# System Behaviour Description Language

SBDL (pronounced ‘speedle’) captures the essence of system composition and behaviour in a simple yet powerful domain-specific language, allowing that essence to be placed close to where it matters. [HTML] [PDF]

contact@sbdl.dev

1.13.1 (SBDL Compiler), 25.6 (SBDL Language) [#694749b]

## Contents

*Interested in a short introduction to SBDL?* Have a look at this high-level overview presentation.

## Overview

SBDL models system behaviour as a set of related, typed elements, where each element describes a particular facet of system design and the specified relationships between those elements constitute overall system behaviour.

***The language...***

```
sbdl_purpose is requirement { description is "Let's make defining system models quick and easy" }
```

The ‘System Behaviour Description Language’ (SBDL) is a Domain-Specific Language designed for the terse and locally-expressed attribution of system behaviour by way of minimalist Domain-Specific Model. More elaborately put: SBDL allows for the engineer to define key behavioural properties of a system, the relationship between then, and to place those properties close to where their definition is realised. To this end, SBDL is intended to annotate existing design and development materials; this is in contrast to the creation of separate (and often unrepresentative) design materials.

Typical usecases for SBDL include:

- Capturing the design of a system as a light-weight, verified model
- Embedding and coupling model information close to where it is defined/realised
- Easily applying change and revision control to the model (e.g. Git et al)
- Automating the generation of design output material
- Integrating design risk analysis (FMEA) with model evolution

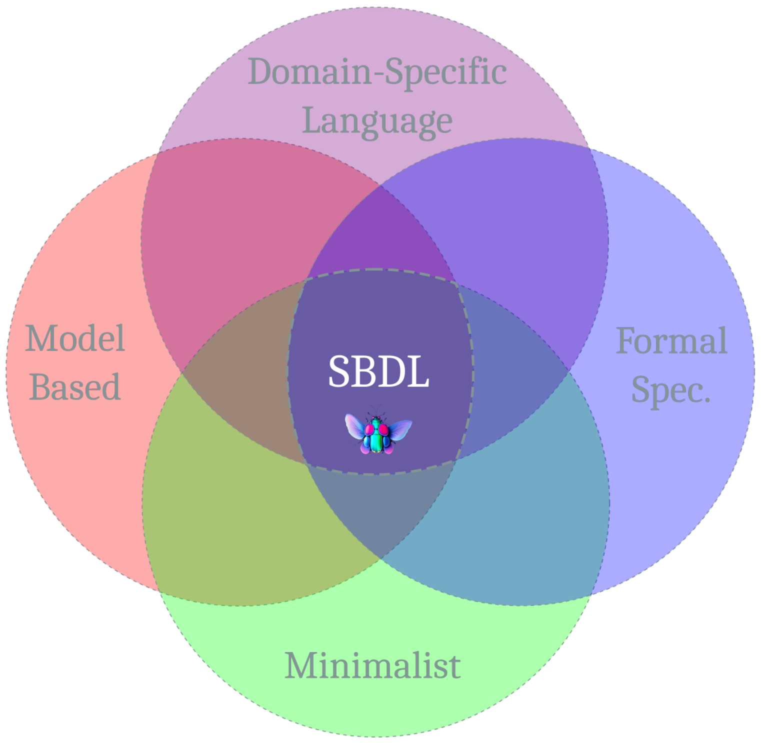


Figure 1: SBDL Venn Diagram of Influences

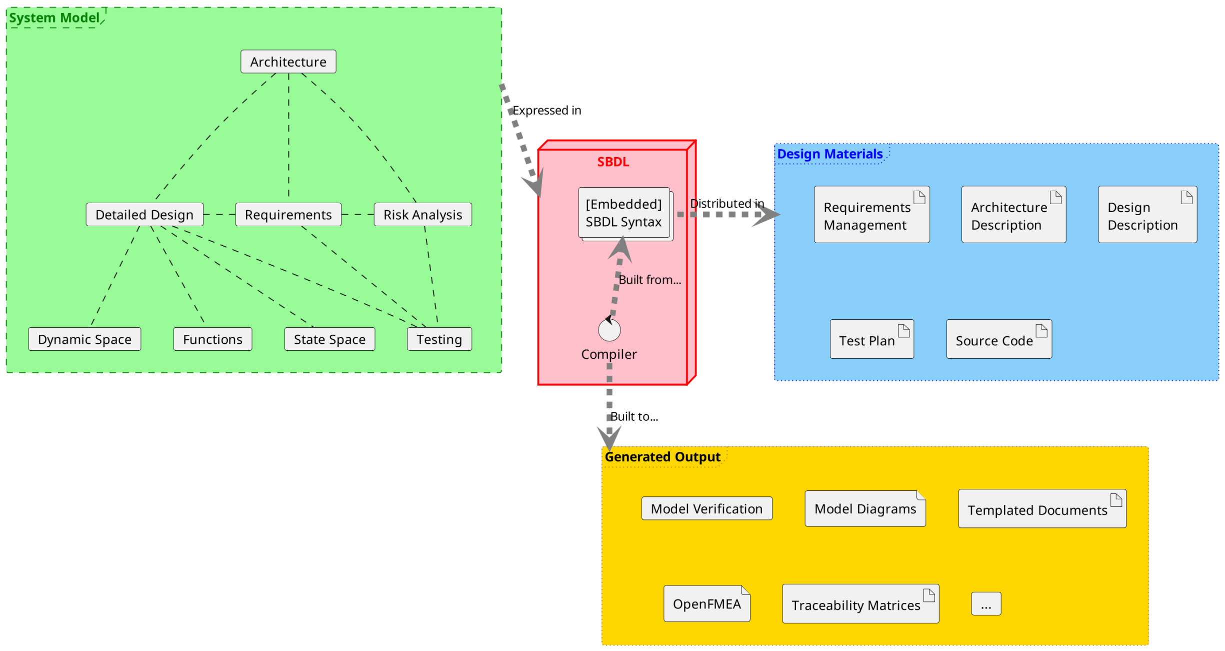


Figure 2: Overview of SBDL

In the near future:

