Document Number: P2990R0
Date: 2023-08-29
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

C++ Modules Roadmap

Abstract

This paper presents a proposal for a roadmap for ecosystem support for C++ Modules and the work that is required in the Tooling Study Group (SG15) to enable that to happen. There are a number of existing gaps in terms of what the tooling ecosystem needs to support in order for C++ Modules to become viable in most production use cases. This paper proposes a list of priorities for the Study Group, inviting contributions from implementers to enable wider adoption of the language feature. The goal of this paper is to serve as a call to action to resolve the remaining issues related to the specification for tooling interoperability.

1. Introduction

C++ Modules has been a controversial topic among implementers, particularly when you step beyond the compiler itself. A number of papers in 2019 and 2020 raised concerns about the implementability of the language feature.

There is now consensus in SG15 that we can achieve complete production-ready support for C++ modules in our tooling ecosystem. However, there are many steps that must be taken to get there. There are some initial reports of adoption in well-controlled environments, but if widely used libraries switched to a modules-only interface today, it would still be significantly disruptive to the ecosystem in general.

The challenge, therefore, is in creating consensus and driving implementation support to get us to a point where C++ modules can be considered as supported as any other code using the C++20 standard.

2. Current State of Support for C++ Modules

2.1. Compilers

All major compilers have support for producing a BMI from a module translation unit or importable header and to process a translation unit that performs an import, given that the path to the BMI is provided.

Dependency scanning for named modules is currently supported by all major compilers (as of this writing, GCC only offers that support in trunk, not in any released version). The output format for the module dependencies is de-facto standardized using the format specified in P1689R4[1].

However, dependency scanning for header units needs to perform an emulation of the import process since the state of the preprocessor is different from what source inclusion would do. It also needs to be aware that the header unit needs to be parsed without any Local Preprocessor Arguments. Currently, this is not supported by any compiler.

There are other areas that need work to refine how the tooling is meant to implement the language specification. For example, the way that module initialization is being handled in Clang creates a mandatory link-time requirement into the object produced by the translation of the primary module interface unit. This precludes the idea of "module-interface-only libraries," which would be the equivalent of what "header-only libraries" do in the pre-modules world.

2.2. Build Systems

2.2.1. CMake

As of this writing, CMake is close to officially releasing its support for C++ Modules, which is expected in Version 3.28. However, that initial release only supports those cases where all translation units are compiled in a consistent-enough way that allows the BMI to be reused. Future work is needed to allow the same module interface to be translated more than once to contribute to different translation units.

2.2.2. Visual Studio / MSBuild

Visual Studio is able to create solutions and projects that produce and consume modules.

2.2.3. Buck

There is no documentation available on how to use C++ Modules.

2.2.4. Buck2

There is no documentation available on how to use C++ Modules, but the build system should be able to support it in principle.

2.2.5. Bazel

There is an open issue[2] in the Bazel repository requesting support for C++ Modules, but it doesn't seem to have been prioritized yet.

[1] BOECKEL Ben, KING Brad, Format for describing dependencies of source files. https://wg21.link/P1689R4
[2] https://github.com/bazelbuild/bazel/issues/4005

2.2.6. Xmake

Xmake advertises support for C++ Modules[3].

2.2.7. Build2

Build2 advertises support for C++ Modules[4].

2.3. Static Analysis Tools

2.3.1. Clang-based tools

This includes tools such as clangd, which implements the Language Server Protocol. This is currently viable as long as the tool is built with a version of clang that exactly matches the version of the compiler used during the build.

The reason this works in that limited scenario is because the unmodified compilation command will point at BMI files that can be parsed by the tool. If the version of clang used to build the tool doesn't exactly match the version of the compiler used in the project, it will be unable to do it.

This is particularly relevant when you consider that IDEs, such as Visual Studio Code, have a configuration of clangd for the entire IDE, rather than for each project or compile command. Therefore, if a user has projects that use different versions of clang or different compilers entirely, clangd will not work as expected.

