the executable requires telling CMaize in which directory the source code resides. Finally line 7 tells CMaize to create a new package from the `hello_world` executable. Of particular note, CMaize will automatically create the packaging files (Figure 5), assume that the executable should be installed in the operating system appropriate runtime location (e.g., `${CMAKE_INSTALL_PREFIX}/hello_world/bin` on Unix-like operating systems), and assume that the packaging files will be installed in the appropriate CMake packaging file location (e.g., `${CMAKE_INSTALL_PREFIX}/hello_world/lib/cmake` on -nix operating systems).

B. PluginPlay

The previous "Hello World" example demonstrated the basics of writing a CMaize-based build system, but ultimately portrays an unrealistically simple build system. We anticipate that most software packages designed for inter-package modularity will not be single-file, no-dependency executables. To showcase a more realistic example we consider the build system of a CMaize early adopter: PluginPlay[7].

The `CMakeLists.txt` file defining a modified build system for version 1.0.19 of PluginPlay

```
1  cmake_minimum_required(VERSION 3.14)
2  project(pluginplay VERSION 1.0.18 LANGUAGES CXX)
3
4  include(FetchContent)
5  FetchContent_Declare(
6      cmaize
7      GIT_REPOSITORY "https://github.com/CMakePP/CMakePackagingProject"
8      GIT_TAG 1.0.0
9  )
10 FetchContent_MakeAvailable(cmaize)
11
12 include(cmaize/cmaize)
13
14 ### Options ###
15 cmaize_option_list(
16     BUILD_TESTING OFF "Should we build unit tests?"
17     BUILD_PYBIND11_PYBINDINGS OFF "Build Pybind11 Python bindings?"
18     BUILD_ROCKSDB OFF "Enable RocksDB backend of the cache?"
19 )
20
21 ### Dependendencies ###
22 cmaize_find_or_build_dependency(
23     utilities
24     URL "github.com/NWChemEx/utilities"
25     VERSION 0.1.14
26     BUILD_TARGET utilities
27     FIND_TARGET nwx::utilities
28 )
29
30 cmaize_find_or_build_dependency(
31     libfort
32     URL "github.com/seleznevae/libfort"
33     BUILD_TARGET fort
34     FIND_TARGET libfort::fort
35 )
36
37 cmaize_find_or_build_dependency(
38     parallelzone
39     URL "github.com/NWChemEx/ParallelZone"
40     BUILD_TARGET parallelzone
41     FIND_TARGET nwx::parallelzone
42 )
```

FIG. 13. Part one of the `CMakeLists.txt` for PluginPlay's build system.

is split across Figure 13, Figure 14, and Figure 15. The file has been modified for clarity. For comparison, the full build system for version 1.0.19 of PluginPlay can be seen the PluginPlay repository[88]. The full file is of a similar size (i.e., 203 lines). Thus even though our presentation is simplified, it is still representative of the actual build system's complexity.