- Dynamic trace model verification: verify a set of dynamic log trace elements against a model description
- Inclusion of more formal aspects, such as OCL, process algebras…
- AI scrutability: use SBDL to capture the design structure an decisions made by artificial intelligence when realising complex implementations
- Automate design review and risk analysis: describe a design in SBDL and have an AI system offer an automated design critique and risk analysis proposal

***The compiler…***

The SBDL Compiler allows the engineer to *extract and verify* the aforementioned behavioural model, and also generate various different output materials, including:

- architectural models (UML/SysML),
- requirement models (SysML),
- function models (UML/SysML)
- state models (UML/SysML)
- usecase models (UML/SysML)
- Failure Mode and Effect Analysis (OpenFMEA),
- traceability matrices (including test coverage)
- template-defined documents (Jinja) and more.

Using the SBDL compiler, behavioural annotations can be placed in almost any form of design material where textual-input is allowed, including: source code, markdown, word processor documents, issue management systems, and so on. Such embedded definitions can also be coupled with syntactic artefacts of the containing language or material.

In addition, the compiler is extensible, such that users may make programmatic use of SBDL definitions in their own applications.

## Quick Start

Looking for a syntax introduction? Or perhaps a reference of the Metamodel?

To get started with SBDL, you'll need at least the SBDL compiler and its dependencies. It's also useful to fetch and build the example project; both to familiarise yourself with the expressive form of SBDL, and also to check that your environment is working correctly.

### Get the compiler

**Prerequisites** The SBDL compiler is a Python project, and therefore requires that Python is installed*. The SBDL compiler itself is cross-platform (Linux, Windows … ).

**Install SBDL Compiler** The SBDL compiler itself can be installed using PIP:

```
python -m pip install https://sbdl.dev/sbdl-package.tar.gz
```

This will install the development version of SBDL and its dependencies.

You can then execute the SBDL compiler as follows:

```
python -m sbdl -h
```

The '-h' switch above displays the help output.

If the PIP installation location is in your path (as is the case by default on Linux) then calling the SBDL compiler can be further abbreviated to simply:

```
sbdl -h
```

***Standalone SBDL Compiler** *Currently the standalone build is available for Linux (x86-64) only.*

In case you'd like to simply fetch and use SBDL as a standalone executable and not as a python project, there is a self-contained build of the compiler available. This includes all necessary Python dependencies in a single file.

**Linux (x86-64)**

```
wget https://sbdl.dev/sbdl-standalone-linux-x86-64 -O sbdl
chmod ugo+x sbdl
./sbdl -h
```

**External Dependencies** Several of the output (UML) models use PlantUML for rendering and therefore require that it is installed and available in the path. PlantUML can be installed manually from the source or, as in most Linux installations, from a package manager.

In Debian/Ubuntu, for example:

```
sudo apt install plantuml
```

### Build the example

The SBDL example is a synthetic project contrived to demonstrate the basics of the SBDL language and its application within a build structure. It resides in its own repository: you can find the repository here.

To build the SBDL example you will need at least the SBDL compiler and CMake. In order to build the optional (but recommended) parts of the example, you will also need PlantUML and a C++ compiler.

The basic steps to build the example are as follows:

```
git clone 'https://sbdl.dev/sbdl-example.git'
cd sbdl-example
cmake -S . -B build
cmake --build build
```

After the build completes, there are a set of output files available in the 'build' directory (including: a generate document, model images, traceability matrices, OpenFMEA file . . . ). You can view the pre-compiled output here.

### Demonstration

Step-by-step demonstration of SBDL installation and example testing.

(interactive-demo-here)

