Document Number: p2191r0
Date: 2020-07-10
To: SC22/WG21 EWG
Reply to: Nathan Sidwell
nathan@acm.org / nathans@fb.com

Modules: ADL & GMFs do not play together well (anymore)

Nathan Sidwell

The semantics of how dependent (instantiation-time) Argument Dependent Lookup interacts with Global Module Fragments of module interfaces on the path of instantiation are underspecified, complex and fragile. Some, probably unintentional, requirements are unimplementable. How did we get here, and how do we get out?

This paper presents a design change, restoring behaviour to an earlier draft of the modules-ts.

1 Dependent ADL in a Module World

A named module may export a template whose instantiation could rely on something from that module's GMF. How should an instantiation of that template, triggered from outside the module behave?

The working draft adds the following bullet to [basic.lookup.argdep]/4

> If the lookup is for a dependent name (13.8.2, 13.8.4.2), any declaration D in N is visible if D would be visible to qualified name lookup (6.5.3.2) at any point in the instantiation context (10.6) of the lookup, unless D is declared in another translation unit, attached to the global module, and is either discarded (10.4) or has internal linkage.
>
> [basic.lookup.argdep]/4.5

This makes non-discarded external linkage *global-module-fragment* entities visible. Such entities may be:

- Textually declared in header files included in the GMF
- Declared in header units imported from such header files

Only the former are subject to discarding, the latter may happen via include translation.

2 Background

The origins of this problem predate ATOM. Much of CWG's time at the Kona'17 meeting reviewing the modules specification was spent on it. Here is an example from Richard Smith given in those minutes:

```
// X.h
namespace Q {
   struct X { };
}
```

```
// Module M1
module;
#include "X.h"  // (global module)
// [At that time, the GMF could contain arbitrary code
//  Consider this being in a private header file]
namespace Q {
  X operator+ (X,X); // Private implementation detail
}
export module M1;
export template<class T>
void g(T t) {
 // [n4647's solution: add 'using Q::operator+;' here]
 t + Q::X();   // takes X from this TU and from the M2 TU
}
```

```
void j(Q::X x) {
  g(x);    // ok
}
```

```
// User code
#include "X.h"  // global module
import M1;
void h(Q::X x)
{
  g(x);    // ill-formed, can't find operator+
}
```

There was no desire to make the entirety of the GMF visible to such ADLs, as that would make Compiled Module Interfaces large – expected to often contain significant fractions of the standard library. The solution added in n4647:'Working Draft, Extensions to C++ for Modules' was to require M1 to bring `Q::operator+ (X, X)` into scope with a *using-declaration*. Nothing in the GMF is found via dependent ADL. Its change to ADL adds the following bullet to [basic.lookup.argdep]/4's list:

> – Any function or function template that is owned by a module M other than the global module (7.7), that is declared in the module interface unit of M, and that has the same

> innermost enclosing non-inline namespace as some entity owned by M in the set of associated entities, is visible within its namespace even if it is not exported.

Thus the customization point is located during definition parsing via unqualified name lookup, and retained for the eventual instantiation when it will be augmented with those found via ADL. This requires the module author to (a) know the sets of customization points of any global module template they use, (b) know that a particular type provides at least one such customization and (c) remember to place using declarations in scope.

It was an important feature of the TS that GMF entities were not findable by name lookup outside of the interface. This allows, for instance, the GMF inclusion of a header file, and exportation of a subset of that header file from the module.

2.1 ATOM

Since that time, the ATOM proposal, p0947:'Another take on Modules', was made, and subsequently merged with the TS in p1103:'Merging Modules'. ATOM, and its subsequent merging, introduced a number of new concepts. The ones relevant to this paper are:

- GMF Discarding. Entities declared in the GMF but not used in the exported interface are discarded.
- Partitions. A module interface may be organized into smaller source fragments. This organization is not visible from outside the module.
- Header-units. Header files can be turned into (unnamed) module units. Like the GMF, these are conceptually potentially overlapping views onto a Global Module.
- Include Translation. A #include directive may be treated as if it was an import declaration for the header it names. (That header must be available as a header-unit.)

It is apparent that the interaction of these new features with dependent ADL has not been fully resolved.

2.1.1 Partitions

A module may be separated into several partitions.

> Module partitions are an implementation detail of their containing module that is not observable to code outside the module. [p0947/3.3]

The partitions are distinguished with unique module-local names, but otherwise share the same structure as the primary interface. Only module units (including the primary) of the same module may

import partitions. The primary interface can, and indeed must, export the module's interface partitions. We now have the possibility of multiple GMFs visible from extra-module instantiations.