Beginning with Figure 13, note lines 4-12. These lines are all that are needed to transparently download and include CMaize from within your build system. With these lines the build system fully encapsulates the fact that it relies on CMaize. Unfortunately lines 4-12 are necessary boilerplate because CMaize is not an internal CMake module and must

```
1  cmaize_find_dependency(
2      boost_container
3      URL "github.com/boostorg/container"
4      FIND_TARGET Boost::container
5  )
6
7  ## Optional Dependencies ##
8  cmaize_find_optional_dependency(
9      RocksDB
10     BUILD_ROCKSDB
11     URL "github.com/facebook/rocksdb"
12     FIND_TARGET RocksDB::rocksdb-shared
13 )
14
15 set(pluginplay_depends utilities parallelzone libfort boost_container RocksDB)
16
17 # As of 1.0.0 CMaize does not support multiple build or find targets. This will
18 # be fixed in a future feature release. For now we handle the
19 if("${BUILD_PYBIND11_PYBINDINGS}")
20     cmaize_find_or_build_dependency(
21         pybind11
22         URL "github.com/pybind/pybind11"
23         BUILD_TARGET pybind11_headers
24         FIND_TARGET pybind11::embed
25         CMAKE_ARGS PYBIND11_INSTALL=ON
26                    PYBIND11_FINDPYTHON=ON
27     )
28     list(
29         APPEND pluginplay_depends
30         pybind11_headers pybind11::embed Python::Python
31     )
32 endif()
33
34 # Work out full paths to the project's include/source dirs
35 set(project_inc_dir "${CMAKE_CURRENT_LIST_DIR}/include/pluginplay")
36 set(project_src_dir "${CMAKE_CURRENT_LIST_DIR}/src/pluginplay")
37
38 cmaize_add_library(
39     pluginplay
40     SOURCE_DIR "${project_src_dir}"
41     INCLUDE_DIRS "${project_inc_dir}"
42     DEPENDS "${pluginplay_depends}"
43 )
44
45 include(nwx_pybind11) # Thin wrapper around cmaize_add_library
46 nwx_add_pybind11_module(
47     py_pluginplay
48     SOURCE_DIR "${CMAKE_CURRENT_LIST_DIR}/src/python"
49     DEPENDS pluginplay
50 )
```

FIG. 14. Part two of the `CMakeLists.txt` for PluginPlay's build system.

be obtained before it can be used. This should be compared with say the `FetchContent` module which, as line 4 shows, simply needs to be included before it can be used.

The `cmaize_option_list` function shown on lines 15-19 defines configuration options for PluginPlay and registers them with CMaize. When users build PluginPlay they are free to

```
if("${BUILD_TESTING}")
    set(cxx_test_dir "${CMAKE_CURRENT_LIST_DIR}/tests/cxx")
    set(python_test_dir "${CMAKE_CURRENT_LIST_DIR}/tests/python")
    set(tests_src_dir "${cxx_test_dir}/unit_tests/pluginplay")
    set(examples_src_dir "${cxx_test_dir}/doc_snippets")

    ### C++ Testing ###
    cmaize_find_or_build_dependency(
        Catch2
        URL "github.com/catchorg/Catch2"
        BUILD_TARGET Catch2
        FIND_TARGET Catch2::Catch2
    )
    cmaize_add_tests(
        test_unit_pluginplay
        SOURCE_DIR ${tests_src_dir}
        INCLUDE_DIRS "${CMAKE_CURRENT_LIST_DIR}/src/pluginplay"
        DEPENDS Catch2::Catch2 pluginplay
    )

    ### Python Tests ###
    cmaize_find_or_build_dependency(
        tox
        PACKAGE_MANAGER pip
    )

    nwx_tox_test(
        py_pluginplay ${python_test_dir}/unit_tests
        SUBMODULES parallelzone
    )
endif()

cmaize_add_package(pluginplay NAMESPACE nwx::)
```

FIG. 15. Part three of the `CMakeLists.txt` for PluginPlay's build system.

specify values for these configuration options to control the variant of PluginPlay which is built. `cmaize_option_list` automatically ensures that the user's values take precedence over the defaults as well as ensuring that configuration options are recorded in the `CMaizeProject` object (which is made under the hood). In the case of PluginPlay the configuration options respectively determine whether or not: the tests should be built, Python bindings should be built, and the RocksDB[89] database should be used for internal cacheing, with all options defaulting to `OFF`.

For required dependencies CMaize defines two functions, `cmaize_find_dependency` and `cmaize_find_or_build_dependency`. Examples of using these functions are shown on lines 22-47 (note that lines 43-47 are in Figure 14). As the names imply, the difference between the two functions is that `cmaize_find_or_build_dependency` will build the dependency if it is not found whereas `cmaize_find_dependency` will error if an already installed version is not found. The `BUILD_TARGET` and `FIND_TARGET` inputs are perhaps the least straightforward inputs and are used to tell CMaize what CMake targets it should be finding. There are two input fields because, as shown, often times the target's name depends on whether we are building a dependency via `FetchContent` or if we are finding the target via `find_package`. As for PluginPlay's dependencies: Utilities serves as an extension to the C++ standard library for the NWChemEx project, Libfort[90] facilitates the creation of formatted tables (used by PluginPlay to automatically generate documentation for modules), ParallelZone is NWChemEx's parallel runtime, and internally PluginPlay relies on Boost's container library[91].