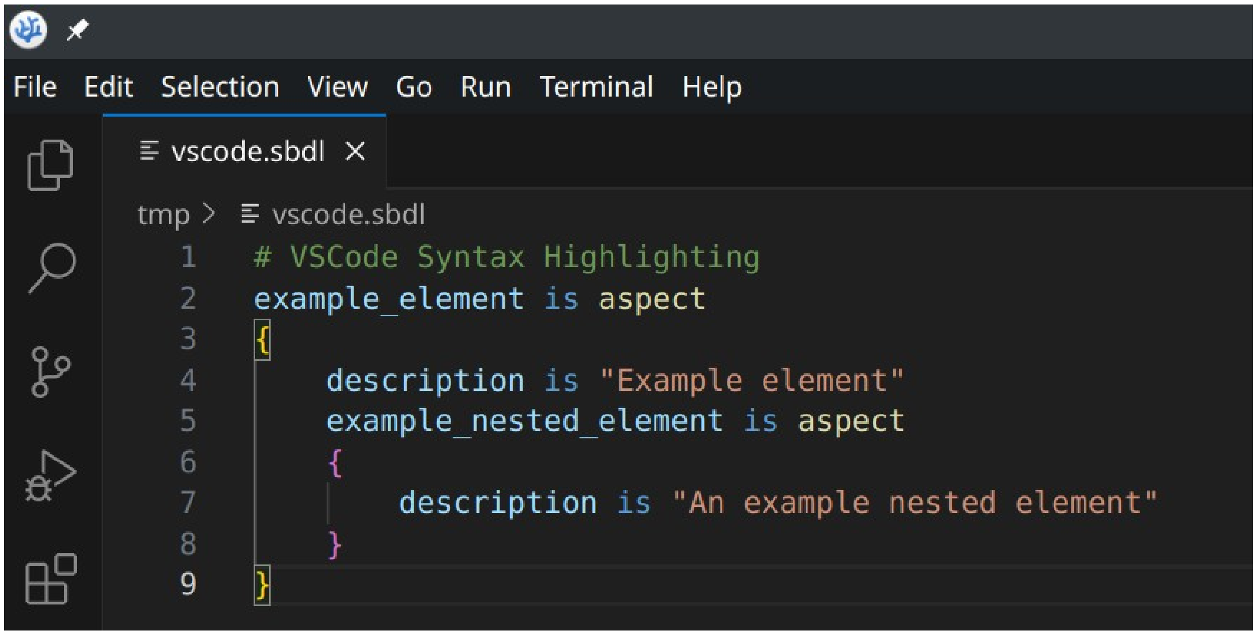
### Syntax Highlighting & More

A *Visual Studio Code* Extension is available that provides syntax highlighting and GUI integration of compiler functions.

Download and install the SBDL Visual Studio Code Extension.

After installation, syntax highlighting is available immediately.

Right clicking in the explorer panel exposes the available compiler shortcuts.



```
# VSCode Syntax Highlighting
example_element is aspect
{
    description is "Example element"
    example_nested_element is aspect
    {
        description is "An example nested element"
    }
}
```

Figure 3: VSCode Syntax Highlighting

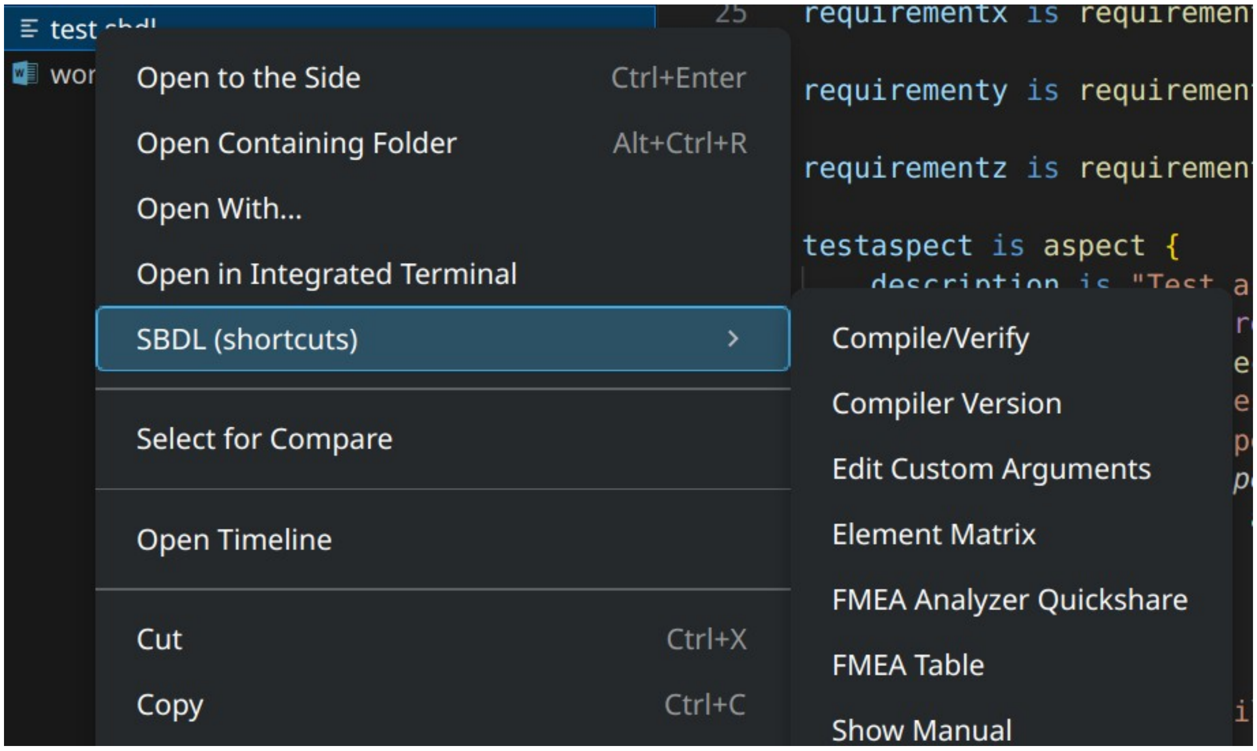
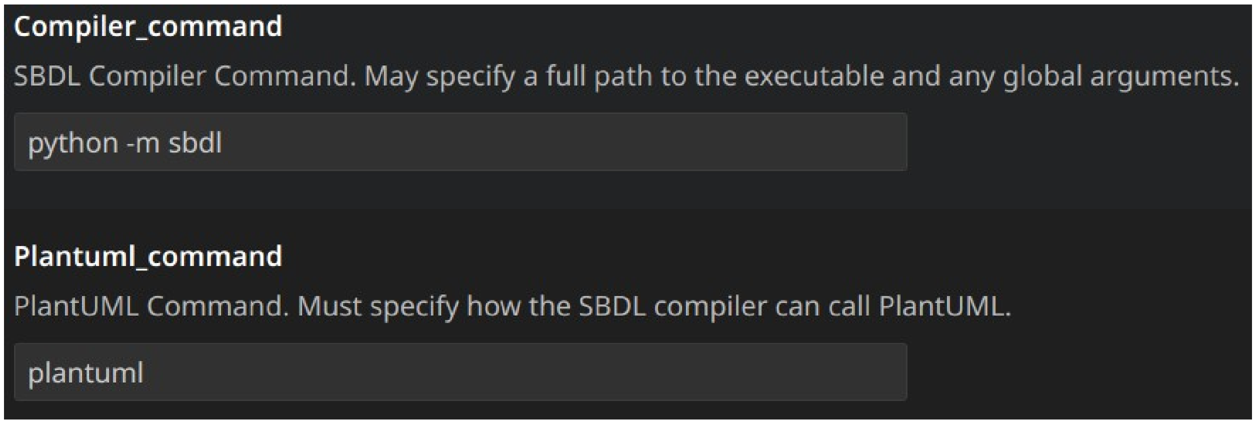