2.3.2. Visual Studio IntelliSense

Visual Studio does not currently support modules in its IntelliSense feature. There is a bug report open[5] for the issue.

2.3.2. GitHub Advanced Security

“C++20 support is currently in beta. Support is for GCC on Linux only. Modules are not supported”[6]

2.3.3. Coverity

Coverity does not yet support C++ Modules.

[3] https://github.com/xmake-io/xmake/wiki/Xmake-v2.7.1-Released,-Better-Cplusplus-Modules-Support
[4] https://build2.org/blog/build2-cxx20-modules-gcc.xhtml
[5] https://developercommunity.visualstudio.com/t/When-importing-a-C20-module-or-header-/1550846
[6] https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks/#id13

2.3.4. Fortify

As of this writing, there was no documentation on the support for C++ modules.

2.3.5. SonarQube

There is currently no support for C++ Modules in the SonarQube C++ scanner. There is an open thread requesting that feature[7].

2.4. Packaging

There are currently no interoperable facilities for C++ Module Libraries to be distributed, pre-built or not, and used across different build systems. Any usage of modules is restricted to interoperability within a single build system. In most cases, this is limited to projects where all sources can be compiled with arguments that are coherent enough for a single BMI to be used by the entire project.

Specifically in the case of CMake, it is possible to provide an installation with a CMake Export file and have that be imported by another CMake project. However, that will only work insofar as both projects use coherent-enough compile commands.

Visual Studio also currently does not have structured support for modules provided by libraries where the build is not a part of the solution itself.

3. Gaps for Tooling Interoperability

3.1. Releasing C++ Module Libraries

While we have reached consensus on a mechanism to identify the location of metadata files describing the modules provided by a library[8], we still don't have a specification for the format of that file. In particular, we need that to indicate how source files are going to be found, what the local preprocessor arguments are that need to be used when translating those modules, as well as information about any BMI that is shipped alongside a library.

The interaction with package managers is also currently under-explored, and it is likely that we have unknown unknowns on the impact of the adoption of C++ Modules.

Furthermore, it is likely that the modules provided by the standard library need to be advertised the same way, such that build systems can use a single mechanism to identify modules

[7] https://community.sonarsource.com/t/support-for-c-20-modules/63778
[8] RUOSO Daniel, Translating Linker Input Files to Module Metadata Files. https://wg21.link/P2701R0

provided by libraries, regardless of whether those are modules provided by the standard library or otherwise.

3.2. Identifying BMI Compatibility

This topic has already been addressed in SG15[9]. The current understanding is that the build system needs to, in principle, generate all BMIs needed for every translation unit, but de-duplicate those rules based on whether they are equivalent or not.

While the consensus involved a recommendation that compilers should offer an interface for identifying the compatibility of a given compiler command, this has not yet been implemented by any compiler. The status quo is that build systems need to checksum the baseline compile command to de-duplicate the rules, which may result in unnecessary translations.

It is important to further explore the impact of that limitation and evaluate which barriers are being seen by implementers in terms of getting that feature to be supported by the compilers.

3.3. Translating Importable Headers

In the 2023 meeting in Varna, we achieved significant advancements in terms of the requirements to implement the support for Importable Headers[10]. However, there are still gaps on how that will be manifested by the different parts of the tooling ecosystem, particularly in terms of how to communicate the baseline compile command to the dependency scanning process, as well as how to mark headers as importable. This is needed both for build systems and for other parts of the ecosystem, as noted in Section 2.3. They'll likely be implemented in different ways.

3.4. Introspecting the Build System for Static Analysis

Beyond the build and packaging workflows, static analysis workflows are another fundamental requirement for the adoption of C++ Modules[11]. However, the support for static analysis is currently very limited. One important aspect is that Static Analysis tools will need to be able to assemble their own topologically-sorted build plan to produce the Built Module Interface files themselves.

There are currently no specified mechanisms that would allow a static analysis tool that runs externally to access the build to create its own plan in order to analyze the code.