Figure 14, lines 50-74 define two optional dependencies. Optional dependencies usually are enabled/disabled by a configuration option (e.g., RocksDB is enabled by setting `ENABLE_ROCKSDB` to true). The `cmaize_find_optional_dependency` function takes the name of the dependency, the flag which enables/disables it, and the specification of the dependency. The `cmaize_find_optional_dependency` function encapsulates the process of conditionally finding the dependency, when enabled, or creating a "do-nothing" target when disabled; the do-nothing target has the same name as the real target and is used everywhere the real target is used to avoid having to propagate the conditional logic.

CMaize is admittedly not as feature complete as we would like. For example, at the time of this writing, due to an oversight, the logic underlying the finding of dependencies currently does not support locating multiple targets. This is why lines 61-74 do not use the `cmaize_find_or_build_optional_dependency` function, but instead manually implement the optional logical. Nonetheless, this highlights an important feature of CMaize, its backwards compatibility with CMake. More specifically build system maintainers are able to customize the build system logic to their liking by using the CMake programming language.

The remainder of Figure 14 defines the PluginPlay library and its Python bindings. As mentioned in Section I, PluginPlay is an IOC framework for writing scientific software. Features are added to the framework by writing plugins. The plugins are dynamically loaded into the framework by Python, i.e., the plugins are either C++ libraries with Python bindings or they are pure Python modules. The `nwx_add_pybind11_module` is a thin wrapper around `cmaize_add_library` which is used by the NWChemEx team to facilitate creating Python modules for use with PluginPlay. In particular the `nwx_add_pybind11_module` function ensures the resulting library is named correctly (no “lib” prefix) and is installed to the PluginPlay plugin directory. This emphasizes another important feature of CMaize, namely that users can leverage CMaize to create their own project-specific functionality.

Figure 15 contains the remainder of the build system. Lines 93-123 include a subset of PluginPlay's testing (the actual repository contains several additional tests which have been removed for clarity). For convenience, CMaize defines the `cmaize_add_tests` function which wraps the process of creating a C++ executable and registering it with CMake's `CTest` program. Up to this point, all dependencies have gone through CMaize's default `CMakePackageManager`. Lines 114-117 demonstrate the use of the `PipPackageManager` by relying on pip to find the `Tox` package[92] (facilitates testing Python builds in different environments). Conceptually the only difference to previous `cmaize_find_or_build_dependency` calls is that we have to tell CMaize to use pip (n.b., setting `PACKAGE_MANAGER` to “cmake” would restore the default behavior). At present, CMaize can not automatically register Python tests with `CTest` which is why the NWChemEx team wrote the `nwx_tox_text` function. We expect a future release of CMaize to rectify this deficiency. Finally line 125 declares that the `pluginplay` target should be packaged for installation and that the name of the resulting installed target should be prefixed with `nwx::`.

To summarize, in this subsection we have shown the CMaize-based build system for PluginPlay, an actual production-level scientific framework. The build system for PluginPlay is non-trivial and consists of two programming languages, four required dependencies, two optional dependencies, two testing dependencies, and several test suites. Arguably the most complicated part of the CMaize-based build system is needing to identify the find and build targets for the dependencies. As a slight aside, a planned CMaize feature will auto-detect when targets have been added to the build system, making user-specification optional. In our opinion the largest benefit of using CMaize is in simplifying the packaging step. In our experience, packaging a CMake-based build system is an error-prone, laborious effort. Nonetheless, with a single line of code (i.e., line 125 of Figure 15), CMaize is able to package PluginPlay for reuse. This is because CMaize relies on a combination of the already supplied target information and reasonable, but overrideable, defaults for install locations. Even when CMaize does not provide the exact functionality a user wants, CMaize is usually an excellent starting point (see for example lines 61-74 of Figure 14). Finally, we want to stress the stability of the CMaize APIs. Best practices for CMake-based build systems have evolved quite a bit over the last decade or so. CMaize encapsulates those best practices helping ensure build systems remain backwards compatible.

