Unspecified Behavior in KerML Name Resolution: Issue, Consequences, and a Proposed Solution

Author: Sensmetry (SysIDE developer)
Contact person: vytautas.astrauskas@sensmetry.com

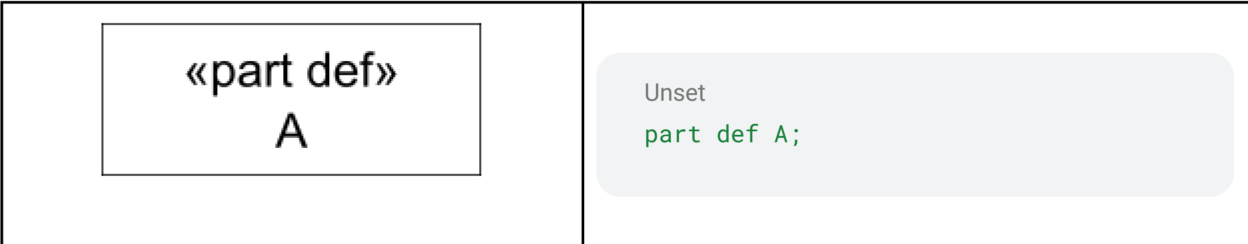
Introduction

In SysMLv2, name resolution is ambiguous (known as ‘unspecified behavior’ in programming language design). In this document, we discuss the negative consequences of this design choice and propose a solution resolving them without affecting the end user experience and only requires minimal changes to the current KerML, SysML v2, and APIs & Services specifications.

This document is structured as follows. First, we remind the reader what the name resolution is and show how unspecified behavior appears. Second, we discuss the far reaching negative consequences of this unspecified behavior. Then, we present our solution proposal, and, finally, list the specific changes required to the specifications to implement the proposed solution.

Issue description: unspecified behavior in name resolution

Name resolution is an algorithm that resolves (finds) a concrete element based on its name. For example, if user 1 defines a part definition `A`:



And user 2 defines a part `x` of type `A`:

<table>
<tr><td></td><td>Unset<pre>part x: A;</pre></td></tr>
</table>

The name resolution algorithm resolves `A` in the definition of part `x` to the element `part def A`. The name resolution is such a fundamental part of user experience that most users do not even think about it and this is exactly the reason why it must work without any surprises.

Unfortunately, the name resolution, as it is currently defined in the specification, is ambiguous (unspecified behavior). Consider user 3 who works in a different team than user 1 and is, therefore, potentially unaware of part definition `A` (or could have simply forgotten that part definition `A` exists, which is not unlikely if the model is large). If user 3 defines a new part definition `A`:

<table>
<tr><td>«part def»
A
...</td><td>Unset<pre>part def A {
  // ..−.
}</pre></td></tr>
</table>

Then, `A` in the definition of `x` will become ambiguous: it may be resolved either to user 1 or to user 3 definition.

That such ambiguous behavior is allowed is explicitly stated in KerML subclause 8.2.3.5.4 Full Resolution:

Unset
```
Note. It is possible that there will be more than one Membership in the global
Namespace that resolves a given simple name. In this case, one of these
Memberships is chosen for the resolution of the name, but which one is chosen
is not otherwise determined by this specification.
```

In the next section, we discuss the negative consequences of this design decision.

Downstream consequences of the unspecified behavior

Deterministic name resolution is critical for supporting workflows that SysML users expect. In the following subsections we give three examples of workflows that are either completely blocked or significantly impacted by ambiguity in the name resolution.

Distributed collaboration on models hindered and hurts SysML v2 ecosystem

One hope for SysMLv2 is that it will enable distributed development workflows that are common in programming languages. Such workflows include:

1. Using optimistic version control to enable a group of developers to work on a single software project concurrently.
2. Encapsulating functionality in libraries and reusing them.
3. Using package management systems such as npm, maven, pip, conda, cargo to efficiently share and integrate libraries.
4. Using access control on software packages to control access to the software.
5. Effective communication of effort required to update a version of a used library based on changes in its semantic version.