Figure 4: VSCode Explorer Context Menu

Depending on the host system, it may be necessary to customize the extension configuration. The configuration can be found in: File->Preferences->Settings->Extensions->SBDL



**Compiler_command**

SBDL Compiler Command. May specify a full path to the executable and any global arguments.

```
python -m sbdl
```

**Plantuml_command**

PlantUML Command. Must specify how the SBDL compiler can call PlantUML.

```
plantuml
```

Figure 5: VSCode Extension Configuration

#### Configuration options

*Compiler_command*: Command to invoke SBDL. This can be as simple as 'sbdl', 'python -m sbdl' or, if there are multiple python versions installed, 'py -XX.YY -m sbdl'.

*Plantuml_command*: Command to invoke PlantUML (external tool). This might be simply 'plantuml' or 'java -jar /path/to/jar'.

## SBDL Language

The SBDL language is terse, simple and easy for humans to read. It is intended to be written directly but is also amenable to automation as a target of other tools.

***Model Based*** SBDL, at its core, captures the relationships between different parts of a system's domain model. This includes static, dynamic and state based facets of a system, in addition to testing and failure mode and effect analyses.

***Declarative*** SBDL captures the structure and relationships of the elements of a system's behaviour but does not describe *how* they achieve target behaviour; it is a not a *programming* language.

***Immutable*** SBDL elements are defined once, at a single location, and may not be modified thereafter.

***Distributable*** SBDL definitions may reside in a dedicated SBDL source file but, importantly, they may also be distributed as annotations within other design materials; such materials include source-code, documents, requirement management systems . . .

***Traceable*** Every SBDL definition has traceable relations and is traceable to a specific location of definition.

#### Semantics

The SBDL metamodel represents a system as a set of *elements*. Elements have an identifier, *type*, *properties*, and may be *related* to one another. In this sense, the metamodel can be formally described as a coloured graph, or network.

**Elements, types, relations and properties** Each system element has a particular *type* (e.g. aspect, requirement, definition, interface, usecase . . . ) and represents some facet of the system's behaviour. Detailed information about types can be composed into a system can be found in the Metamodel Reference.

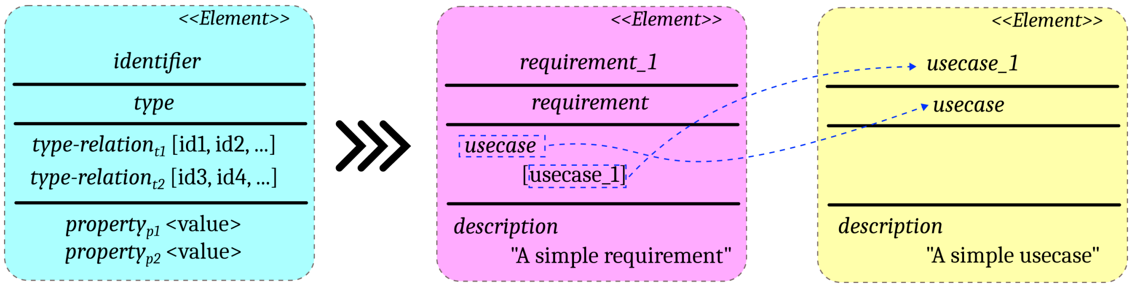


Figure 6: Simple representation of SBDL model elements and relations

Each element may be *related* to other elements of the system in a typed way (not all elements may be related to one another – only those relations which have a defined meaning).

Each element also has one or more *properties*, which define a characteristic of the given element. The most basic property, available on all elements, is the textual *description* property; but different elements have various properties which may be attached to them, depending on the element's type.

All elements defined in SBDL are also implicitly spatially sensitive: their locality of definition is an automatic property of the element, anchoring the element's definition to the material location and its position within it.

The composition of all elements together constitutes the overall system behaviour description.

SBDL can capture hierarchical structuring, and relate the properties of the system to that structure. This includes: static properties, dynamic behaviour, stateful behaviour, test definitions, and failure modes, along with the more typical requirements and usecases.

**Distributed Definition** SBDL elements can be defined together in one place, like a conventional language source file, but the intention (and power) of SBDL comes from the ability to embed individual SBDL element definitions close the materials to which they are most relevant. For example, requirements may be defined alongside where they are documented, while a software architecture decomposition may be defined (and distributed throughout) the source code. This distributed definition is the power behind the aforementioned implicit spatial locality of reference: where each element definition is coupled with the location of material to which it is most closely related.

## Syntax

### Core Language *(with examples)*

The syntax of SBDL is designed to be straight-forward. SBDL content consists of a series of statements; the most common of which is an element definition. An element definition specifies a named model element of a specific model type, along with its properties and relations.

The general form of an SBDL definition is as follows:

```
sbdl_id is sbdl_type {
   description is "something";
   some_other_property is a,b,c;
}
```

Where *sbdl_id* is the name of the model element, *sbdl_type* is the type of the model element, and the entries between the curly-braces define the properties and relations of that element (the semi-colon can be used as an *optional* separator). Statements are intended to be expressed on a single line but may be broken over several lines.

