2023

C++ Modules

The Challenges of Implementing Header Units

Daniel Ruoso

C++ now

C++ Modules: The Challenges of Implementing Header Units

Bloomberg

Engineering

C++Now 2023
May 10, 2023

Daniel Ruoso
Software Engineering Manager, Code Governance

TechAtBloomberg.com

About me

- 20+ years working in and around build systems and package managers
- Introduced the package management system at Bloomberg that is used today for 10K+ C++ projects
- In the last five years, I've focused on Static Analysis, Automated Refactoring, and Building Consensus on Engineering practices
- In the last two years, I've collaborated with the ISO C++ Tooling Study Group to help figure out C++ Modules

Agenda

- Background on the work on C++ Modules
- Review of the Named Modules tooling
- The challenges of implementing header units
- Where do we go from here?

Background

- Initial work in modules was focused in highly-regulated environments (i.e., mono-repos)
- Bloomberg has a more open-ended package management approach, closer to what GNU/Linux distributions do
- In July 2021, Bloomberg got more involved in the ISO C++ Tooling Study Group with the focus of making C++ Modules work in our environment

Papers:

- P2409R0: Requirements for Usage of C++ Modules at Bloomberg

Background

- Over the past two years, there has been significant progress in figuring out Named Modules
- Bloomberg has been working with Kitware; CMake now has experimental support for them
- There's still significant work to be done for Named Modules to be usable in production environments

Background

- We still don’t have a coherent plan for Header Units
- At the meetings in Kona and Issaquah, we made very little progress
- One of the sessions where I was presenting a paper ended with me saying “Wait, forget everything I said. None of this works.”
- Hence this talk…

Named Modules

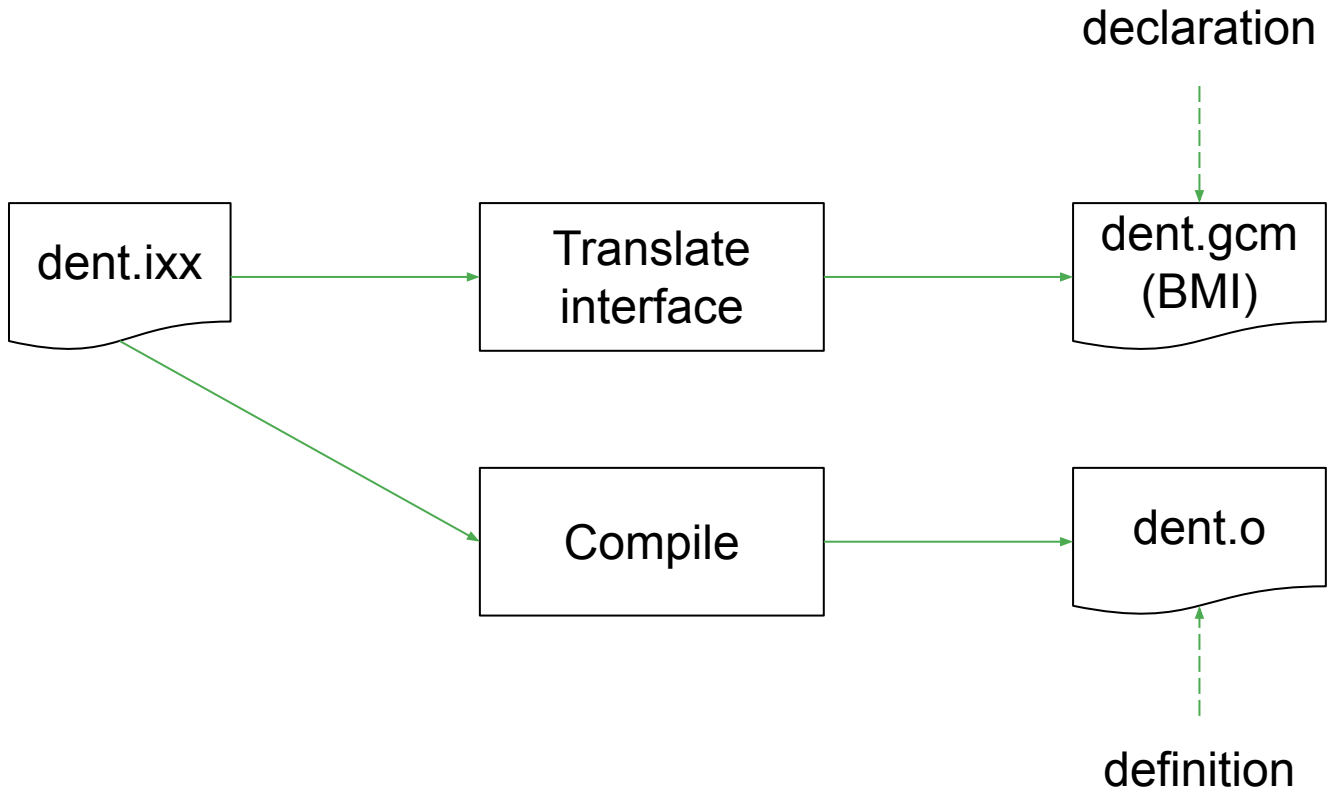
TechAtBloomberg.com

Bloomberg

Engineering

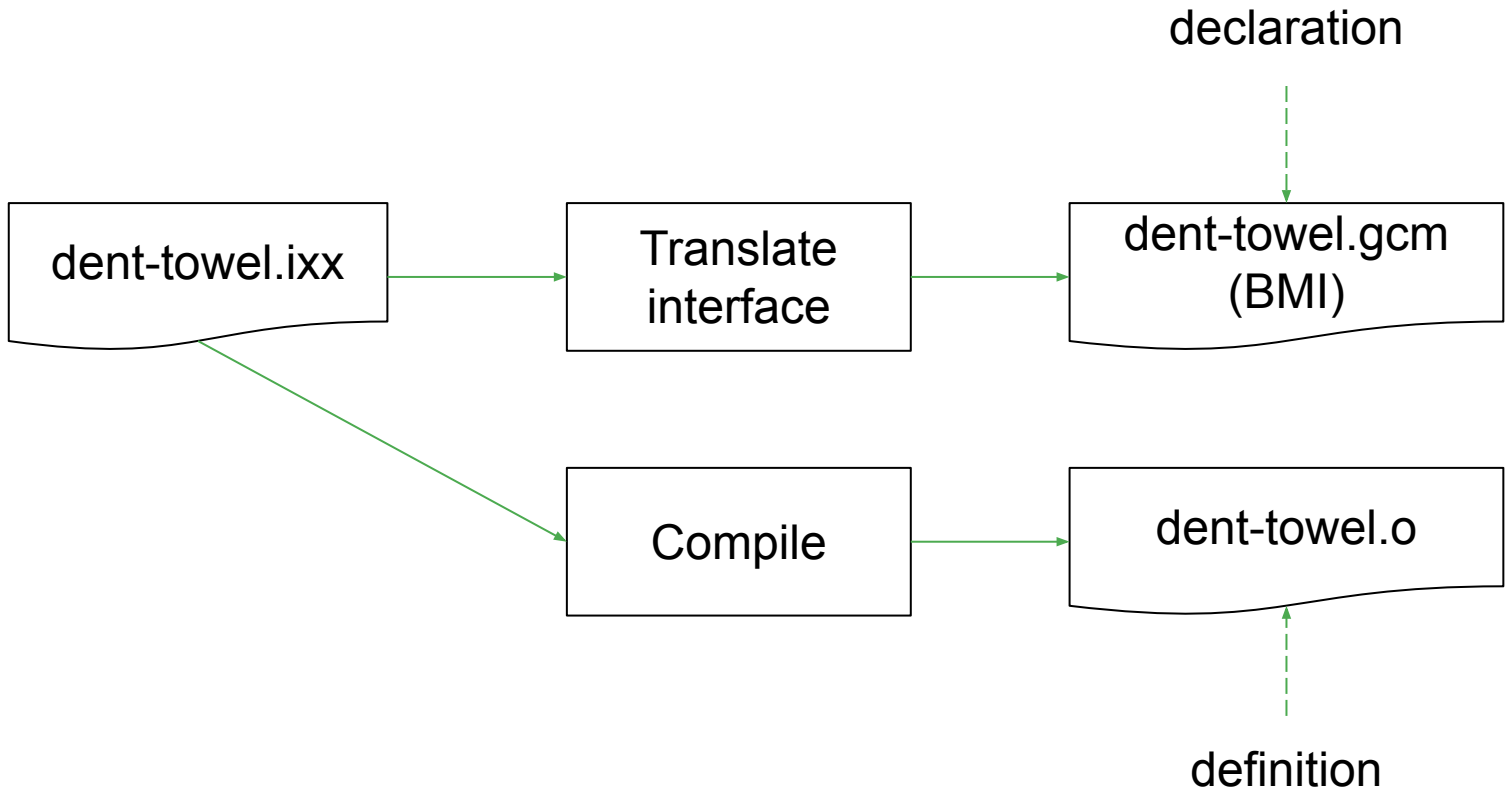
Primary Module Interface Unit

```
1  export module dent;
2  export namespace dent {
3    int dent_value() {
4      return 42;
5    }
6    int other();
7  }
```