The standard specifies that the GMFs of the TU containing the template definition are of interest. But that exposes the different GMFs in the interface partitions. This is in conflict with the requirement that the decomposition of a module into partitions is invisible outside the module, and it hinders certain implementation techniques, discussed below.

The distinction between interface partition and implementation partition is not significant to this paper, except where explicitly discussed in Section 4.1.

2.1.2 Header-Units

Header units are an encapsulation of header files, achieved by some implementation-specified compilation mode. Import declarations may name header-units, using the same syntax as for #include directives. Header-units may be explicitly imported from a header file #included in a module's GMF (they may also be explicitly imported in the module's purview, just as a named module could be).

Such GMF-located imports are visible to dependent ADL, and thus, implementations cannot discard the import graph of a GMF, once a module interface unit has been compiled. Again, the above 'which GMF?' question arises in the form of 'which GMF's import graph?'.

2.1.3 Include Translation

To provide implementation flexibility, #include directives may be translated into import declarations. Whether this occurs is implementation-defined, and it is expected that implementations will provide options for the user to control this.

The original p0947 paper, and earlier version of p1103 specified that, if a header was include translated, it had to be include translated everywhere in the program. However, that requirement was relaxed as part of p1811:'Relaxing redefinition restrictions for re-exportation robustness':

> Previously, we required include translation in part in order to mitigate the effects of the problem that is solved by this proposal. In the simple example above, translation saves us from a redefinition error unless a definition in an imported header can also be found in a textual header.
>
> … At least one implementation vendor believes that migration to modules would be eased by permitting these two decisions to be decoupled. [p1811/3.1]

The background to p1811 was to resolve some issues with GMF entity discarding, which consistent include translation cannot resolve anyway.

In general though, a module author will have no control, or knowledge, over whether particular headers are translated or not. She may develop on a system where include translation occurs, and thus ADL

locates the necessary customization points because the import graph is used. When the module is built on a system that does not translate that way, the customization point is not found, as it was never explicitly mentioned in the module's purview.

2.2 Implementation Strategies

There are two implementation strategies of note.

2.2.1 GMF Entity Discarding

N4647 does not discuss discarding unreached global module entities, as the concept was unnecessary. Nothing in the interface's GMF was nameable outside of the interface. Program ODR correctness did not depend on whether the compiler noticed ODR-violating redefinitions.

The GMF entity discarding is specified in [module.global.frag]. Generally a GMF entity is not discarded if it is referenced (possibly indirectly) from the exported purview of a module. That specification explicitly permits implementation divergence by not specifying some constraints.

As reachability has a semantic effect on correct programs, it is unfortunate that this under-specification is required.

Every global module entity can be multiply declared and defined. Such multiple declarations and definitions must be resolved. This resolution is essentially the same as the resolution that occurs on textually parsing any declaration. One of the attractions of modules is to eliminate that kind of reprocessing, and not discarding unnecessary GMF entities would go against that goal.

Of course, with textual parsing, there can only be one definition in any particular Translation Unit. So resolving multiple definitions is a new kind of resolution. Implementations may choose to rely on the ODR, or they may check that ODR-sameness is present. In either case, this is checking or resolution that would not occur, if duplicates were not present.

2.2.2 Partition Invisibility

Module partitions are a source-code organization technique that is intended to be invisible to users of the partitioned module. Anything that exposes the partitions to importers breaks this.

With partitions, entities owned by the module may have multiple declarations. As with global module entities, these need to be resolved when when importing. Unlike global module entities, there can only be one definition though. Without partitions, there can only be one declaration of a module-declared entity seen by importers.[1] But if a module's primary interface imports partitions, there could be duplicates.

1 The case of a module interface declaring an entity and a (non-partition) module implementation unit defining it is not the difficulty being described here.

Extra-module partition invisibility allows an implementation to apply the declaration merging when producing the CMI of a primary interface. Importers do not need to apply any merging themselves. It also means that the importers only need the CMI of the primary interface, not those of partitions as well.[2]

This is not a theoretical implementation technique, it is at least deployed in the GCC modules implementation.

3 Current Wording

The current behaviour is documented as:

> If the lookup is for a dependent name (13.8.2, 13.8.4.2), any declaration D in N is visible if D would be visible to qualified name lookup (6.5.3.2) at any point in the instantiation context (10.6) of the lookup, unless D is declared in another translation unit, attached to the global module, and is either discarded (10.4) or has internal linkage.
>
> [basic.lookup.argdep]/4.4

For avoidance of doubt that last sentence should be read as ignoring discarded global module entities that were declared in (imported) named module GMFs (internal-linkage GMF entries are also ignored).