Consider the more concrete example below:

```
rocket_system   is aspect { description is "Rocket Launch System" }
rocket_booster  is aspect { description is "Booster Sub-System";  parent is rocket_system }
rocket_steering is aspect { description is "Steering Sub-System"; parent is rocket_system; related is r
```

The above definition set describes a structural decomposition using the ‘aspect’ type. In this case, a ‘Rocket Launch System’ system is described along with two sub-systems: ‘Booster’ and ‘Steering’. The sub-systems are related to the parent via the ‘parent’ relation; conversely, they could also have been related from the parent’s perspective, using the ‘child’ relation. In addition, the rocket steering sub-system is adjacently related to the rocket_booster system.

This simple model can be extended with some affiliated requirements:

```
system_requirement_1   is requirement { description is "The rocket shall launch...";  aspect is rocket_
booster_requirement_1  is requirement { description is "The rocket boster shall fire ...";  aspect is r
steering_requirement_1 is requirement { description is "The rocket shall be steerable ...";  aspect is 
```

Three named requirements are defined and associated with different decompositional aspects by the ‘aspect’ type relation.

The same structure can then be extended with, for example, a dynamic function:

```
launch_protocol is function { description is "Launch Sequence"; aspect is rocket_system; event is fire_
fire_booster    is event    { description is "Fire boosters"; aspect is rocket_booster }
correct_course  is event    { description is "Correct course trajectory"; aspect is rocket_steering }
```

The above defines a single function consisting of two events. Notably, the containing function definition exploits the ordered nature of property definition lists (in this case to order to events).

And, finally, the defitions can be augmented with state information:

```
using { aspect is rocket_system }
rocket_launch    is transition { description is "Rocket Launch" state is rocket_ready,rocket_in_motion 
rocket_ready     is state      { description is "Rocket ready for launch" }
rocket_in_motion is state      { description is "Rocket is in motion" }
```

Notice that the ‘rocket_launch’ transition uses the previously defined ‘fire_booster’ as the causal event of the transition. The use of the ‘using’ statement affiliates all subsequent statements with the rocket_system aspect.

**Strings** As shown in the examples, string properties are encapsulated within *“double-quotes”*.

*Raw strings*, which may contain unescaped double-quotes and preserve whitespace and new lines, can be expressed within triple-backets:

```
[[[this is
a
raw
    "string"
]]]
```

**Stereotyping** Elements and their relations may be extended with *stereotypes*. Stereotypes add additional typing constraints and information.

Stereotypes are expressed by extending identifiers using a caret symbol (‘^’).

The third line of the aspect definition example can be extended in this way:

```
rocket_steering is aspect { description is "Steering Sub-System"; parent is rocket_system; related is r
```

Above, the relation to the rocket_booster sub-system is stereotyped to show that the rocket_steering sub-system *controls* the rocket_booster (^controls). Useful stereotypes for relations between *aspects* include 'inherit' and 'compose', which will also be rendered canonically in a compiled aspect diagram.

An element definition itself can also be stereotyped. Consider the definition of a new aspect, particular to software:

```
steering_firmware^software is aspect { description is "Steering control software"; parent is rocket_ste
```

A new structural aspect is defined above ('steering_firmware') which is part of the rocket_steering subsystem and, importantly, is stereotyped as software ('^software').

**Model Output Generation** Using the SBDL compiler, the above SBDL statements can be aggregated from distributed sources and verified for correctness. Several different model output views are available; a selection of which are shown below.

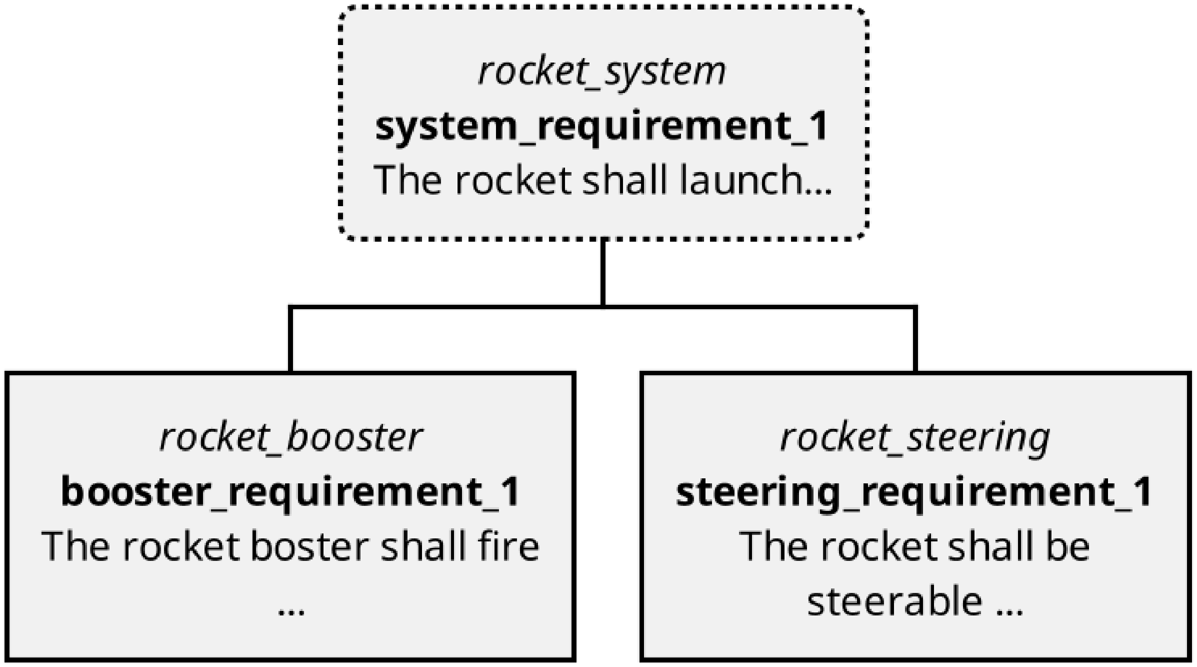


Figure 7: Requirements Model View

Many additional views are available, and the expressive power of the views show can also be extended with additional information. To see a more extensive treatment of the SBDL language, be sure to explore the fuller SBDL example project.