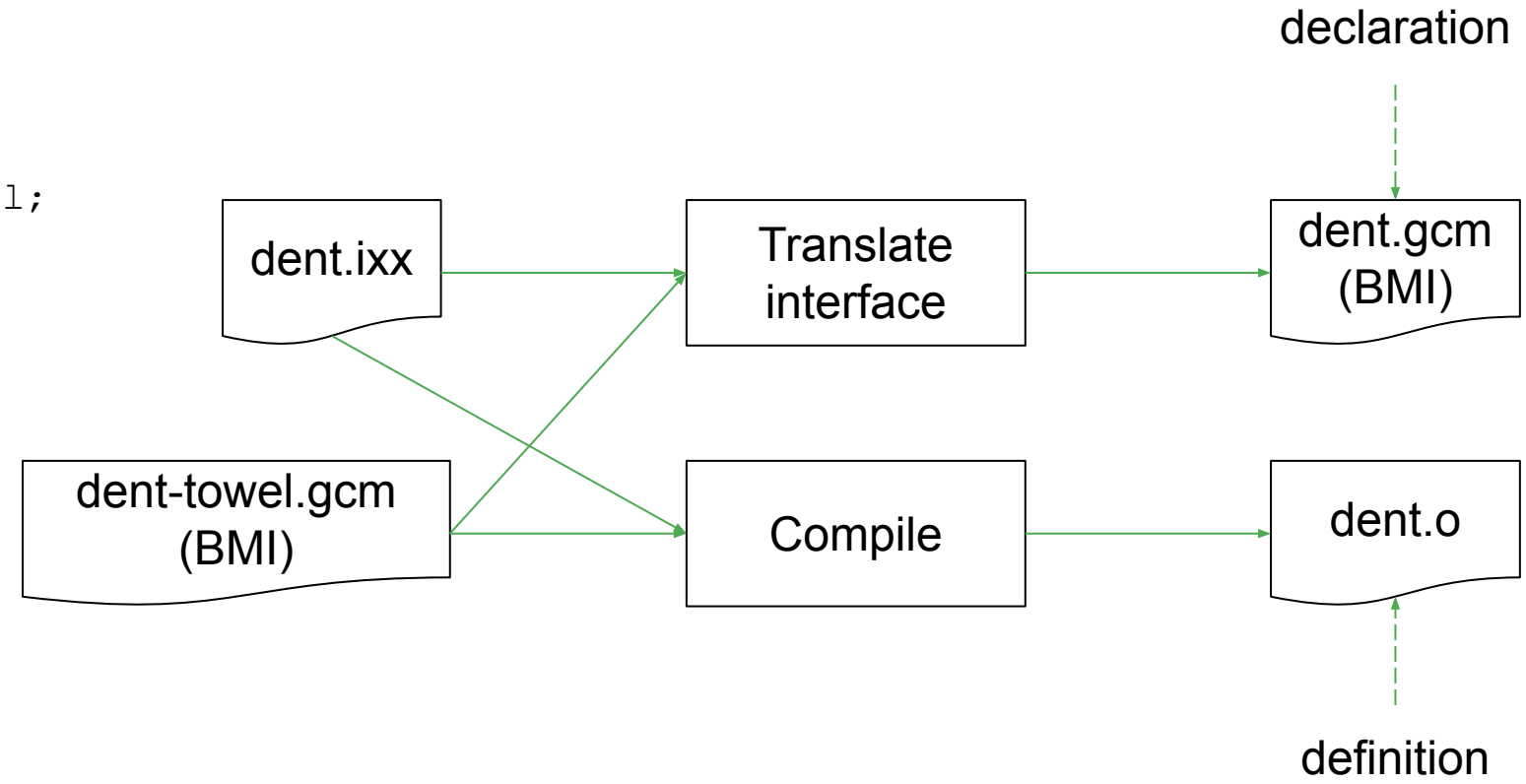
Module Interface Partition Unit

```
1  export module dent:towel;
2  export namespace dent {
3    int towel_size() {
4      return 42;
5    }
6    int something_else();
7  }
```



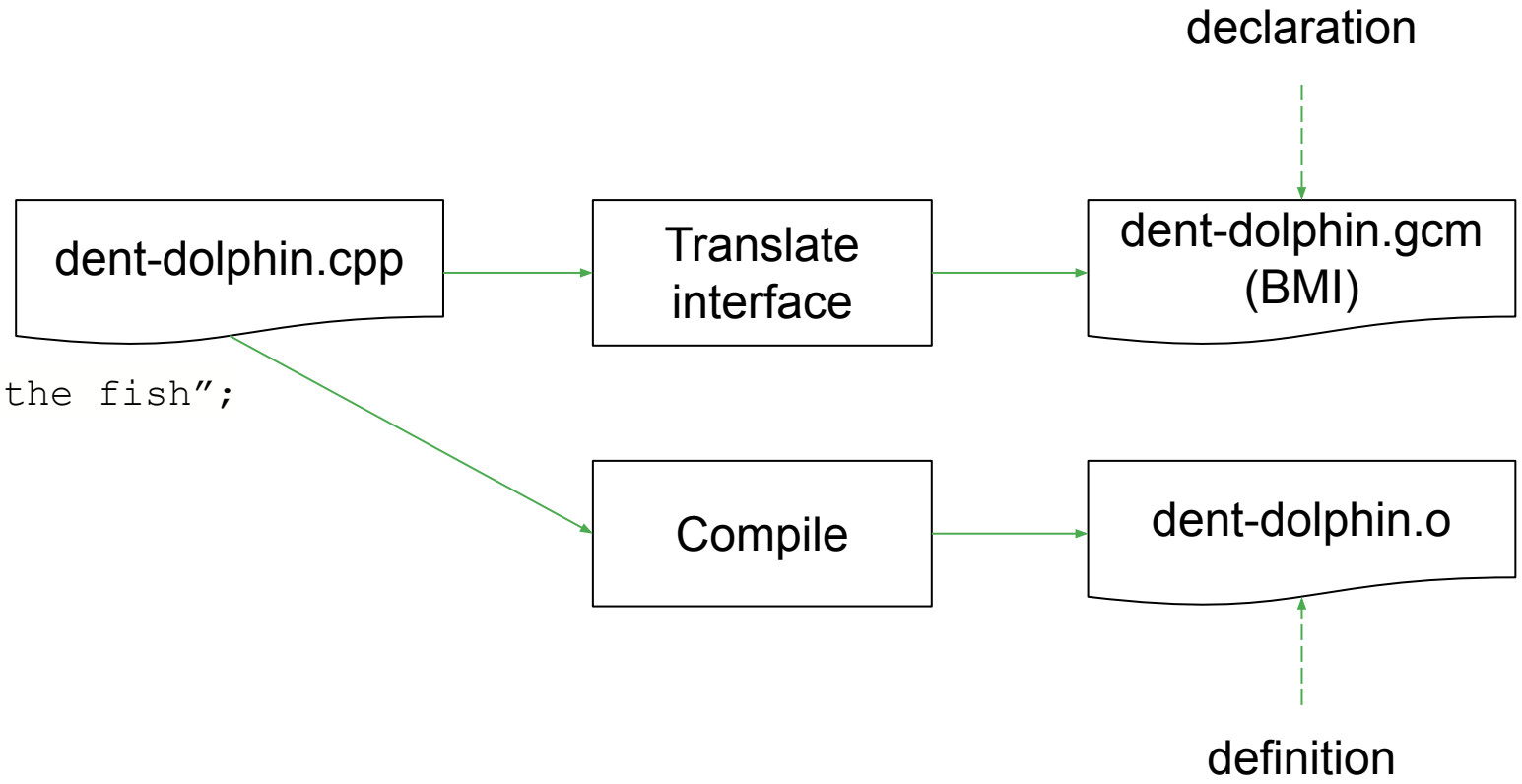
Primary Module Interface Unit, revisited…

```
1  export module dent;

3  export import dent:towel;

2  export namespace dent {

3    int dent_value() {

4      return 42;

5    }

6    int other();

7  }
```



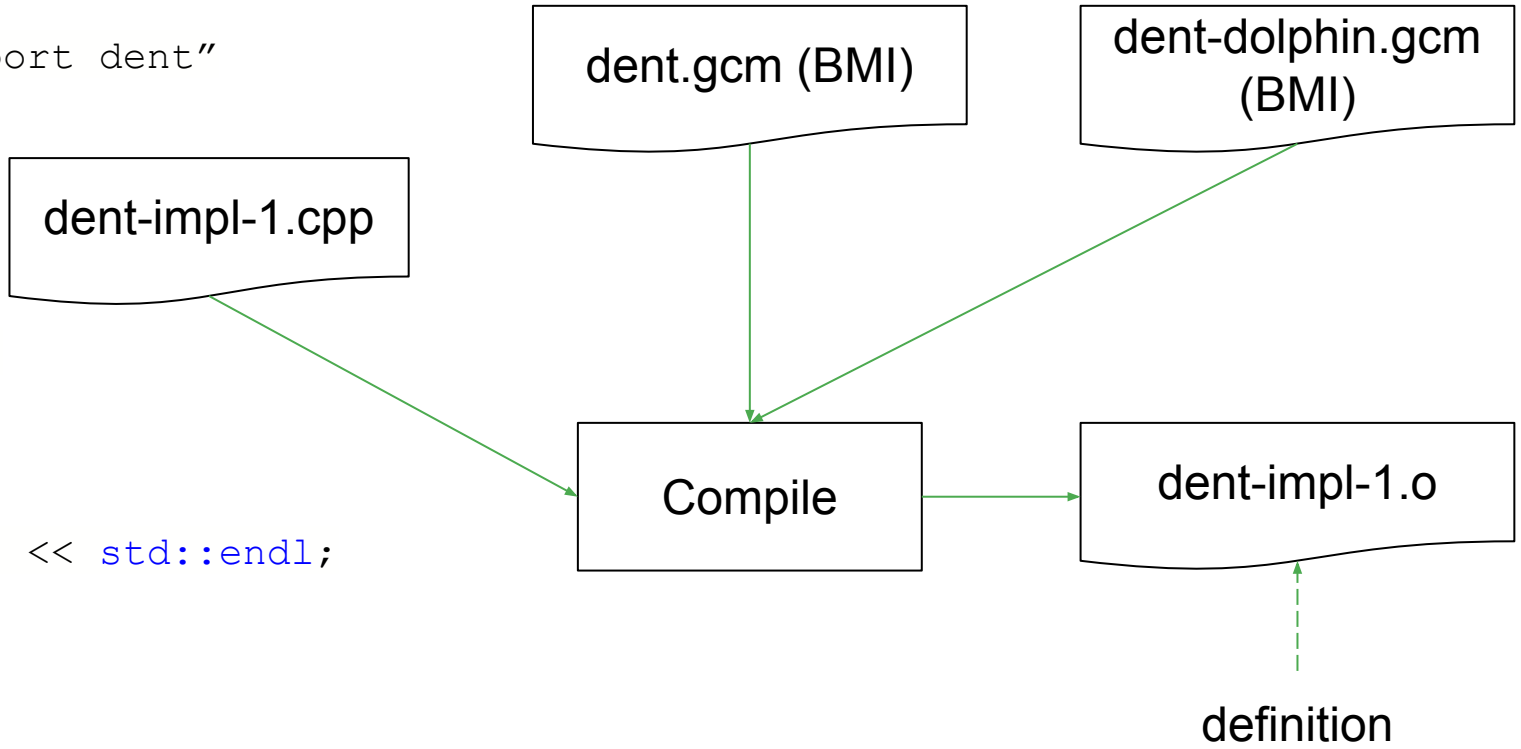
Internal Module Partition Unit

```
1  module dent:dolphin;
2  namespace dent {
3    const char* farewell() {
4      return “thanks for all the fish”;
5    }
6    void filing_cabinet();
7  }
```