The specification of instantiation context includes:

> During the implicit instantiation of a template whose point of instantiation is specified as that of an enclosing specialization (13.8.4.1), the instantiation context is the union of the instantiation context of the enclosing specialization and, if the template is defined in a module interface unit of a module M and the point of instantiation is not in a module interface unit of M, the point at the end of the declaration-seq of the primary module interface unit of M (prior to the private-module-fragment, if any).
>
> [module.context]/3

4 Discussion

4.1 That's Not Implementable!

Let's get the impossible implementation requirement out of the way first. It is a small oversight.

Notice that when a template is defined in a module interface unit of M, but the point of instantiation is not in a module interface unit of M, a point on the path of instantiation is the end of the purview of the primary interface unit, just before any Private Module Fragment. This is ignoring the possibility that the template is defined in an interface partition, and the point of instantiation is in an implementation

2 This could also be achieved by making the CMI a container object.

partition. In that case, the end of the primary interface's purview is not available – it may not have even been compiled.

This can be resolved by specifying the that this applies when the point of instantiation is also not in an implementation partition of M. An alternative formulation would be that it applies when the point of instantiation is not in a module unit of M.

When the point of instantiation is in a non-partition implementation unit of M, the end of the purview of the primary interface (before any PMF) is in scope regardless of where the path of instantiation reaches.

4.2 Multiple GMFs

As mentioned above, when partitions are imported into the primary interface of a module, we have multiple GMFs. If we are to keep the axiom that partitioning is invisible outside of the module, I think there are two options regardless of how we make a particular GMF's contents externally reachable:

- The union of the individual GMFs' reachable entities is reachable to importers.
- The intersection of the individual GMFs' reachable entities is reachable to importers.

Of those, I think only the first is coherent. If the latter option was chosen, breakage could occur when a new partition is created, added to the primary interface, but it fails to make crucial GMF entities reachable.

Either scheme has difficulties for implementations that keep partitions as separate logical units within the compiler. This is resolved in the next section.

4.3 GMF & Name Lookup

The intent (of what became) the Global Module Fragment was to provide a module-private bridge from non-module code to module code. The headers #included in the GMF would not leak to importers of the named module. Importers of named modules would continue to need to #include headers they themselves needed.

Module purviews could explicitly make GMF entities visible to name lookup via using declarations, and that was the mechanism n4647 uses to make customization points visible in instantiations. This was changed via p1103 and the reachable GMF-located imports and non-discarded entities became significant. Of course local using-declarations would continue to behave as before.

That GMF-located imports are significant, means that implementations must retain that set, separate from the module-purview imports of an interface. As mentioned in Section 2.1.3, what this set is exactly, is not easily determinable.

GMF entity discarding has been discussed in Section 2.2.1 as a practical measure to reduce CMI size and importation complexity. Its primary goal is to specify that GMF entities may become known to the compiler, but not nameable by the users' code. Should the user declare that entity (textually or via import), the declarations must match. The user may provide a (duplicate) definition, iff the entity may have multiple definitions within a program (e.g. is a class, template or inline) and no definition is already available *to the user*. Any redefinition must be the same as that already known to the compiler.

GMF entity discarding is essentially blessing an implementation technique, and putting the user on notice that the compiler may know more than the user can reach. But, as non-discarded entities become nameable to dependent ADL, it affects user code using a module. The fragility of that exposing has been described in Sections 2.1.3 & 2.2.1.

Unlike the situation when n4647 was produced, we now have importable header units. We could return to n4647's behaviour of the GMF entities not being visible to (any) name lookup outside that interface.

A named module that requires the availability of particular global module entities during instantiations could explicitly import the header unit in its purview. Such an in-purview import makes clear that the module is relying on the imported header unit in its own interface. Alternatively, placing the import in the module's GMF would be making it clear the header unit is an implementation detail of the module interface. The same would also be true for imports of named modules. (It is unfortunate that placing explicit imports in the GMF is now tricky, as only non-import preprocessor directives are permitted.)

Users who cannot, or do not wish to, use an imported header-unit may use the n4647 technique of adding local using-declarations to their template definitions.

Notice that this renders moot the previous section's discussion of merging the GMF entity or import sets of module partitions – they are all the empty set.