**Custom Types** It is possible to create specialised type variants of the SBDL base types (described by the metamodel). These specialised variants can be created using the 'customtype' keyword, *extending* the semantics of the base type. A newly defined custom type can then be used in place of a base type name with the usual syntax.

Custom types can (re)define:

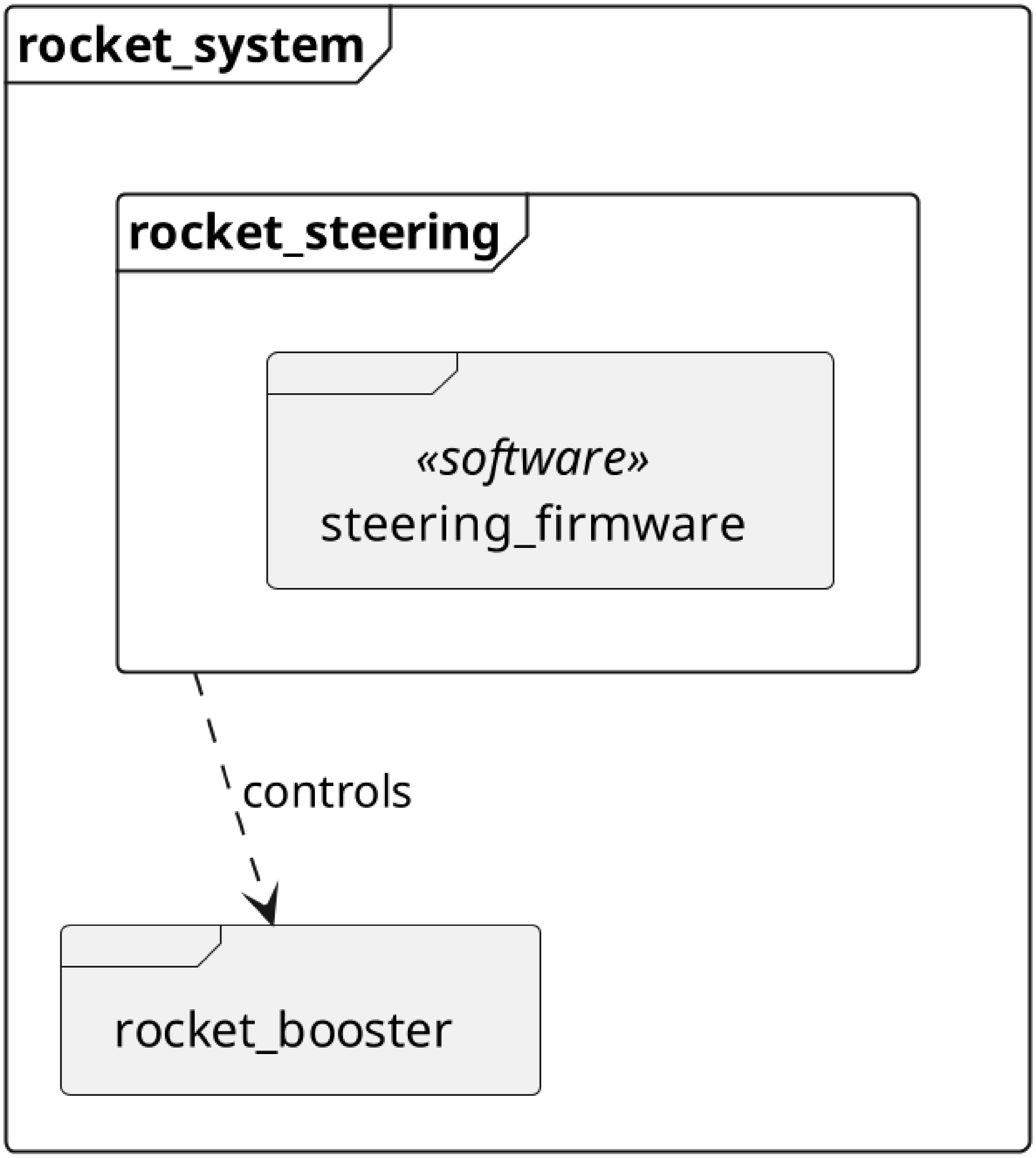