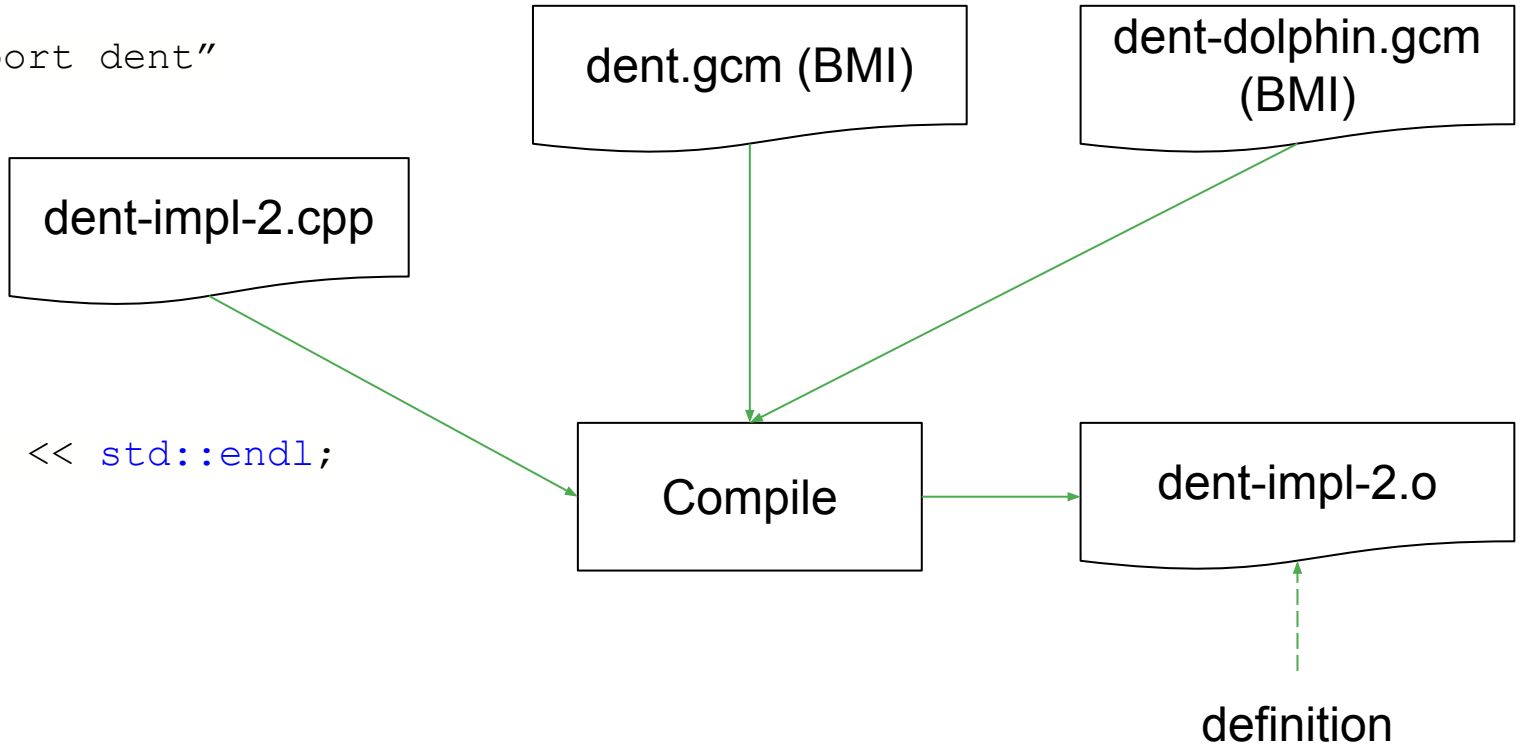
Module Implementation Unit

```
1  module dent; // implies “import dent”
2  import dent:dolphin;
3  namespace dent {
4    int other() { return 42; }
5    int something_else() {
6       std::cout << farewell() << std::endl;
7       return 34;
8    }
9  }
```

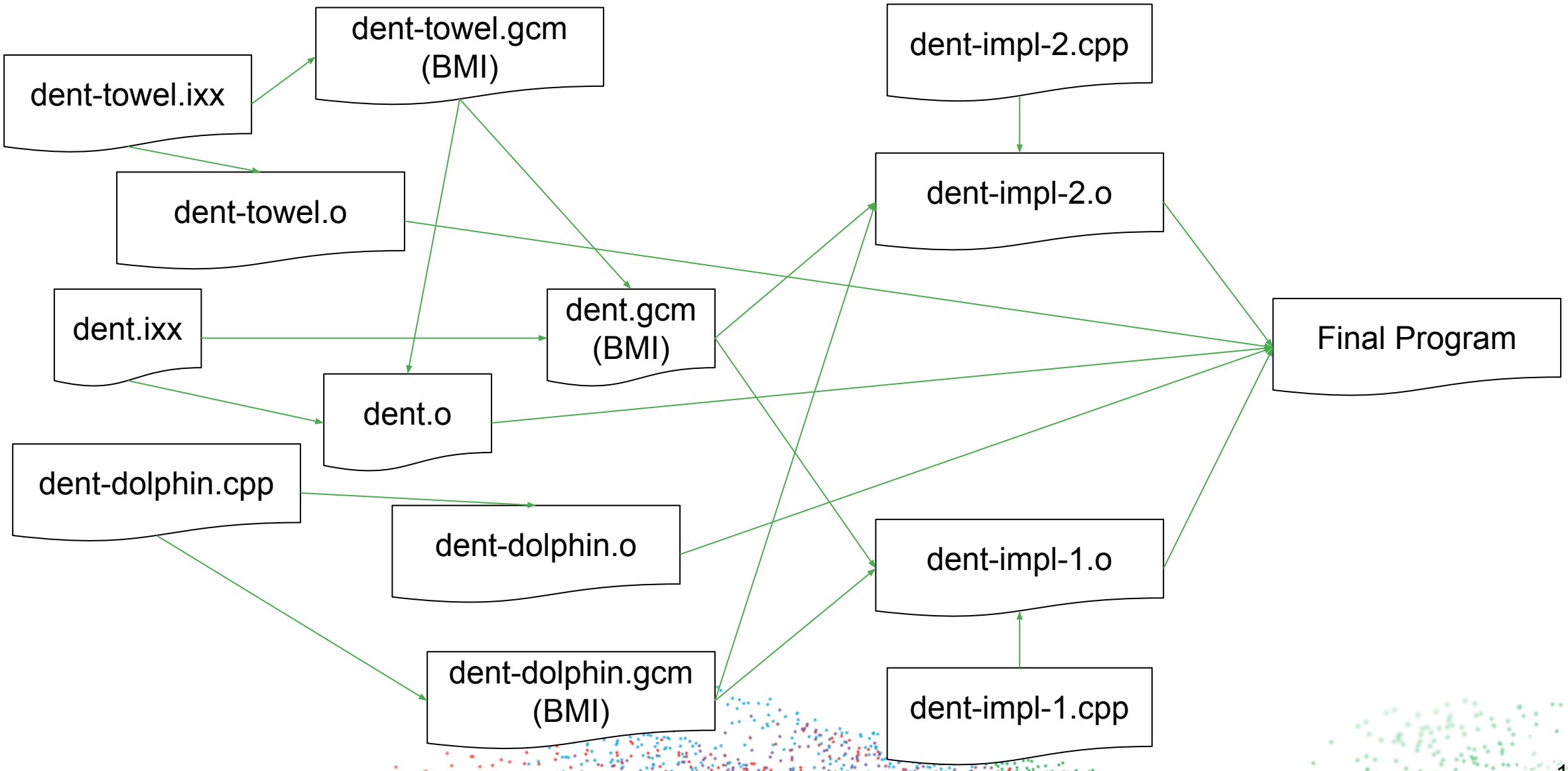


Module Implementation Unit

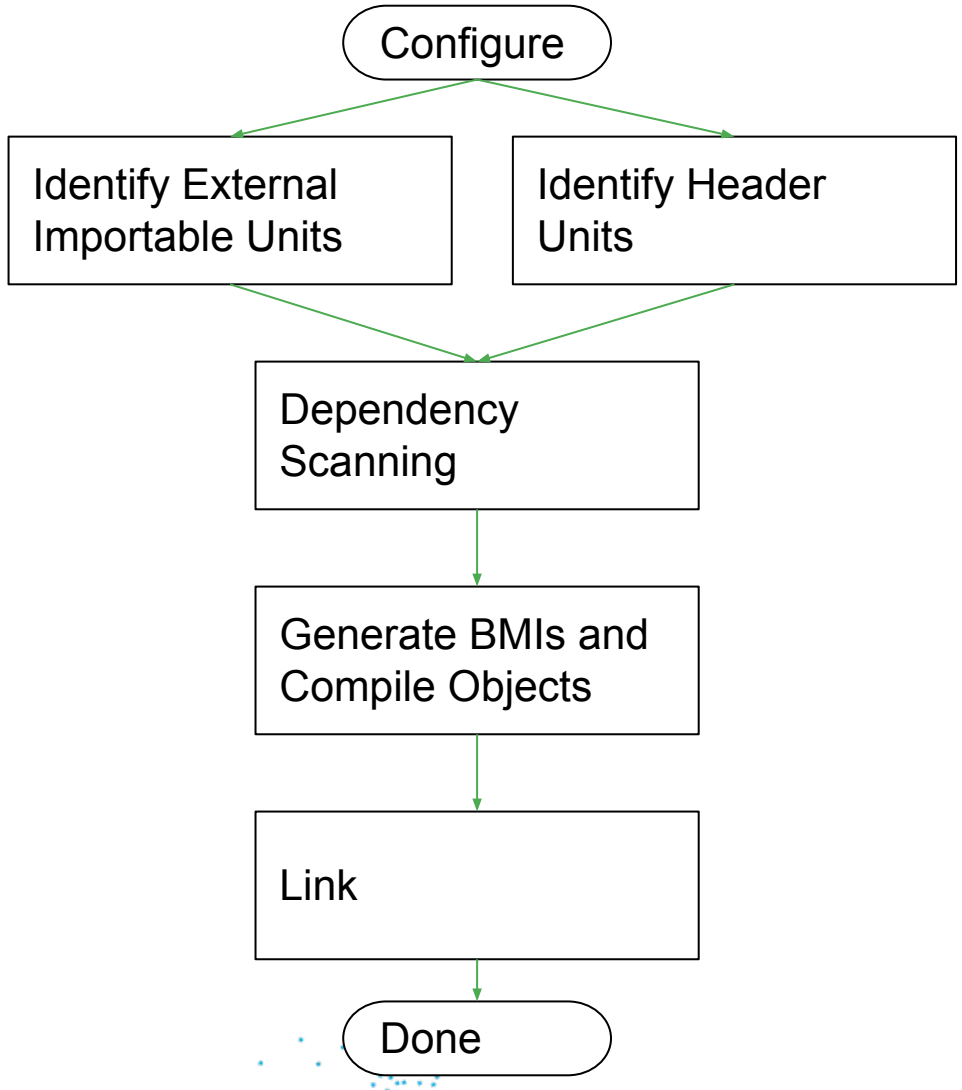
```
1  module dent; // implies “import dent”
2  import dent:dolphin;
3  namespace dent {
4    int filing_cabinet() {
5        std::cout << farewell() << std::endl;
6        return 34;
7    }
8  }
```



Named Modules, visualized



Steps of the Build Process



External Importable Units

- Which modules exist on the system?
- Are there existing BMIs I can use?
- How do I produce my own BMI?

