Class diagram explanations:

The following are currently only explanations for the planned class diagram of the application. They are not necessarily in use at the moment and simply reflect our thoughts on how certain properties of the class diagram should be handled.

Possible approaches for objects:

1. A generic object with ALL variables where the combination of specific variables defines which kind of object it is.
2. A generic object with SOME common variables where specific classes defines which kind of generic object it is.

   Approach 2 was chosen in the project!

NOT SUPPORTED:

NOT SUPPORTED means that a specific definition in the SysML v2 is not implemented in its entirety or graphically since the 2020-12 release.

Keyword synonyms:

Only use keywords rather than symbols when possible. Decide on ONE keyword for an object. I.e DO NOT use synonyms other than when translating a document when input to the application (ex block will be translated to part def when you input the file).

Connections:

When the parser finds a connection in the form of "*connect … to … => ...*" it should translate it to a "*connection def*" and a "*connection: ... connect … to …*".

Searching of non-defined objects:

When the parser finds an object that doesn't have a definition ex "part eng : Engine", it will search the entire document for a definition, ex "part def Engine", and then create the definition object first before creating the non-definition object.
NOTE: *This approach is not taken by the parser at the current version of the application! The current parser simply ignores if a definition exists or not!*

Instances:

The array instances[] is an array that contains all instances of a type of definition. Example all instances of Engine in the following part usage “part eng1 : Engine”, “part eng2 : Engine” will be put in the Engine instances[] array as follows: [eng1, eng2...].

instanceOf:

The instanceOf variable contains the object that another object is an instance of. Example “part eng : Engine” will mean that for the instance part eng it will have its instanceOf variable set to Engine.

Importing:

The import variable can be used by the GenericClass to keep track of imports.

References:

The reference variable can be used by the GenericClass to keep track of references.

Specialization:

When specializing an object, clone the object definition that is to be specialized and then add any additional features ex. extra parts or attributes. This seems to be the same as “subset”.

Subset:

See specialization.

State transitions and their names:

It is necessary to keep track of the correct indexes of the “to” StateClassArray() and the “transitionNames” StringArray() so that the correct names are paired with the correct “to” states.

Timeslices:

Name is confusing because there is no specification of the amount of time that it lasts. What makes it differ from just an individual?

Redefine:

Calls a function somewhere to replace values in an object with the new values/properties.

Parent:

When referring to a parent in the GenericClass the parent refers to the SysML object that encapsulates the target SysML object. An example:

```
package Package {

     part partUse : PartUse;
}
```

Here the parent of the partUse object is the Package object. The parent of the Package object is null since it is on the top-level.

Children:

When referring to children in the GenericClass the children refers to all the SysML objects a target object encapsulates. An example:

```
package Package1 {

    part def OuterPart1 {

        part def InnerPart1 {...}
    }
    part def OuterPart2 {

        part def InnerPart2 {...}
    }
}
```

Here the children of Package1 are [OuterPart1, OuterPart2], the child of OuterPart1 is [InnerPart1] and the child of OuterPart2 is [InnerPart2].

Comment and documentation:

The comment array and documentation variable references the “comment” and “doc” keywords in SysML v2. These contain the comments and documentation that are placed within a certain SysML objects code block.

Parser:

Might use a parser generator:

+ No need to implement a correct and fully functioning parser by ourselves
+ Only need to determine the grammar
+ Relatively easy to continue working on (for future developers)

- Need to learn the parser generators grammar
- No fundamental understanding of how the parser works
- Less flexibility
- Might be harder to connect to the rest of our program

Copying previous projects parser:

+ Faster solution, will probably develop farther during this project course
+ Easy to change to fit our definitions
+ No need to learn grammar for a generator

- Might not be as consistent in implementation
- Not as scaleable
- Bad documentation
- Might not work properly

Antler

+ Left recursive
+ Large amount of grammars available

- Divided in two parts: lexer rules and parser rules, might be complicated

Nearley

+ Left recursive
+ Should never catastrophically fail

- Slow
- Less resources

PEG.js

+ Probably easiest
+ Probably fastest to implement in terms of installation etc

- No tutorials
- Not left recursive

Graphics:

We will move from the use of konva.js framework previously used for the SysML project. This is because we think there is potential for a more specific and better suited framework for SysML diagrams than what konva.js can offer.

We are using a framework for the diagrams:

- mxGraph [first main option]
    - Used in draw.io, a page we are very familiar with that definitely contains the assets we need for SysML diagrams.
- jointJS
    - Built in UML support.
- goJS
    - Seems to support advanced user interactivity, might be nice.
- Paper.js
    - Looks very cool but not very applicable for us

We are using the Eclipse plugin (PlanUML) to decide how to draw the different types of diagrams. Depending on how far implemented a specific diagram is in the Eclipse plugin we might have to make our own decisions.