There is no longer a need for all users to rely on fragile 'do not discard' touching of global module entities to make them available to dependent ADL.

The rule is also simpler than the current standard, it is returning to N4647's simple 'entities declared in a module TU's GMF are not nameable outside of that TU'.

4.4 Top-level GMF Contents

A module's GMF permits an interface (or partition) to import entities that are not also made available to module implementation units. Unfortunately the GMF is required to be constructed of non-import preprocessing directives at the top level (before phase-4 of translation). This restriction was introduced in p1103 so that code analyzers (and humans) can easily find the module-purview starting declaration.

Since that time, p1703:‘Recognizing Header Unit Imports Requires Full Preprocessing’ & p1857:‘Modules Dependency Discovery’ have made import-declarations recognizable by the preprocessor at phase-3 of translation as pp-imports.

We now have a containment mechanism that is only available for non-modular interfaces, that explicitly carves out pp-imports from other preprocessing directives. Further a top-level GMF-located include directive may be translated to an import declaration, but the import declaration it translates to cannot itself be placed in include directive’s location.

5 Proposal

There are three changes proposed, which I categorize as:

- 5.1 a Defect Report. In isolation this would have simply been reported to CWG.
- 5.2 a Design Correction. This removing some semantically-signficant implementation-defined behaviour and returning to an earlier scheme.
- 5.3 a New Feature. Adding consistency between importable header-units and includeable header files.

For avoidance of doubt, the first two changes are much more critical than the last.

5.1 Repair Unimplementable Instantiation Point

The path of instantiation needs rectifying to not (necessarily) include the end of the primary interface, when instantiating from a partition.

5.1.1 Wording

Modify [module.context]/3:

> During the implicit instantiation of a template whose point of instantiation is specified as that of an enclosing specialization (13.8.4.1), the instantiation context is the union of the instantiation context of the enclosing specialization and, if the template is defined in a module interface unit **or partition** of a module M and the point of instantiation is not in a module ~~interface~~ unit of M, the point at the end of the declaration-seq of the primary module interface unit of M (prior to the private-module-fragment, if any).

5.2 GMF Becomes TU-local

Stop allowing any entities introduced in a module unit's Global Module Fragment to be visible to any lookup that occurs after that module unit has been compiled. This removes the complexity from dependent ADL. There is no longer any need to retain those entities in per-module symbol tables, nor is there a need to retain a separate import list, solely for that ADL.

GMF entity discarding becomes purely an implementation detail, with no semantic effect on well formed programs. When GMF entity discarding is an implementation detail, code analyzers have the freedom to retain GMF entities, and ensure the ODR is not violated by importers defining conflicting declarations. This has the effect of making the ODR more consistent to users.

This removal of GMF entities from post-unit compilation lookup renders moot the discussion in Section 4.2. If nothing from them is nameable, it doesn't matter which GMF is considered, or how they are combined – nothing is in any of them.

Module authors that need to make Global Module customization points available to dependent ADL of their exported templates will need to add imports of the appropriate header units in their module's purview. Just as they would naturally do for imported named modules' customization points.

The relaxing of include translation by p1811, and the associated desire to make that optional will no longer affect which entities might be visible to dependent ADL. If a module author must rely on a textually included header, she can require her users to textually include the same header – just as they would have to do with n4637-era modules-ts, or she can add using-declarations, as n4647 module-ts suggests.

5.2.1 Wording

Modify [basic.lookup.argdep]/4.4

> If the lookup is for a dependent name (13.8.2, 13.8.4.2), any declaration D in N is visible if D would be visible to qualified name lookup (6.5.3.2) at any point in the instantiation context (10.6) of the lookup, unless D is **only** declared in**, or made reachable by a module-import-declaration in, the global-module fragment of** another translation unit~~, attached to the global module, and is either discarded (10.4) or has internal linkage~~.

5.3 Relax GMF Syntax

As mentioned above, we currently have a mechanism by which a module interface unit may make global module entities interface-local, but we do not have a mechanism to do the same for named-module entities. We should relax the restriction that the GMF contents consist solely of entities brought in via preprocessor directives.

5.3.1 Wording

[module.global.frag]/1 does not need modifying, as it already permits module-import-declarations:

> [Note: Prior to phase 4 of translation, only preprocessing directives can appear in the declaration-seq (15.1). — end note]

Modify [cpp.pre]/3

> At the start of phase 4 of translation, the group of a pp-global-module-fragment shall **not** contain ~~neither~~ a text-line ~~nor a pp-import~~.