Importantly, the effectiveness of all of these workflows completely depends on the reliability of the name resolution. For example, if two developers add a function with the same signature but different implementation in a large code base, and the compiler and linters do not catch that, it is extremely painful and time consuming to find why the tests fail. We are, of course, assuming that the developers use the best practice of covering all their code with tests. If they do not, or if the language defines redeclaring the function as unspecified behavior, the bug may surface only much later (potentially only in production) making debugging much harder and significantly more costly.

If the name resolution remains ambiguous, all of these workflows are likely to be too painful to use to be practical.

Using SysML for communication may lead to misunderstandings

An important use case of a modelling language is to enable one party to precisely communicate the system design to another party. Since the communication ultimately happens between humans who read names instead of element IDs, it is crucial for the name resolution to be deterministic for the two parties to have the same understanding of the system

Verification results may be invalid

Another important use case of a modelling language is to enable the users to verify a model of a system before building it to significantly reduce the cost of production. Ideally, the model would be verified continuously in a CI/CD pipeline significantly reducing the time it takes to get feedback and iterate on new designs while at the same time ensuring high quality. Unfortunately, due to the ambiguous name resolution, the verified model may not match the model used to produce the system thus making the verification results void.

One way a mismatch could happen is when name resolution for the model itself is performed before each phase. Another way a mismatch could happen is when the verification harness itself uses name resolution (because the human setting up the harness uses names and not IDs

to refer to elements) to determine what to verify and that resolution happens to resolve another element than the intended one.

Note: In some domains, the SysML users may be required by regulatory bodies to ensure that their verification results match the produced systems. Therefore, the vendors have a strong incentive to fix the ambiguous name resolution problem. If we do not provide the fix in the specification, there is a large risk that each vendor will fix the problem in their own ad-hoc manner and break the interoperability between the tools.

Proposed solution: introduce hierarchical project namespaces to avoid duplicate names

While we could just add a validation rule that forbids duplicate names, it is of little use to the users if they have no way of fixing the error. Therefore, we propose a solution that reduces the likelihood of duplicate names occurring and provides the users a way to fix errors in case they do.

In SysML, the global namespace is effectively a union of root namespaces. With the current design, the global namespace will grow linearly with the size of the model making the duplicate names unavoidable without tedious effort. This fast growth happens because of two reasons. First, if a model depends on a model interchange project, the model's global namespace includes all the root namespaces of the interchange project. As a result, the model's global namespace grows linearly to the number of projects used. Second, as reported in KerML issue 43 ("Root namespaces restricted to one single file"), each root namespace is limited to a single file making the number of root namespaces grow fast for models that use textual notation. For example, the standard model library contains 93 root namespaces, which already makes it tedious to navigate and ensure absence of duplicate names.

When searching for a solution, we aimed to satisfy the following requirements:

1. Require minimal changes to the specification since we are running out of time.
2. Does not add additional restrictions on the implementation, for example, a requirement to store textual notation in a specific file structure.
3. Replace ambiguous name resolution with a clear error or deterministic and easy-to-understand behavior.
4. Significantly reduce the risk of running in name conflicts.
5. In case a conflict occurs, enable the user to resolve the conflict without modifying the dependencies (this requirement is crucial in cases when one company buys an off-the-shelf model from another company and tries to integrate into their model).

We present our proposed changes to these two causes separately. However, it is important to note that they depend on each other.

Change 1: Introduce projects explicitly and use them in name resolution

We propose to introduce a project as an explicit concept in KerML and SysML specifications. Currently, in KerML and SysML, the term project is either used in informal examples or to refer to model interchange projects defined in KerML 10.3. Explicitly introducing this concept enables us to avoid ambiguous name resolution as we describe below and bring KerML and SysML specifications closer to the API specification in which the concept of a project is very central.