V. SUMMARY AND OUTLOOK

The increasing focus on developing and maintaining modular scientific software is already a boon for developer productivity and additional inter-package modularity efforts are only likely to further increase productivity. Unfortunately, since many scientific software packages are written in compiled languages, maintaining a software stack which heavily relies on the use of independently developed modules requires maintaining reliable and robust build systems for not just the package, but for the dependencies as well. At present, most (compiled) scientific software is written in C/C++ or Fortran. CMake is the *de facto* build system tool for C/C++, and also supports Fortran. Unfortunately, CMake-based build systems tend to be: overly verbose, unstable with respect to CMake feature releases, poor at managing dependencies, and difficult to extend to additional programming languages and/or tools. To be fair, CMake inherited many aspects of these problems from the fact that C/C++ has not standardized the build process. Nonetheless, with build system support often an afterthought in many scientific packages, a high barrier to writing reliable/robust build systems threatens to dampen the rapid adoption of inter-package modular software. Unable to find a suitable solution to these shortcomings problems, we opted to write CMaize.

CMaize is written in the CMake programming language and can be leveraged by existing CMake-based build systems through CMake's module system. CMaize-based build systems work seamlessly with existing workflows and tools which were designed for CMake-based build systems. While CMaize strives to distill writing a build system down to specifying the bare-minimum information required, this is admittedly not always possible; however, since CMaize is "backwards compatible" with CMake, build system developers can still use traditional CMake commands to fine-tune the build system. The key to CMaize's succinct syntax is that any information which can be computed, or for which there is a reasonable default value, is optional. Since CMaize's API requires minimal information, it is easier for the API to remain stable, even as the CMake feature sets powering them change. With support for existing package managers (currently limited to pip, but additional C++ package manager support is planned) build system developers can delegate dependency management to battle-tested dependency managers. Finally, the functional, CMake-like, user API is actually implemented using an object-oriented extension[79] to the CMake programming language. In turn, the source code of CMaize can easily be extended, via inheritance, to additional programming languages and build tools.

The present article accompanies the 1.0.0 release of CMaize. Perhaps unsurprisingly, adoption of CMaize is currently at an early stage and largely limited to the NWChemEx community. CMaize is developed to be a useful, general build system tool and, given the relative simplicity of using CMaize compared to CMake, it is our hope that more projects will consider rewriting their build systems with CMaize. As already mentioned, future versions of CMaize are expected to increase support for additional programming languages and build tools. We are also currently pursing mechanisms for leveraging the `PackageManager` class to facilitate creating packages compatible with existing package managers, e.g., it is our hope that CMaize will be able to automatically generate the files required to deploy a package to PyPI[22] (n.b., this is not unprecedented as other projects like pybind11[16] have written CMake modules for similar purposes). Ultimately, we see CMaize as an integral part of a modular scientific software ecosystem and hope that CMaize will help accelerate developer adoption/contribution of/to this scientific ecosystem. Due to the many possibilities for extending CMaize, we welcome contributors and contributions to the software.

ACKNOWLEDGMENTS

The authors would like to additionally thank Branden Butler, Blake Mulnix, and David Williams-Young for small contributions to CMaize.

This research was primarily supported 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. Additional funding to ZC was provided by an Iowa State University Liberal Arts & Sciences Dean's Professor Chair to TLW.