External Importable Units

- No interoperability across package managers for discovery…
- No interoperability across build systems…

External Importable Units

- Use the link line to drive discovery
- Co-locate metadata file with the library
- Describe information about modules in metadata file

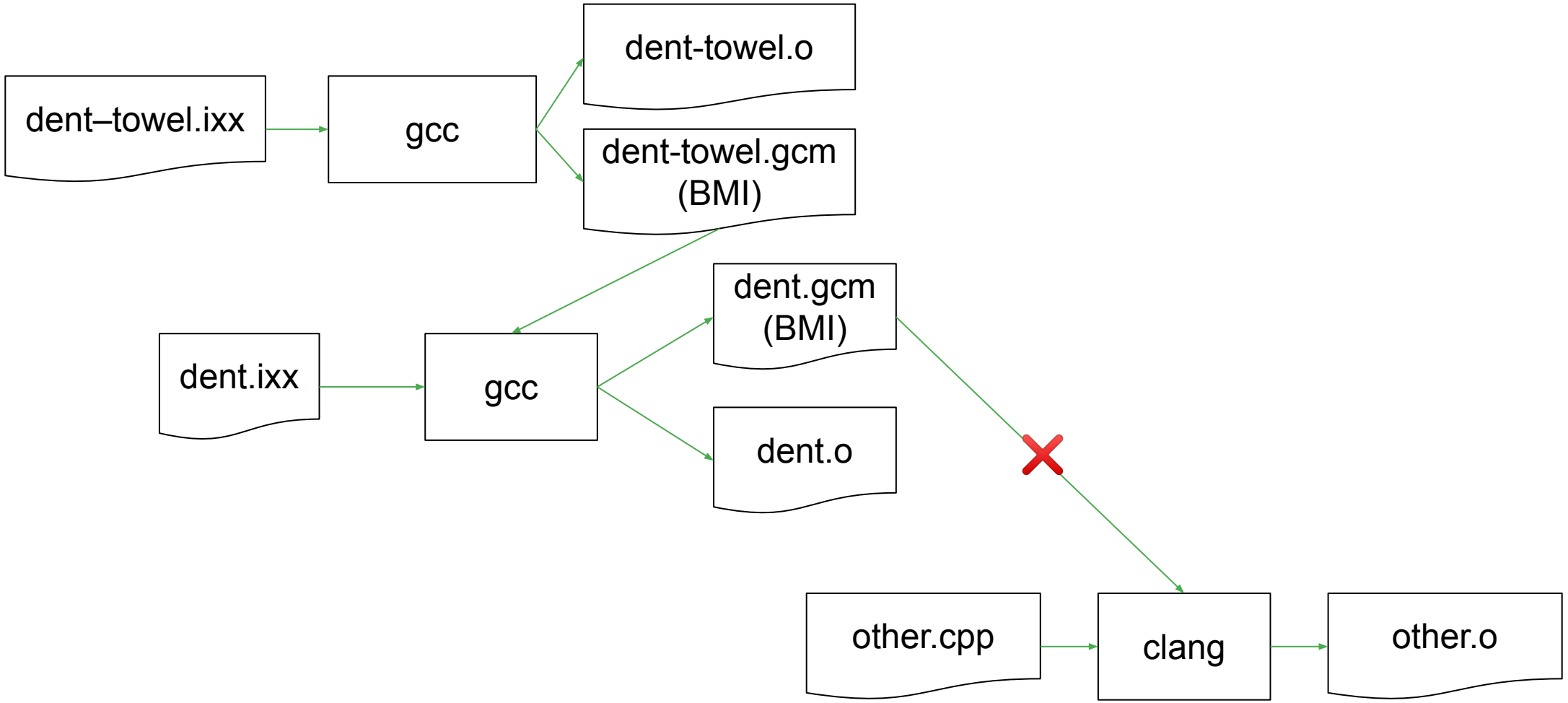
Papers:

- P2577R2: C++ Modules Discovery in Prebuilt Library Releases
- P2701R0: Translating Linker Input Files to Module Metadata Files

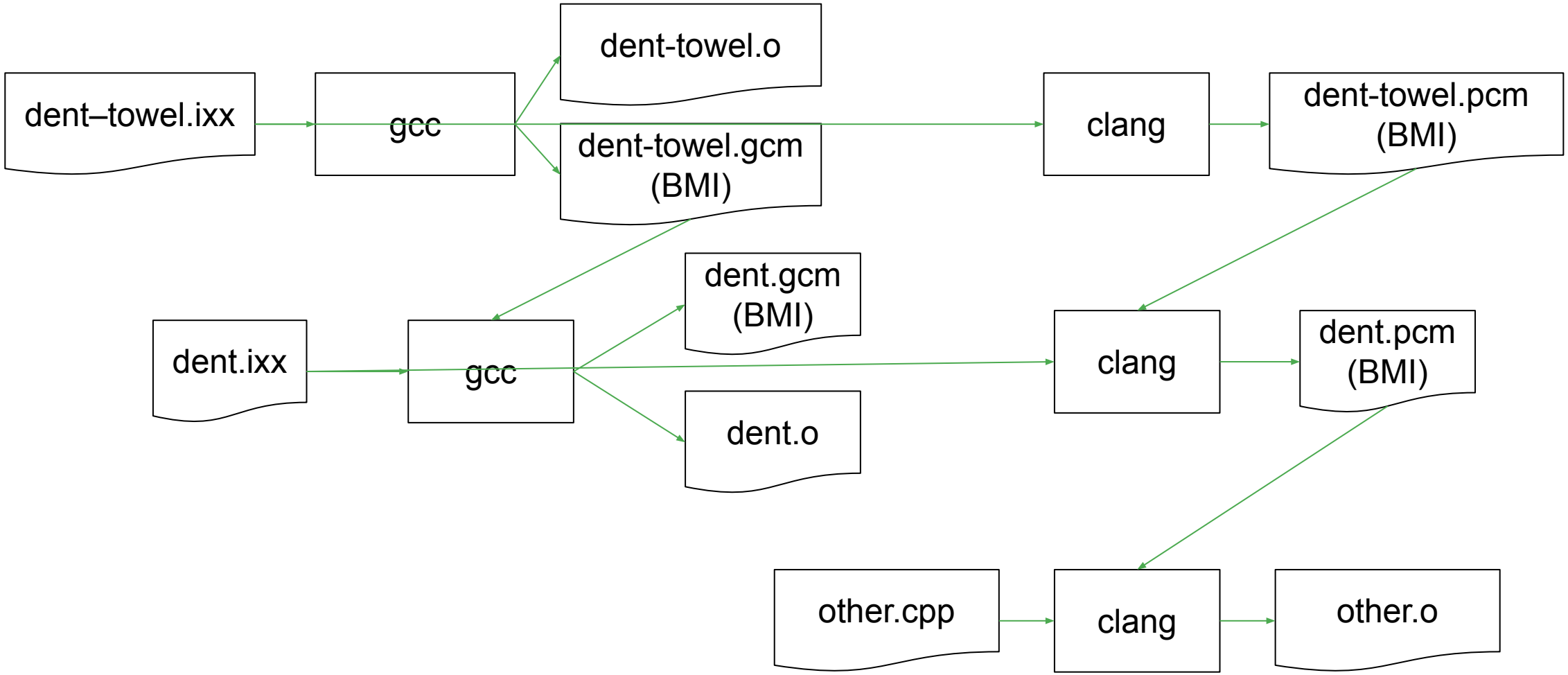
Sidebar: Why do I need to produce my own BMI?

- BMI is much more tightly coupled with the compilation
- Offering pre-built BMI is just an optimization for narrow use cases
- You can: link code from different compilers
- You can't: import a BMI from a different compiler
- ...even a different version of the same compiler.

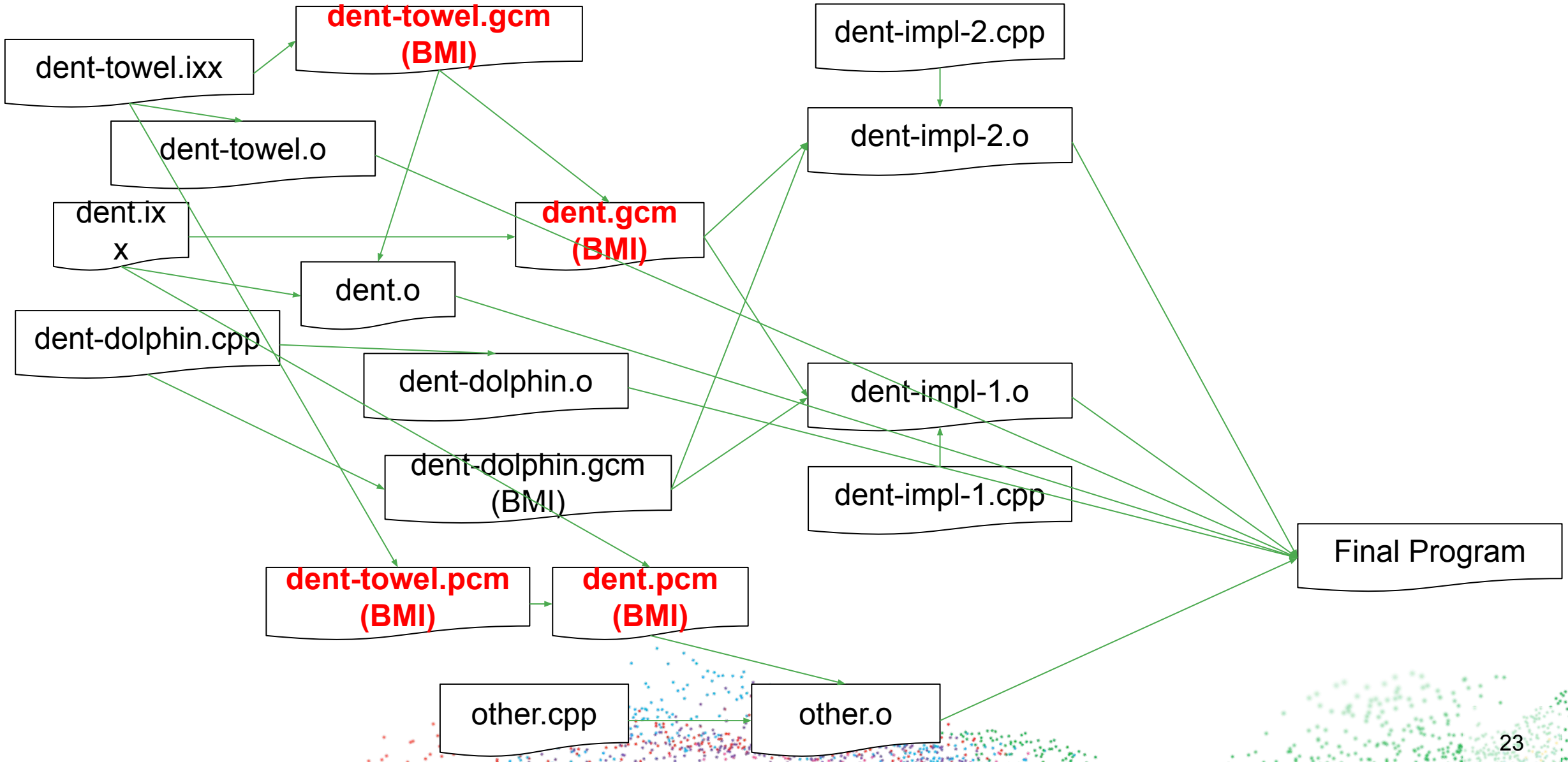
Sidebar: Why do I need to produce my own BMI?