Introducing the concept of a project enables us to talk about project dependencies in the description of the name resolution algorithm. More specifically, we propose two changes to the name resolution:

1. Prepend project names to the qualified names of the elements coming from dependencies. For example, to use the part `Wheel` from the project `WheelShop`, one would need to write `import WheelShop::Wheel`.
2. When resolving a project name, consider only projects that are direct dependencies.

This change should already significantly reduce the possibility of a name clash because we only need to worry about having two projects with the same name. However, it does not allow the user to resolve the problem in case a clash occurs (our requirement 5). Therefore, we propose to enable users to specify local names to the dependencies. This change would have two benefits:

1. Enable the users to have names that are potentially more meaningful in the current project than the ones given by the original author.
2. Enable the users to resolve the clash in case two dependencies have the same name.

Also, for technical reasons, we limit the number of root namespaces per project to one (each element in the qualified name must be a membership, and the global namespace cannot be a target of a membership). This modification has an advantage that it makes the semantics of the namespaces cleaner by removing the global namespace with its special rules; we address the drawbacks with Change 2. Adding this restriction has an important consequence that the name resolution becomes deterministic because duplicate names are forbidden within the same root namespace.

Note: We propose to keep the old behavior for the standard library to keep the number of changes we need to make to the specification to the minimum.

Change 2: Introduce extern packages to enable hierarchical structure in textual notation

The previous change fixes the ambiguous name resolution, but it requires textual notation to support projects with a single root namespace. Therefore, we propose to introduce the concept of extern packages that is inspired by the Rust module system.

In Rust, modules (that correspond to packages in SysML) can be written in two ways:

1. Inline modules use the same syntax as SysML packages:

```
Unset
mod foo {
      // The body of module foo.
}
```

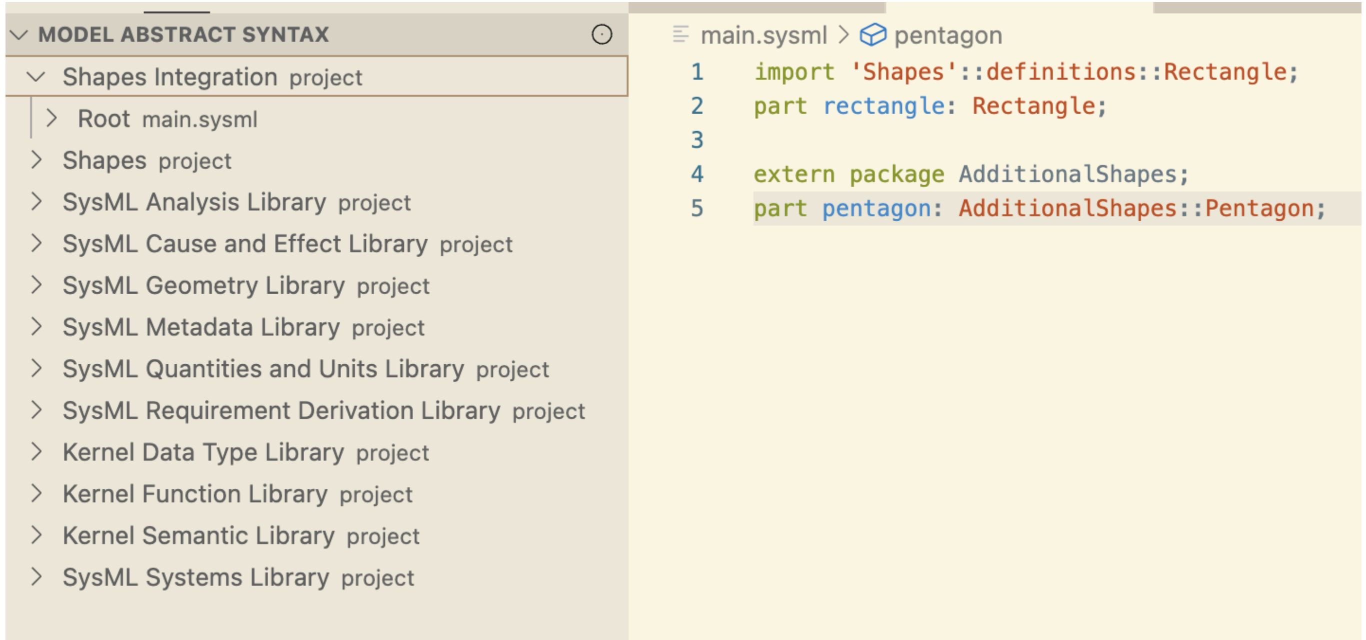
2. If the module is defined with a semicolon (`mod foo;`) instead of braces, Rust compiler uses an external file as the body of module `foo`. More specifically, it tries the file `foo.rs` and if it does not exist, tries `foo/mod.rs`.