REFERENCES

[1]A. I. Krylov, J. M. Herbert, F. Furche, M. Head-Gordon, P. J. Knowles, R. Lindh, F. R. Manby, P. Pulay, C.-K. Skylaris, and H.-J. Werner, "What is the price of open-source

software?" The Journal of Physical Chemistry Letters **6**, 2751–2754 (2015).

[2]M. J. T. Oliveira, N. Papior, Y. Pouillon, V. Blum, E. Artacho, D. Caliste, F. Corsetti, S. de Gironcoli, A. M. Elena, A. García, V. M. García-Suárez, L. Genovese, W. P. Huhn, G. Huhs, S. Kokott, E. Küçükbenli, A. H. Larsen, A. Lazzaro, I. V. Lebedeva, Y. Li, D. López-Durán, P. López-Tarifa, M. Lüders, M. A. L. Marques, J. Minar, S. Mohr, A. A. Mostofi, A. O'Cais, M. C. Payne, T. Ruh, D. G. A. Smith, J. M. Soler, D. A. Strubbe, N. Tancogne-Dejean, D. Tildesley, M. Torrent, and V. W.-z. Yu, "The CECAM electronic structure library and the modular software development paradigm," The Journal of Chemical Physics **153** (2020), 10.1063/5.0012901.

[3]S. Lehtola, "A call to arms: Making the case for more reusable libraries," The Journal of Chemical Physics **159** (2023), 10.1063/5.0175165.

[4]R. Di Felice, M. L. Mayes, R. M. Richard, D. B. Williams-Young, G. K.-L. Chan, W. A. de Jong, N. Govind, M. Head-Gordon, M. R. Hermes, K. Kowalski, X. Li, H. Lischka, K. T. Mueller, E. Mutlu, A. M. N. Niklasson, M. R. Pederson, B. Peng, R. Shepard, E. F. Valeev, M. van Schilfgaarde, B. Vlaisavljevich, T. L. Windus, S. S. Xantheas, X. Zhang, and P. M. Zimmerman, "A perspective on sustainable computational chemistry software development and integration," Journal of Chemical Theory and Computation **19**, 7056–7076 (2023).

[5]A. Krylov, T. L. Windus, T. Barnes, E. Marin-Rimoldi, J. A. Nash, B. Pritchard, D. G. A. Smith, D. Altarawy, P. Saxe, C. Clementi, T. D. Crawford, R. J. Harrison, S. Jha, V. S. Pande, and T. Head-Gordon, "Perspective: Computational chemistry software and its advancement as illustrated through three grand challenge cases for molecular science," The Journal of Chemical Physics **149** (2018), 10.1063/1.5052551.

[6]C. D. Sherrill, D. E. Manolopoulos, T. J. Martínez, and A. Michaelides, "Electronic structure software," The Journal of Chemical Physics **153** (2020), 10.1063/5.0023185.

[7]R. M. Richard, K. Keipert, J. Waldrop, M. Keçeli, D. Williams-Young, R. Bair, J. Boschen, Z. Crandall, K. Gasperich, Q. I. Mahmud, A. Panyala, E. Valeev, H. van Dam, W. A. de Jong, and T. L. Windus, "PluginPlay: Enabling exascale scientific software one module at a time," The Journal of Chemical Physics **158**, 184801 (2023).

[8]E. F. Valeev, "Libint: A library for the evaluation of molecular integrals of many-body operators over gaussian functions," http://libint.valeyev.net/ (2022), version 2.8.0.

[9]S. Lehtola, C. Steigemann, M. J. Oliveira, and M. A. Marques, "Recent developments in

libxc — a comprehensive library of functionals for density functional theory," SoftwareX **7**, 1–5 (2018).

[10]A. Petrone, D. B. Williams-Young, S. Sun, T. F. Stetina, and X. Li, "An efficient implementation of two-component relativistic density functional theory with torque-free auxiliary variables," The European Physical Journal B **91** (2018), 10.1140/epjb/e2018-90170-1.