Sidebar: Why do I need to produce my own BMI?



Side Sidebar: Important note about BMIs



Side Sidebar: So, how do I produce my own BMI?

- I don’t know how the original library was compiled
- Even if I did, it may have been a different compiler
- Even if it was the same, I could end up producing an incompatible BMI

Side Sidebar: So, how do I produce my own BMI?

- New concept of “Local Preprocessor Arguments”
- Metadata for the module must provide that
- Transform the compile command for the translation unit doing the import into one to generate the BMI

Paper:

- P2581R2: Specifying the Interoperability of Built Module Interface Files

Side Sidebar: So, how do I produce my own BMI?

```
g++ -std=c++20 -fmodule-mapper=dent.map \
 -I/path/to/some/header -DOPTION_FOR_SOME_HEADER=1 \
 -c dent.ixx -o dent.o

clang++ -std=c++20 -fmodule-mapper=other.map \
 -I/some/other/ -DOPTION_FOR_OTHER=1 \
 -c other.cpp -o other.o

clang++ -std=c++20 -fmodule-mapper=dent_for_other.map \
 -I/path/to/some/header -DOPTION_FOR_SOME_HEADER=1 \
 -c dent.ixx
```

Remove Local Preprocessor Arguments from the one doing the import

Add Local Preprocessor Arguments from the one being imported

Sidebar: Why do I need to produce my own BMI?

- All this before the dependency scanning starts

Which is a good segue into…

Dependency Scanning

- You need a preprocessor in order to know which modules are provided and imported by a translation unit
- There's some relief in the standard to allow only selectively expanding macros as an optimization
- It needs to run on every source file

Dependency Scanning

- Must scan sources from modules external to the current project
- The dependency scanning needs to be re-run if arguments change
- It depends on resolving the compiler command for all modules that may be imported

Dependency Scanning

- Modules don’t change preprocessor state
- Scan is a single pass to identify nodes and edges in the module graph
- De-facto standard format for the output

Papers:

- P1689R5: Format for describing dependencies of source files

Dependency Scanning

- All possible compile commands need to be known ahead of time
- The build graph is complete and stable after the dependency scan
- Files changed need to be re-scanned, which can adjust the shape of the graph

Sidebar: What about incompatible BMIs?

- Only the compiler knows which flags can cause a BMI to be incompatible with another translation unit
- The build system needs to defer that logic to the compiler
- Worst-case scenario, build all BMIs for each translation unit that needs to import those modules

Sidebar: What about incompatible BMIs?

Current implementation in CMake:

- BMI is owned by the target declaring the module
- Assume BMIs are going to be usable
- User error otherwise

Sidebar: What about incompatible BMIs?

What we want instead?

- BMI is owned by the translation unit doing the import
- Work is de-duplicated when possible
- Usage of ABI-compatible flags shouldn’t be an user error just because of BMI-incompatible flags
- External libraries can ship BMIs, which may or may not be used by the build

Sidebar: What about incompatible BMIs?

- The compiler should offer a new interface to “give me an identifier for the compatibility of BMIs”
- Build system strips the “Local Preprocessor Arguments” and gets the identifier
- Use the identifier in the target names to de-duplicate the BMI
- No compiler supports this yet

Paper:

- P2581R2: Specifying the Interoperability of Built Module Interface Files

Sidebar: Specifying dependent modules

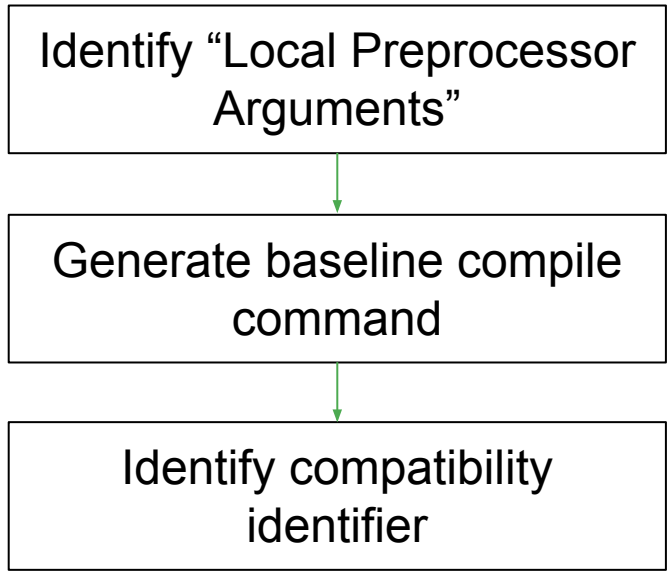
- The “module mapper” has been one of the hottest debate topics in early discussions about modules
- An early proposal included a network protocol between the compiler and the build system to resolve and schedule the translation of BMIs
- This is incompatible with remote execution, because that requires all inputs to be known beforehand

Sidebar: Specifying dependent modules

- Alternative is to provide a static module map to the compiler with the BMI files that need to be loaded
- But if you over-specify, it invalidates unrelated translations when a module changes
- Solution is to have the build system generate per-translation-unit input module maps (CMake does that)

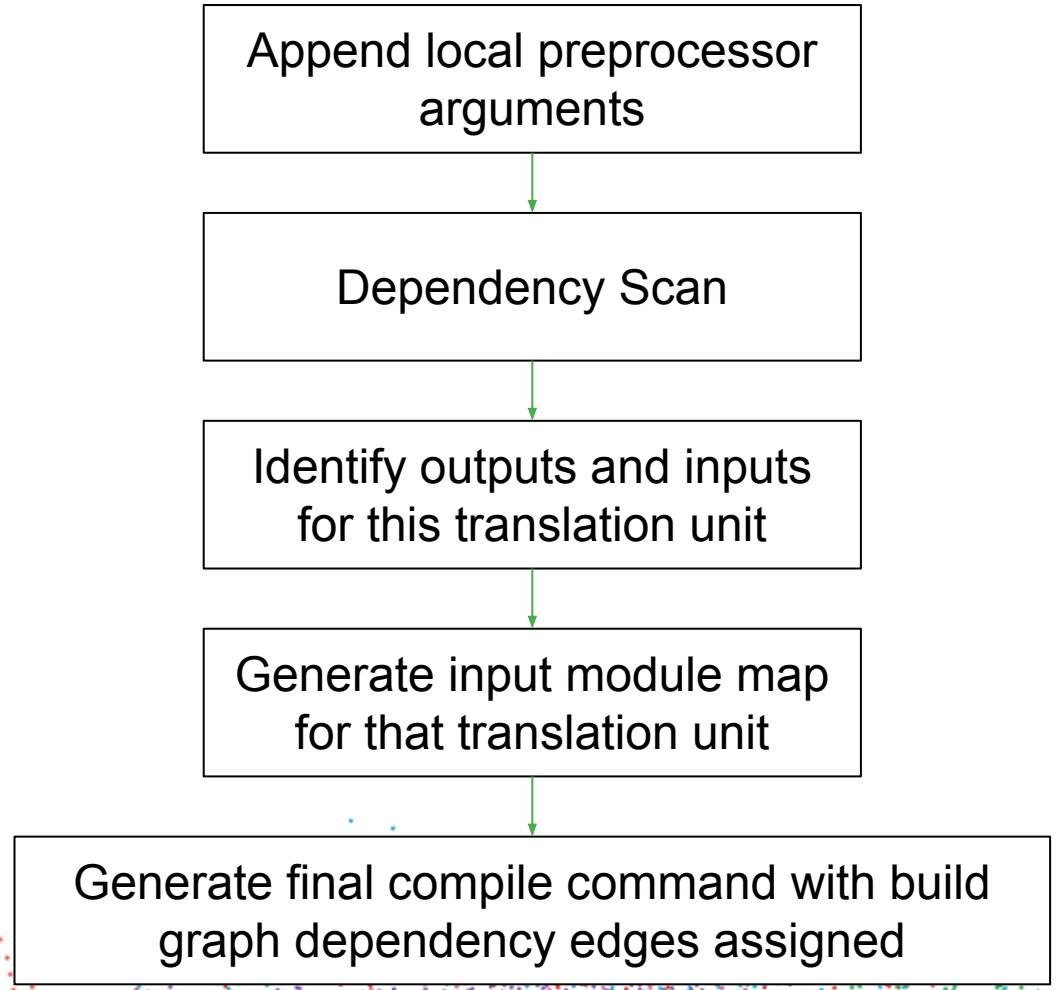
Generate BMIs and Compile Objects

For each source file in the project:



Generate BMIs and Compile Objects

For each compatibility identifier in the project:
  For each translation unit (external module, or internal to the project):



Generate BMIs and Compile Objects

- At that point you have a complete build plan and graph
- You can now invoke ninja or make with the generated plan
- Objects can depend on many BMIs
- BMIs can depend on many BMIs