[9] RUOSO Daniel, Specifying the Interoperability of Binary Module Interface Files. https://wg21.link/P2581R2
[10] RUOSO Daniel, Build System Requirements for Importable Headers, https://wg21.link/P2898R1
[11] RUOSO Daniel, Requirements for Usage of C++ Modules at Bloomberg, https://wg21.link/P2409R0

4. Roadmap

The goal of this paper is to serve as a call to action on resolving the remaining issues on the specification for tooling interoperability. This paper hopes to make it clear for tool providers that this interoperability is stable and that it is safe to make investments to support C++ Modules.

The roadmap is presented in priority order for each section, with sub-sections that can be developed in parallel.

4.1. Working in a single project

The first deliverable of this roadmap is making sure module usage within a single project has the specifications needed for interoperability across tooling. There are two main parts for that deliverable.

4.1.1. Supporting multiple translations of a module interface unit

The initial support for modules in build systems have focused on being able to use the modules as long as the compilation commands are producing BMI that are usable across the entire project.

However, this is not sufficient for large scale deployments, as it's fairly frequent that different translation units will have compilation commands that diverge in ways that result in the need for multiple translations of the same module interface.

The goal is to clear any interoperability questions between build systems and compilers on correctly identifying how many translations are necessary, as well as establishing a model on what needs to be implemented for build systems that currently don't support modules.

4.1.2. Static analysis support from outside the build system

Even within a single project, being able to perform static analysis without a deep integration with the build system is a must-have requirement for a lot of organizations. For instance, someone using an IDE has an expectation that it will be able to do live correctness checks and syntactic completion and lookup.

For this to be possible, it's important that build systems produce enough information for tools to introspect what the build is doing and reproduce all necessary translations to perform the analysis of the code.

While there will be work required by individual static analysis tools to correctly perform the topologically-sorted translation of module interface units, the goal is to make sure that enough information is presented by the build system with a documented interface, such that static analysis tools only need to implement that integration once.

4.2. Deploying pre-built libraries

While the ongoing work in SG15 to drive the convergence in the area of package management will eventually have to include support for C++ Modules, we should not couple those two things together. Instead, we should promote interoperability across build systems and package managers in the support for C++ Modules.

4.2.1. Describing modules in pre-built libraries

In P2701R0, we reached consensus on a mechanism for a library to describe which modules it provides. However, we still need to finalize the specification of the schema for that metadata file.

4.2.2. Metadata for standard library modules

C++23 includes modules for the standard library. While the compiler itself is probably in a position to communicate the information about those modules to the build system and package managers, it is also important to consider that the code may be translated by an external tool, such as for static analysis. Therefore, interoperability in that description is also important.

4.3. Importable Headers

In P2898R1, we reached consensus on important aspects for the implementation of support for header units. However, we are still far from a complete interoperable solution.

4.3.1. Identification of Importable Headers

We need collaboration with tool implementers to understand what problems will arise as build systems and package managers need to identify what the importable headers are, in the context of a project.

4.3.2. Dependency scanning and the preprocessor state

There are still unanswered questions as to the impact of the requirements for the emulation of the import mechanism when performing the dependency scanning.

5. Methodology

One of the challenges encountered by SG15 over the past two years has been the limited number of engineers who are contributing papers exploring the implementation of C++ Modules by the tooling ecosystem.

The following sections will describe the steps that are being proposed to navigate each roadmap item.

5.1. Call for Requirements Papers

The first step is an explicit Call for Papers, specific to the topic of what challenges implementers foresee for that particular feature. This will require more active engagement from SG15 with tooling implementers in order to motivate them to participate in this exercise.

5.2. Consolidated Requirements

Once implementers present their papers, SG15 will work in consolidating those requirements into a single unified document.

5.3. Call for Proposal Papers

Once we reach consensus on the consolidated requirements, we will do a second call for papers to ask for proposals on how to address those requirements.

5.4. Merge into Modules Ecosystem TR Draft

Once we reach consensus on specific proposals, they will be merged into the draft Modules Ecosystem Technical Report Draft, settling the direction forward in a way that will instill confidence among implementers on the practicality of investing on the support for C++ modules.