Since syntax `package foo;` is already used for empty packages, we propose to add an additional keyword `extern` to indicate that the package's body should be loaded from a different place. We propose to leave it implementation defined from where exactly the textual notation is loaded: this flexibility avoids mentioning file system before KerML 10.3 and enables us to support this feature even in textual notation that is nested inside graphical notation. The only place that would precisely define how extern packages are resolved would be KerML 10.3 since that is needed for interoperability. For interoperability, we would also need to update Interchange Project Metadata to include a field that lists the project's roots since it cannot be assumed anymore that every file inside a `.kpar` file is a root namespace.

The key benefit of this design compared to the alternatives mentioned in KerML issue 43 is that it works completely on the parser level and has no effect on the abstract syntax thus keeping the required changes to the specification to the minimum.

Prototype Implementation

At Sensmetry, we have implemented the proposed solution as a prototype in SysIDE CE, an open source SysML v2 editing and analysis sytem. The following screenshot shows the key elements of the updated experience.



- The panel "Model Abstract Syntax" is similar to the "Outline" panel in the Pilot implementation with the main difference that it shows the current project (`Shapes Integration`) and its dependencies.
- As can be seen from line 1, to import an element from project `Shapes` we need to use the project name as a prefix.
- When parsing `extern package AdditionalShapes` on line 4, SysIDE CE tries to load its content from file `AdditionalShapes.sysml` and if that fails loads it from file `AdditionalShapes/root.sysml`.
- As can be seen on line 5, elements from extern packages can be used in the same way as from regular packages.

The source code of the prototype implementation is available to be shared with any interested parties and is intended to be made available as part of the SysIDE project.

Required changes to the specifications

Changes to KerML

On a high level, we need to make the following changes:

1. Change 1:
    a. Define a project as a concept early in the document. The project must have the following attributes:
        i. The list of root namespaces.
        ii. Project usages (dependencies) with their names.

b. Limit the number of root namespaces per project to one, except for the standard library.
c. Change the name resolution algorithm to use the name of a used project as a name of the root namespace. We can achieve this by adding a membership to the current project with the name set to the name of a used project and target set to the root namespace of the used project (this membership is the reason why we need to limit the number of root namespaces to one).
d. In 10.3, add field `roots` that lists the files containing the project roots to interchange project metadata.
e. In 10.3, add the field `name` to `InterchangeProjectUsage`.
f. In 10.3, change the phrasing that root namespaces are only in files listed in the root namespaces field.

2. Change 2:
   a. Introduce a keyword `extern`.
   b. Specify that when parsing `extern package P;` the **parser** loads a resource corresponding to `P` and parses it as the body of the package `P`. Note: it is important that it is left to the tool to decide which resource to load to not tie the implementation to a file system (currently, textual notation is not related to the file system).
   c. In 10.3, specify which `.kerml` file is loaded when parsing `extern package P` in a KerML file.

Changes to SysML

In SysML, we just mirror the changes from KerML.

Changes to Systems Modeling Application Programming Interface (API) and Services

On a high level, we need to make the following changes:

1. Add an explicit list of root namespaces to `Project`. For non-standard library projects this list is required to have a single element. This list is returned by `getRootElements`.
2. Add field `name` to `ProjectUsage`.