Figure 8: Aspect Model View

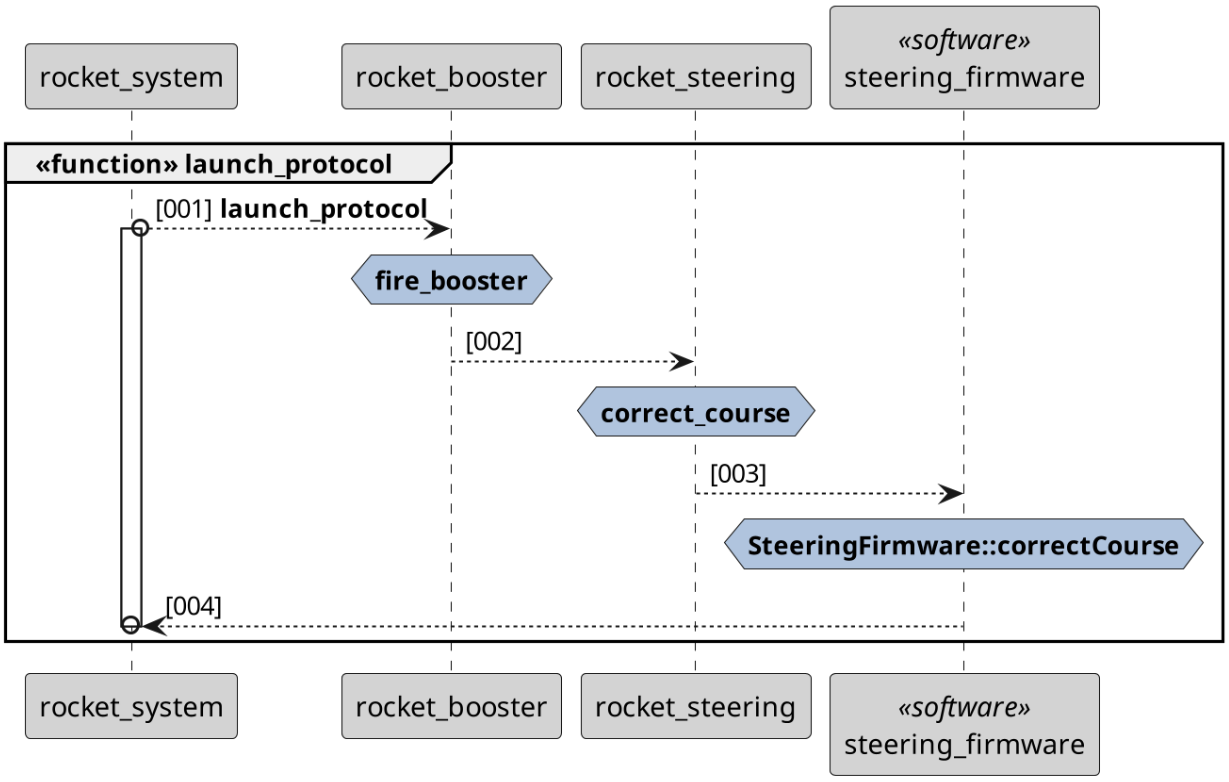


Figure 9: Function Model View

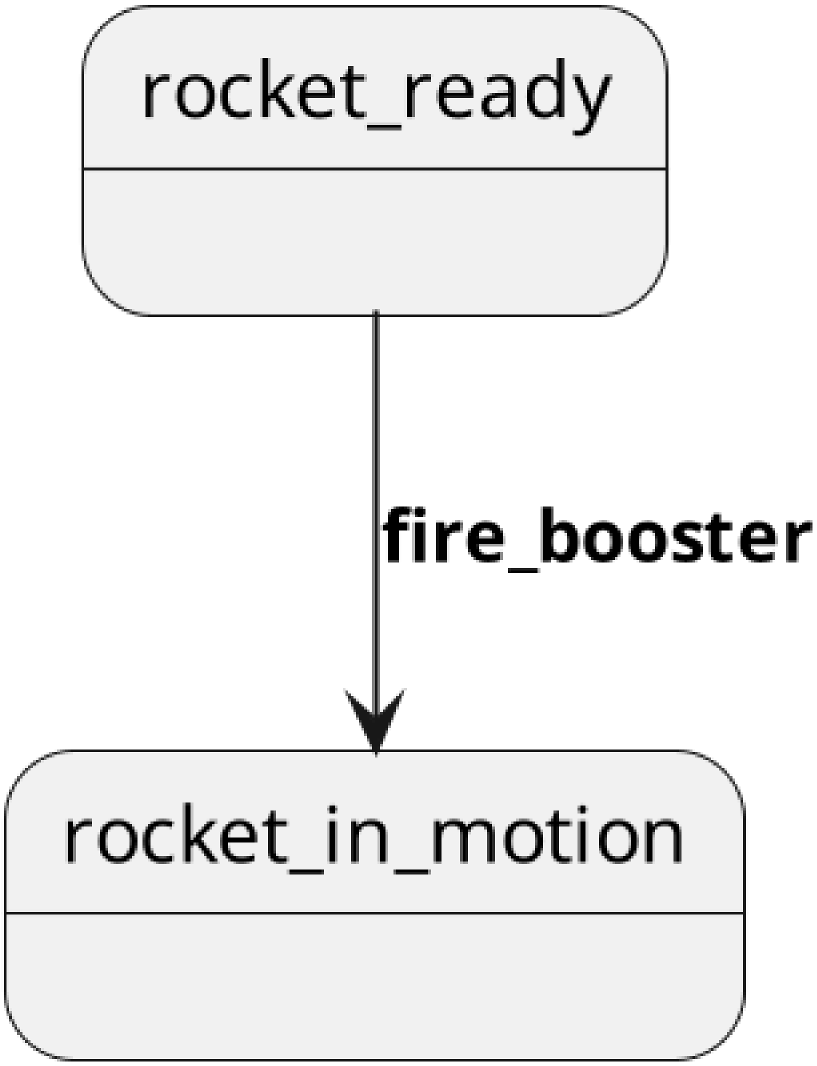


Figure 10: State Model View

- the name of the type
- permitted relation types
- mandatory properties
- optional properties
- default values properties

Custom types can therefore be used to realise an equivalent of Profiles in UML/SysML.