[11]D. B. Williams-Young, W. A. de Jong, H. J. J. van Dam, and C. Yang, "On the efficient evaluation of the exchange correlation potential on graphics processing unit clusters," Frontiers in Chemistry **8** (2020), 10.3389/fchem.2020.581058.

[12]D. B. Williams-Young, A. Bagusetty, W. A. de Jong, D. Doerfler, H. J. van Dam, Á. Vázquez-Mayagoitia, T. L. Windus, and C. Yang, "Achieving performance portability in Gaussian basis set density functional theory on accelerator based architectures in NWChemEx," Parallel Computing **108**, 102829 (2021).

[13]D. B. Williams-Young, A. Asadchev, D. T. Popovici, D. Clark, J. Waldrop, T. L. Windus, E. F. Valeev, and W. A. de Jong, "Distributed memory, GPU accelerated Fock construction for hybrid, Gaussian basis density functional theory," The Journal of Chemical Physics **158** (2023), 10.1063/5.0151070.

[14]R. Di Remigio, A. H. Steindal, K. Mozgawa, V. Weijo, H. Cao, and L. Frediani, "PCMSolver: An open-source library for solvation modeling," International Journal of Quantum Chemistry **119**, e25685 (2019).

[15]D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. Mclnnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou, "A component architecture for high-performance scientific computing," The International Journal of High Performance Computing Applications **20**, 163–202 (2006).

[16]W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between C++11 and Python," (2017), https://github.com/pybind/pybind11.

[17]W. T. Lavrijsen and A. Dutta, *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)* (IEEE, 2016).

[18]"CMake," `https://cmake.org/` (n.d.), accessed: 11-2-2023.

[19]N. J. Smith and T. Kluyver, "Pep 517 - a build-system independent format for source trees," `https://peps.python.org/pep-0517/` (2015), accessed: 12-19-2023.

[20] B. Cannon, N. J. Smith, and D. Stufft, "Pep 518 - specifying minimum build system requirements for python projects," https://peps.python.org/pep-0518/ (2016), accessed: 12-19-2023.

[21] The pip developers, "pip," https://pip.pypa.io/en/stable/ (n.d.), accessed: 12-19-2023.

[22] Python Software Foundation, "PyPI: The python package index," https://pypi.org/ (n.d.), accessed: 12-19-2023.

[23] "Conan 2.0: C and C++ open-source package manager," https://conan.io/ (n.d.), accessed: 12-19-2023.

[24] Microsoft, "vcpkg," https://github.com/Microsoft/vcpkg (n.d.), accessed: 12-19-2023.

[25] "Hunter," https://github.com/cpp-pm/hunter (n.d.), accessed: 11-17-2023.

[26] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack Package Manager: Bringing Order to HPC Software Chaos," (Austin, Texas, US, 2015).

[27] "CPM," https://github.com/iauns/cpm (n.d.), accessed: 11-2-2023.

[28] R. Bast and R. D. Remigio, *CMake cookbook* (Packt Publishing Ltd., Birmingham, UK, 2018).

[29] "An introduction to modern CMake," https://cliutils.gitlab.io/modern-cmake/ (n.d.), accessed: 11-2-2023.

[30] "Importing and exporting guide," https://cmake.org/cmake/help/latest/guide/importing-exporting/index.html (n.d.), accessed: 11-9-2023.

[31] "It's time to do CMake right," https://pabloariasal.github.io/2018/02/19/its-time-to-do-cmake-right/ (2018), accessed: 11-2-2023.

[32] "Samples fo learning modern CMake," https://github.com/toeb/moderncmake (n.d.), accessed: 11-2-2023.

[33] "cmake_template," https://github.com/cpp-best-practices/cmake_template (n.d.), accessed: 11-1-2023.

[34] "cpp-boilerplate," https://github.com/Lectem/cpp-boilerplate (n.d.), accessed: 11-1-2023.

[35] "cppbase," https://github.com/kartikkumar/cppbase (n.d.), accessed: 11-1-2023.

[36] "cpp-lib-template," https://github.com/pananton/cpp-lib-template (n.d.), accessed: 11-2-2023.