Generate BMIs and Compile Objects

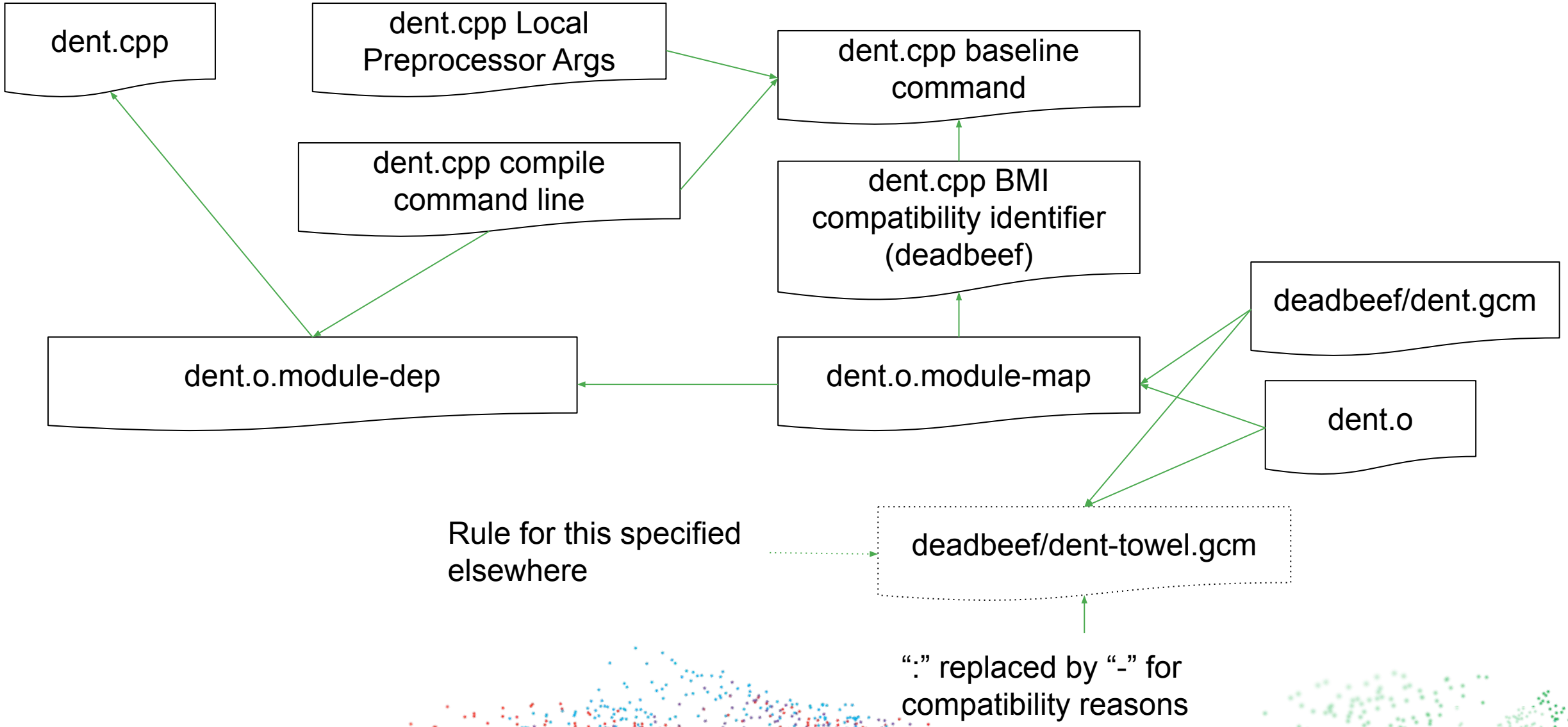
Changes may:

- Introduce a new module
- Remove a module
- Add a new dependency
- Remove a dependency
- Change the compatibility identifier of the BMI

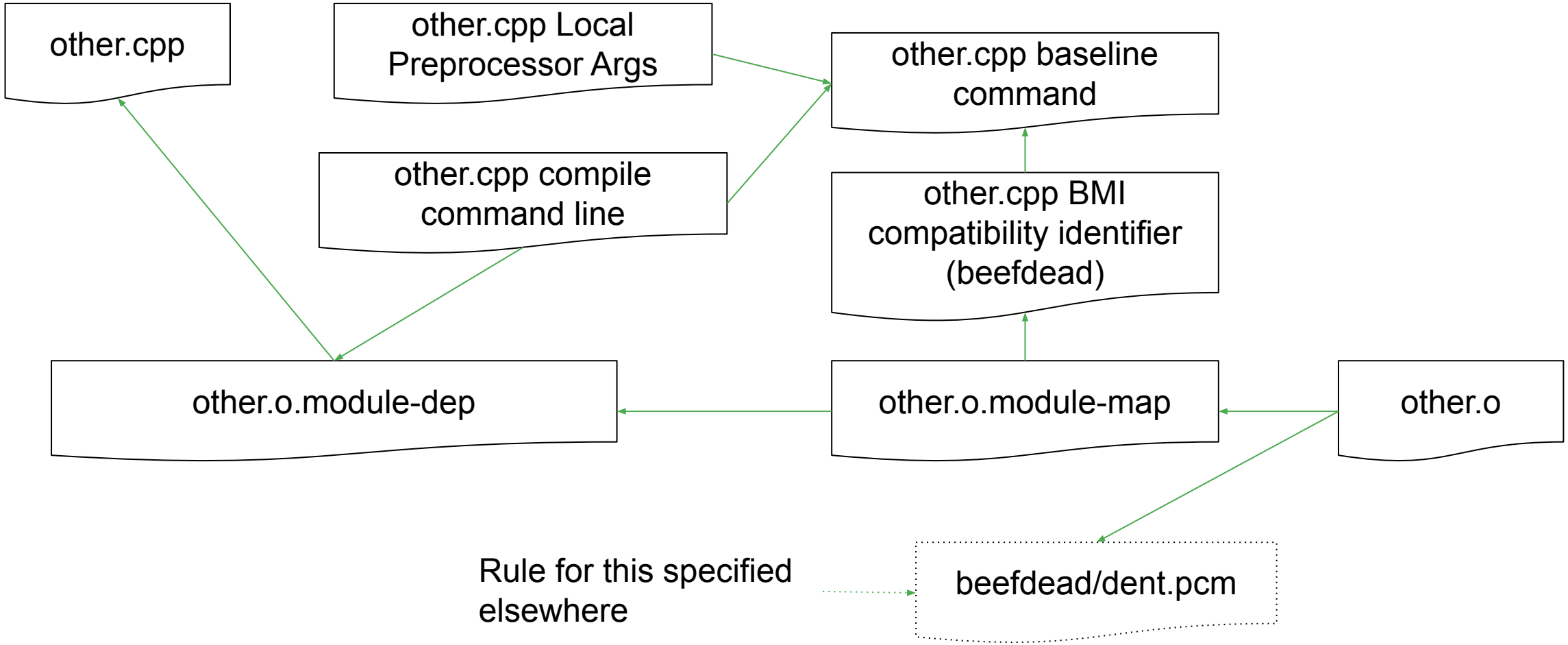
Generate BMIs and Compile Objects

- Use the compatibility identifier in the name of the BMI file
- Use the name of the module for the output BMI file
- That will reduce the number of times global information is needed