[37] "Minimal CMake Example," `https://github.com/krux02/minimal_cmake_example` (n.d.), accessed: 11-2-2023.

[38] "moderncpp-project-template," `https://github.com/madduci/moderncpp-project-template` (n.d.), accessed: 11-1-2023.

[39] "pacakge-example," `https://github.com/forexample/package-example` (n.d.), accessed: 11-1-2023.

[40] "JAWS," `https://github.com/DevSolar/jaws` (n.d.), accessed: 11-1-2023.

[41] A. Hunt and D. Thomas, *The Pragmatic Programmer* (Addison-Wesley, 1999).

[42] R. Maynard, "CMake 3.0.0 available for download," `https://www.kitware.com/cmake-3-0-0-available-for-download/` (2014), accessed: 12-19-2023.

[43] J. Parent, "CMake 3.28.1 available for download," `https://www.kitware.com/cmake-3-28-1-available-for-download/` (2023), accessed: 12-19-2023.

[44] "Boost C++ libraries," `https://www.boost.org/` (n.d.), accessed: 12-19-2023.

[45] R. Maynard, "CMake 3.5.0 available for download," `https://www.kitware.com/cmake-3-5-0-available-for-download/` (2016), accessed: 12-19-2023.

[46] R. Maynard, "CMake 3.11.0 available for download," `https://www.kitware.com/cmake-3-11-0-available-for-download/` (2018), accessed: 12-19-2023.

[47] M. D. Hanwell, "CMake superbuilds and Git submodules," `https://www.kitware.com/cmake-superbuilds-git-submodules/` (2017), accessed: 11-2-2023.

[48] "find_package," `https://cmake.org/cmake/help/latest/command/find_package.html` (n.d.), accessed: 12-20-2023.

[49] "FetchContent," `https://cmake.org/cmake/help/latest/module/FetchContent.html` (n.d.), accessed: 12-20-2023.

[50] M. Jang, *Linux Annoyances for Geeks* (O'Riley Media, Inc., 2006).

[51] S. Lehtola and A. J. Karttunen, "Free and open source software for computational chemistry education," WIREs Computational Molecular Science **12** (2022), 10.1002/wcms.1610.

[52] M. Kowalewski and P. Seeber, "Sustainable packaging of quantum chemistry software with the nix package manager," International Journal of Quantum Chemistry **122** (2022), https://onlinelibrary.wiley.com/doi/10.1002/qua.26872.

[53] "Aptitude," `https://wiki.debian.org/Aptitude` (n.d.), accessed: 12-20-2023.

[54] "Yellowdog updater modified," `http://yum.baseurl.org/` (n.d.), accessed: 2-13-2024.

[55] "Dandified yum," `https://rpm-software-management.github.io/` (n.d.), accessed:

2-13-2024.

[56] "Homebrew - the missing package manager for macOS (or Linux)," https://brew.sh/ (n.d.), accessed: 12-20-2023.

[57] J. Parent, "CMake 3.24.0 is available for download," https://www.kitware.com/cmake-3-24-0-is-available-for-download/ (2022), accessed: 12-19-2023.

[58] Kitware, "CMake 2.2 to be released in September," https://www.kitware.com/cmake-2-2-to-be-released-in-september/ (2005), accessed: 12-20-2023.

[59] R. Maynard, "CMake 3.8.0 available for download," https://www.kitware.com/cmake-3-8-0-available-for-download/ (2017), accessed: 11-17-2023.

[60] R. Maynard, "CMake 3.12.0 available for download," https://www.kitware.com/cmake-3-12-0-available-for-download/ (2018), accessed: 11-17-2023.

[61] "IDE integration guide," https://cmake.org/cmake/help/latest/guide/ide-integration/index.html#ides-with-cmake-integration (n.d.), accessed: 12-20-2023.

[62] JetBrains, "CLion: A cross-platform IDE for C and C++," https://www.jetbrains.com/clion/ (n.d.), accessed: 12-20-2023.