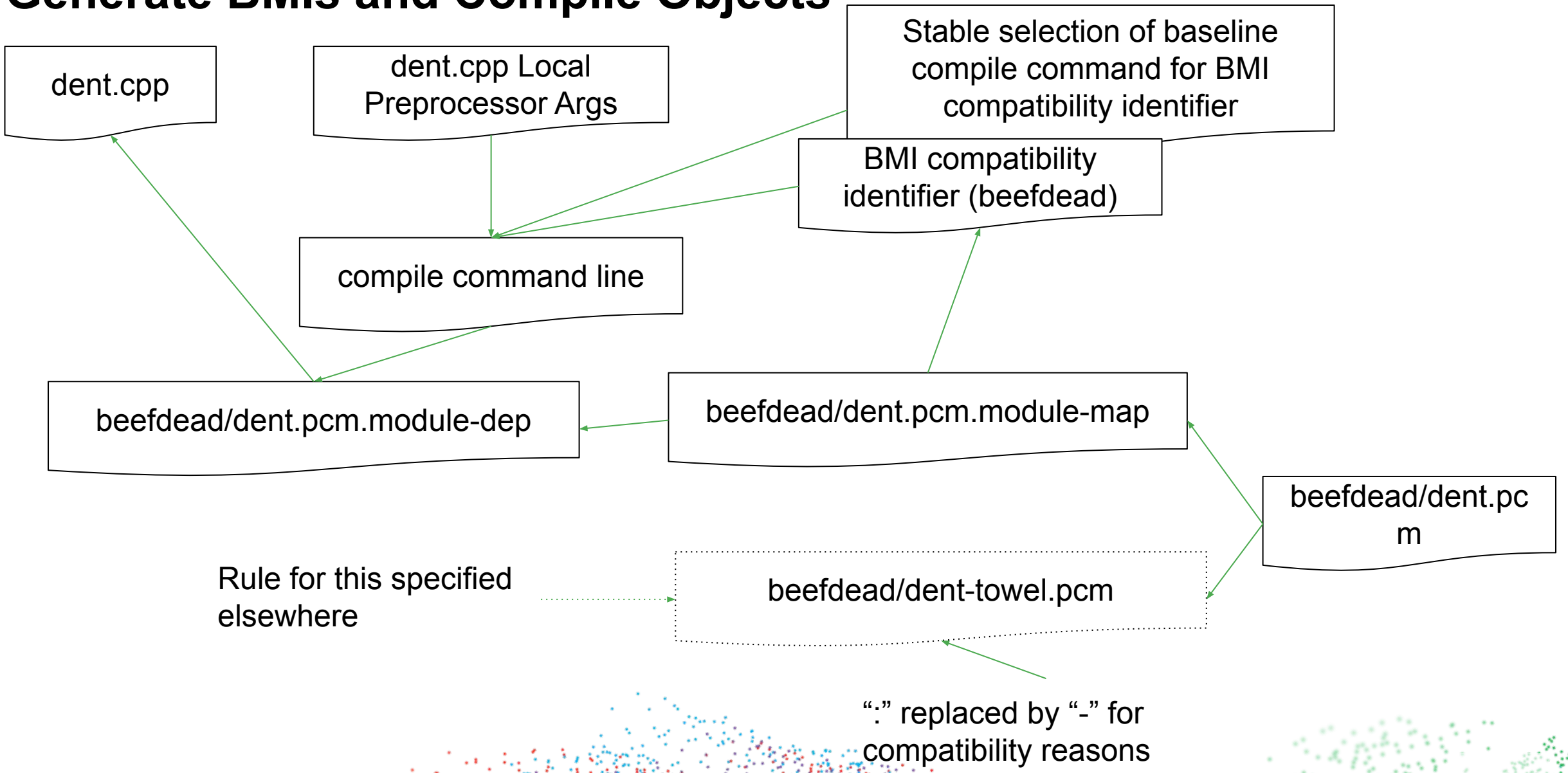
Generate BMIs and Compile Objects



Generate BMIs and Compile Objects



Generate BMIs and Compile Objects



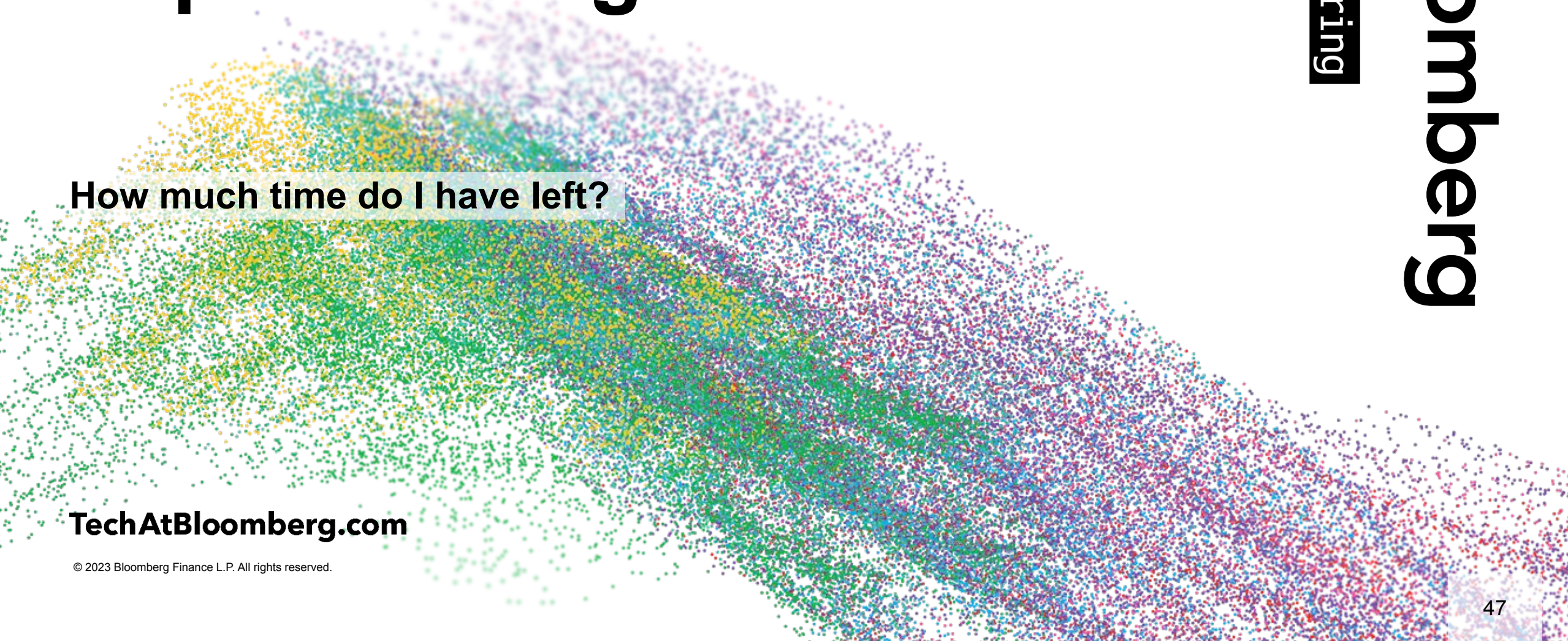
Questions so far?

I hope I already answered why you can’t “just use make” anymore

The Challenges of Implementing Header Units

Bloomberg

Engineering



How much time do I have left?

TechAtBloomberg.com

Preprocessor State

Preprocessor state in source inclusion

DENT is defined at the time of #include

```
1  #define DENT
2  #include <bar.h>
3  int main() {
4      return BAR_VALUE;
5  }
```

```
1  #ifdef DENT
2  #define BAR_VALUE 42
3  #else
4  #define BAR_VALUE 1
5  #endif
```

Preprocessor state in header imports

```
1  #define DENT
2  import <bar.h>
3  int main() {
4      return BAR_VALUE;
5  }
```

???

DENT may or may not be defined, it depends on the command line. It is isolated from the TU doing the import

```
1  #ifdef DENT
2  #define BAR_VALUE 42
3  #else
4  #define BAR_VALUE 1
5  #endif
```

Okay, so I need to translate header units first…

Unfortunately, it’s not that simple…

Preprocessor state in header imports affects dependency graph

```
1  #define DENT
2  import <bar.h>
3  import BAR_VALUE
4  int main() {
4      return BAR_NUMBER;
5  }
```

```
1  #ifdef DENT
2  #define BAR_VALUE <one_bar.h>
3  #else
4  #define BAR_VALUE <other_bar.h>
5  #endif
```

```
1  #define BAR_NUMBER 1
```

```
1  #define BAR_NUMBER 2
```


Okay, so I need to translate the header units in the right order

- Yes, but you need that **before** you can do the dependency scanning
- The dependency scanning has to **emulate** the import behavior

Emulating the import in the dependency scanning

```
1  #define DENT
2  import <bar.h>
3  import BAR_VALUE
4  int main() {
4      return BAR_NUMBER;
5  }
```

Push the state of the preprocessor away, start a new one and read the imported header

```
1  #ifdef DENT
2  #define BAR_VALUE <one_bar.h>
3  #else
4  #define BAR_VALUE <other_bar.h>
5  #endif
```

```
1  #define BAR_NUMBER 1
```

```
1  #define BAR_NUMBER 2
```

Pop the state and merge what was defined in the header unit

Emulating the import in the dependency scanning

```
1  #define DENT
2  import <bar.h>
3  import BAR_VALUE
4  int main() {
4      return BAR_NUMBER;
5  }
```

But where does DENT comes from anyway?

```
1  #ifdef DENT
2  #define BAR_VALUE <one_bar.h>
3  #else
4  #define BAR_VALUE <other_bar.h>
5  #endif
```

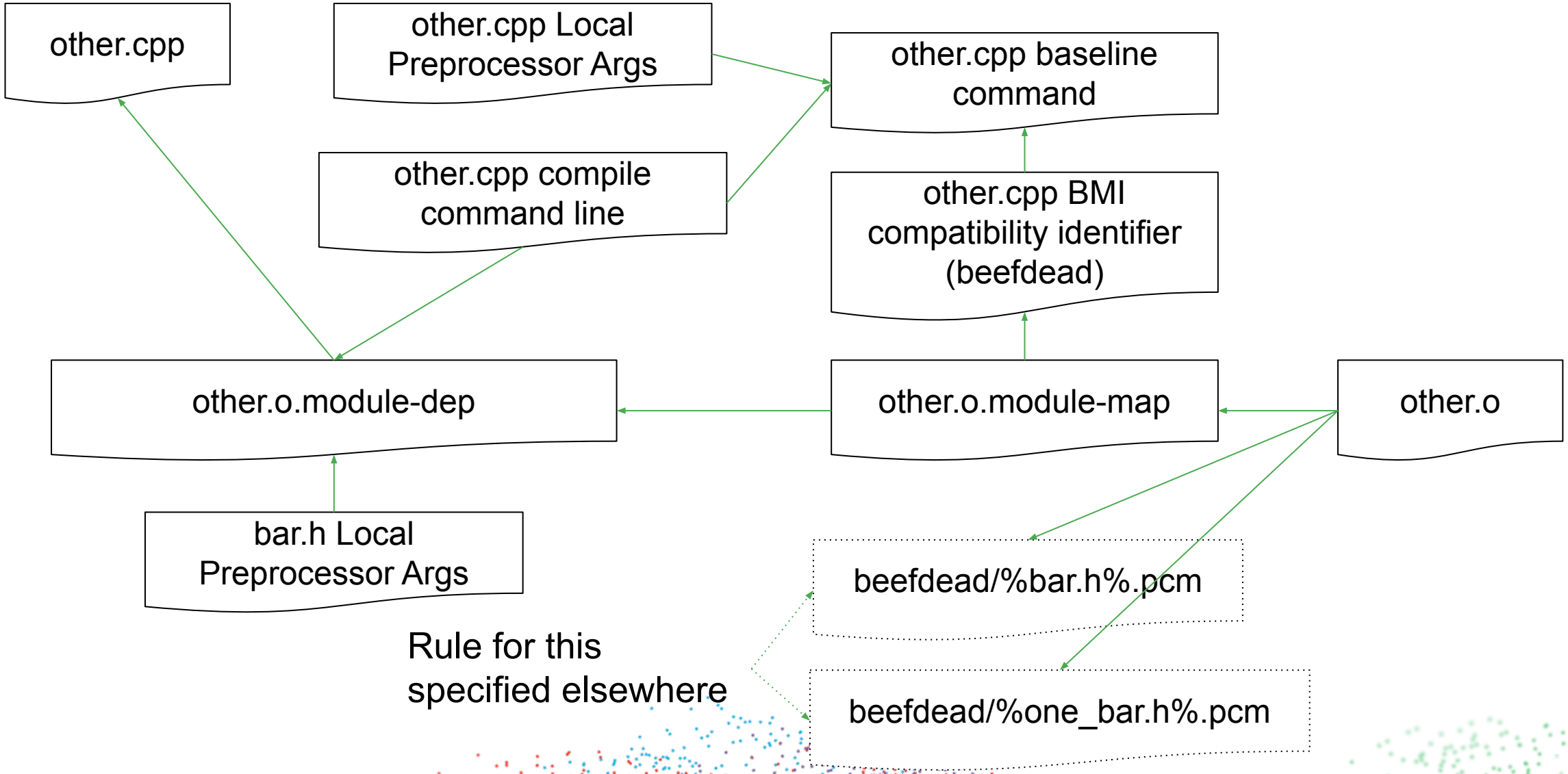
```
1  #define BAR_NUMBER 1
```

```
1  #define BAR_NUMBER 2
```

The command line for the header unit defines the starting state

- This is where it is fundamentally different than named modules
- With named modules, the compile command of a module you imported has no bearing on your dependency graph
- With header units, the “Local Preprocessor Arguments” influences the dependency graph

Generate BMIs and Compile Objects



But wait, how do I know I need bar.h?