[63] "Vim - the ubiquitous text editor," https://www.vim.org/ (n.d.), accessed: 12-20-2023.

[64] SCons Foundation, "SCons: A software construction tool," https://www.scons.org/ (n.d.), accessed: 12-20-2023.

[65] "Gradle build tool," https://gradle.org/ (n.d.), accessed: 12-20-2023.

[66] "Ninja," https://ninja-build.org/ (n.d.), accessed: 12-20-2023.

[67] "Bazel," https://bazel.build/ (n.d.), accessed: 12-20-2023.

[68] R. Bast, R. Di Remigio, and J. Juselius, "Autocmake," (2020).

[69] "cmake-init," https://github.com/friendlyanon/cmake-init (n.d), accessed: 11-2-2023.

[70] "BLT," https://github.com/llnl/blt (n.d.), accessed: 11-1-2023.

[71] "CMake++," https://github.com/toeb/cmakepp (n.d.), accessed: 11-1-2023.

[72] "Izzy's eXtension Modules," https://github.com/ixm-one/legacy (n.d.), accessed: 11-2-2023.

[73] "project_options," https://github.com/aminya/project_options (n.d.), accessed: 11-2-2023.

[74] "CPM.cmake," https://github.com/cpm-cmake/CPM.cmake (n.d.), accessed: 11-2-2023.

[75] "Cinch," https://github.com/laristra/cinch (n.d.), accessed: 11-1-2023.

[76] "cmake-get," https://github.com/pfultz2/cmake-get (n.d.), accessed: 11-2-2023.

[77] "Cmakebuild," https://github.com/NWChemEx-Project/CMakeBuild (n.d.), accessed: 11-20-2023.

[78] "Developing CMaize — CMaize 1.0.0 documentation," https://cmakepp.github.io/CMaize/developer (n.d.), accessed: 12-06-2023.

[79] Z. Crandall, B. Mulnix, B. Butler, T. L. Windus, and R. M. Richard, "CMakePPLang: An object-oriented extension toCMake," Journal of Open Source Software **8**, 5711 (2023).

[80] "CMaize," https://cmakepp.github.io/CMaize/ (n.d.), accessed: 12-20-2023.

[81] "add_custom_command," https://cmake.org/cmake/help/latest/command/add_custom_command.html (n.d.), accessed: 11-21-2023.

[82] "Apache License, Version 2.0," https://www.apache.org/licenses/LICENSE-2.0 (2004), accessed: 12-05-2023.

[83] L. Torvalds and J. Hamano, "Git," https://git.kernel.org/pub/scm/git/git.git (2023), accessed: 12-5-2023.

[84] "CMakePP," https://github.com/CMakePP (n.d.), accessed: 12-06-2023.

[85] T. Preston-Werner, "Semantic Versioning 2.0.0," https://semver.org/ (n.d.), accessed: 12-05-2023.

[86] R. Richard and B. Butler, "CMakeTest," https://github.com/CMakePP/CMakeTest (2023), accessed: 12-06-2023.

[87] B. Butler and R. M. Richard, "CMinx: A CMake documentation generator," Journal of Open Source Software **7**, 4680 (2022).

[88] "PluginPlay," https://github.com/NWChemEx/PluginPlay (n.d.), accessed: 12-21-2023.

[89] Meta Platforms, Inc., "RocksDB: A persistent key-value store for fast storage environments," https://rocksdb.org/ (2022), accessed: 12-27-2023.

[90] S. Anton, "libfort (Library to create FORmatted Tables)," https://github.com/seleznevae/libfort (2020), accessed: 12-27-2023.

[91] I. Gaztanaga, "Boost.Container," https://www.boost.org/doc/libs/1_84_0/doc/html/container.html (2018), accessed: 12-27-2023.

[92] "tox - automation project," https://tox.wiki/en/4.11.4/ (n.d.), accessed: 12-27-2023.