- Right, this happens before the dependency scanning
- The answer is… you don’t

Really important takeaway

- The list of header units and the local preprocessor arguments of every header unit are an input to the dependency scanning process
- Any change on the list of header units, or on the local preprocessor arguments for those will cause a full rescan of dependencies
- Some build systems may know how to avoid redoing downstream targets if the output is identical… `make` does not

It gets trickier

- The standard allows an “#include” to be transparently replaced by an “import”
- Therefore, you may think you know the preprocessor state at the beginning of the source inclusion
- But, if the configuration told the compiler this is an importable header, it may be completely different

Identity of the Header Unit

What is the thing being included?

- In source inclusion, the compiler will do the search based on the “include” path and move on
- With import, the source for the importable header may not even be on the “include” directory
- If the same source is included more than once, we hope you use “include” guards
- This question is part of the answer as to why “pragma once” never made into the standard

The identity between “include” and “import”

- There’s currently no convention to make sure “include” and “import” point to the same “thing”
- This will likely become a major source of confusion for developers in large codebases

A contrived, but realistic, example

```
$PREFIX/include/
  dent/
    bar.h
  hmm -> dent
  bla/
    baz.h -> ../dent/bar.h
```

```
1  #include <dent/bar.h>
2  #include <hmm/bar.h>
3  import <bla/baz.h>
```

What behavior do we expect?

Takeaway

- It’s really hard to specify the identity of a header
- Importing a header that was already included will result in the entities appearing duplicated in the translation unit

Misleading performance costs

Bloomberg

Engineering

A header unit exports everything

- Early implementations of named modules are showing that pruning the BMI of unreachable syntactic entities will be necessary for the expected performance

- For header units, everything is reachable

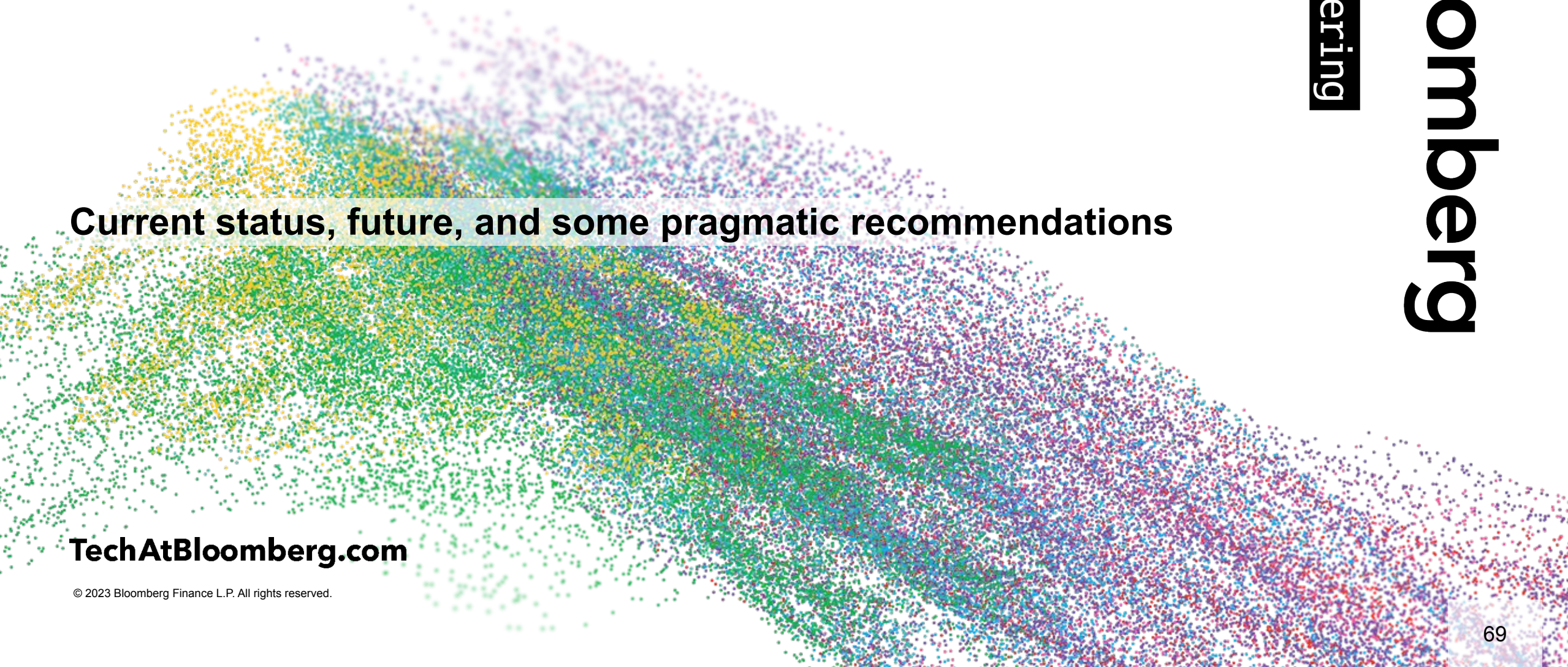
Include guards

- Include guards across header units have no effect, as each header unit is translated from scratch
- If two header units include a lot of the same non-header-unit headers, the duplication may actually result in worse performance
- Header unit adoption has to start from the bottom up, to allow that deduplication to happen
- Using header units without taking this into account will actually lead to worse performance

Where does that leave us?

Bloomberg

Engineering



Current status, future, and some pragmatic recommendations

TechAtBloomberg.com

How much support is there for header units?

- No implementation has header units fully supported
- GCC and Clang barely have any support
- MSVC has some support, but not with the semantics described here
- No open source build system seems to have support for header units

Isn’t this the same as pre-compiled headers?

- No
- None of the restrictions that make pre-compiled headers work are in the specification

Isn’t this the same as Clang header modules?

- No
- Implicit Clang header modules work on some environments, but can’t be used with remote execution
- Explicit Clang header modules work on environments where everything is fully normalized, we have reasons to believe it wouldn’t be successful at all on open-ended environments.

A realistic workflow for header units

- A filesystem-based lookup mechanism for the metadata for a header unit would allow the compiler to find those without having to be given all arguments for all header units

- Removing the implicit conversion from “#include” to “import” would remove the need for the dependency scanning to be given the list of header units

Is that worth implementing?

- Requiring changes to existing libraries for them to deploy new metadata so they can be imported as header units seem to defeat the purpose; the library author can just offer a wrapper module instead

- The emulation of the import in the dependency scanning has unknown performance characteristics at scale

- A human trying to understand the preprocessor state will have to do the same emulation the compiler does (Raise your hand if you want to teach this to every current and future C++ engineer?)

What's the alternative?

- Work with library maintainers so they ship a modular interface
- Create a wrapper module library otherwise
- If you need macros, work with library maintainers to move macros to a standalone header, then wrap everything else in named modules

```
1  module;
2  #include <mylibrary.h>
3  export module mylibrary;
4  export namespace mylibrary {
5     using SomeClass;
6  }
```

Final thoughts

- The specification from header units started from precompiled headers and Clang header modules
- It evolved into something very different from both
- It’s unclear that the requirements from implementers will result in something that achieves the original goals

My personal take

We should drop this feature from the specification and focus on named modules, and make the ecosystem around them great

TechAtBloomberg.com

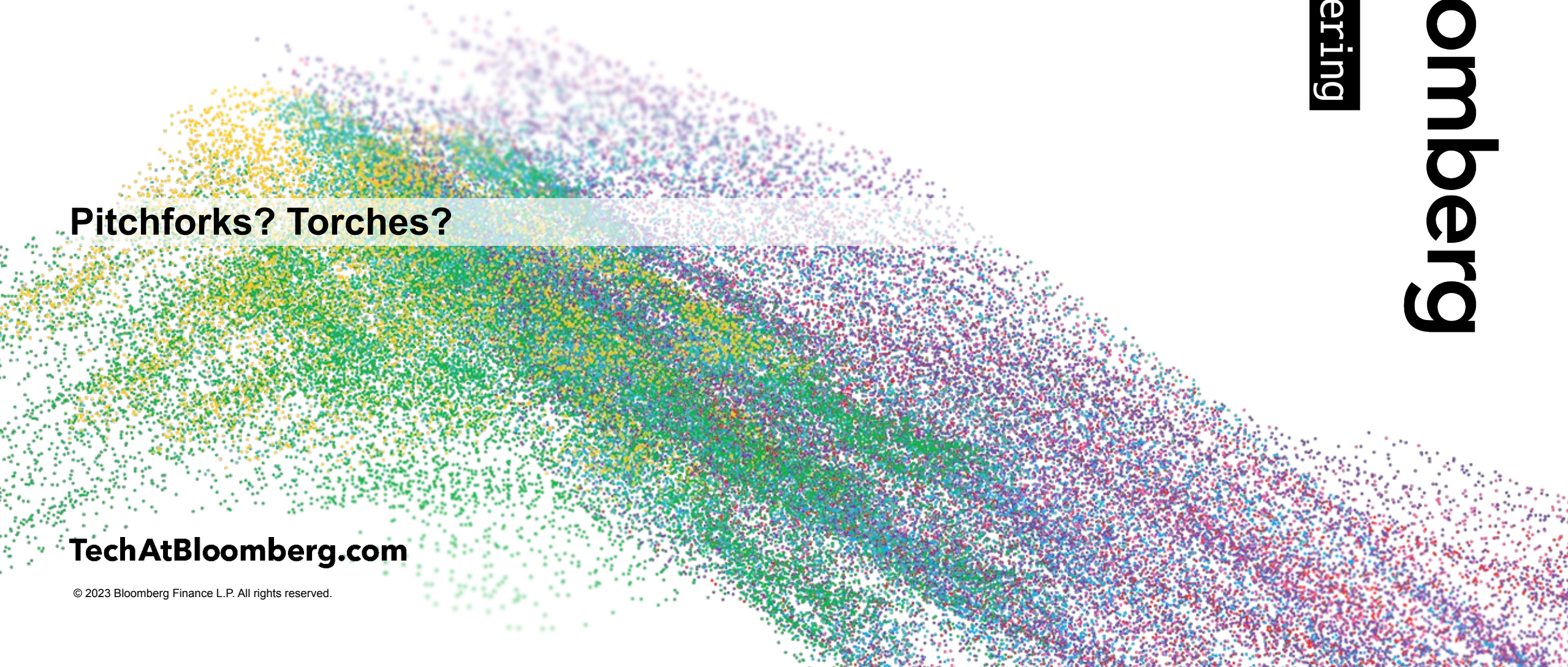
Bloomberg

Engineering

Questions?

Bloomberg

Engineering



Pitchforks? Torches?

TechAtBloomberg.com