Consider the following example:

```
customtype FunctionalElement is aspect {
    relation_type         is requirement;
    required_property     is UID;
    optional_property     is Rationale;
    some_default_property is "Default text here";
}
new_functional_element is FunctionalElement { requirement is System_Requirement1; UID is "XXXX:YYYY" }
```

The statements above creates a new type called 'FunctionalElement' which is used to define an element 'new_functional_element' that:

- can be linked to an element of type 'requirement'
- must have a property 'UUID'
- may optionally have a property 'Rationale'
- by default has a property 'some_default_property' with a default string assigned (which may be overwritten)

**Nested Statements (Parentage Shorthand)** When elements have a hierarchical relationship (parent/child) they may be expressed by nesting their associated statements. The nesting of statements creates an implicit parenting relation from the nested element to its containing element.

The previous example snippet describing the aspect elements of the rocket system could be alternatively expressed as follows:

```
rocket_system is aspect {
    description     is "Rocket Launch System"
    rocket_booster  is aspect { description is "Booster Sub-System" }
    rocket_steering is aspect { description is "Steering Sub-System" related is rocket_booster }
}
```

Notice how the explicit referencing of parents is no longer required. Element nesting does not alter the scope of element identifiers.

**Relationship Operators (Relation Shorthand)** A common necessity is establishing a relationship between two adjacent input elements; for example:

```
element_a is aspect       { description is "Example element a" }
element_b is requirement  { description is "Example requirement 1"; aspect is element_a }
element_c is requirement  { description is "Example requirement 2"; aspect is element_a }
```

Above, the two requirements are explicitly related to element a.

The relation operators '||' and '~|' provide a 'syntactic sugar' for achieving this with greater brevity than explicitly specifying the named relations.

The '||' operating implies a relation between two adjacent elements in the input. In the case of a chain of such relations, the '~|' operating implies a relation between the current input element and source of the relation chain.

The example above could therefore be re-written as:

```
element_a is aspect       { description is "Example element a" }
||
element_b is requirement  { description is "Example requirement 1" }
~|
element_c is requirement  { description is "Example requirement 2" }
```

Notice that the aspect relations are no longer explicitly specified in the requirement elements because they are now implied by the subsequent relation operators.

**Unit Scope Management / Namespaces** SBDL facilitates scope and namespace management at the level of compilation units (typically files).

Two basic mechanisms allow for this: the 'using' and 'scope' keywords.

```
example_aspect is aspect { description is "Example aspect" }
using { aspect is example_aspect }
scope { identifier is PackageName }
example_requirement1 is requirement { description is "Example requirement 1" }
example_requirement2 is requirement { description is "Example requirement 2" }
```

... is compiled to:

```
example_aspect is aspect { description is "Example aspect"; reference is -:2; }
PackageName::example_requirement1 is requirement { aspect is example_aspect; description is "Example re
PackageName::example_requirement2 is requirement { aspect is example_aspect; description is "Example re
```

**using**: applies the contained relations (in the case of the above: a relation to the rocket_system aspect) to all subsequent element definitions.

**scope**: applies the namespace prefix (in the case of the above, element definition identifier) to all subsequent element definitions.

Both *using* and *scope* can be reset/cleared to modify behaviour for further elements defined in the compilation unit:

```
example_aspect is aspect { description is "Example aspect" }
using { aspect is example_aspect }
scope { identifier is PackageName }
example_requirement1 is requirement { description is "Example requirement 1" }
using { aspect is "" }
scope { identifier is "" }
example_requirement2 is requirement { description is "Example requirement 2" }
```

... is compiled to:

```
example_aspect is aspect { description is "Example aspect"; reference is -:2; }
PackageName::example_requirement1 is requirement { aspect is example_aspect; description is "Example re
example_requirement2 is requirement { description is "Example requirement 2"; }
```

**Relationship Hash Checks** Often, two distantly defined elements have a critical relation. For example:

**File A**

```
some_requirement is requirement { description is "Some critical definition" }
```

**File B**

```
some_aspect is aspect {
  description is "Highly dependent on the requirement"
  requirement is some_requirement
}
```

It is possible that the *content* of elements in 'File A' may change independently of those in 'File B', without any element identifiers being modified or removed. This would mean that, despite the content potentially changing radically, compilation and first-order relation verificatin would *not* fail.

If such a relation has a strong semantic binding to the content of a definition, and compilation failure upon a change would be desirable, a relation reference may be augmented with a hash value:

**File B**

```
some_aspect is aspect {
  description is "Highly dependent on the requirement"
  requirement is some_requirement~41575
}
```

Above, the 'some_requirement' relation has been extended with an SBDL element hash reference, using the '~' relation operator. When 'some_requirement' is modified without a coordinated change to its relation reference in 'some_aspect', compilation will fail.

Element hash values can be retrieved using the 'query' mode of the SBDL compiler or by simply entering an arbutrary hash value in the relation reference and extracting the correct hash value from the subsequent compilation error message.

**Embedding statements** SBDL definitions may be expressed together in an SBDL-native file or they may be embedded as annotations within another file. Such other files might include design descriptions, documents, source code etc.

Embedding SBDL in another file-type is as simple as prefixing the line with the SBDL directive indicator: "@sbdl".

For example, an additional event in the launch_protocol function could be defined in a (C++) source file as follows:

```
CourseStatus SteeringFirmware::correctCourse ( ... )
{
  // Correct the course based on the firmware algorithm output ...
  // @sbdl update_course is event { description is "Correct course"; aspect is steering_firmware }
  ...
}
```

The SBDL compiler will extract such SBDL statements and include them in the compilation context.

**Embedding Blocks (multiline)** If syntactically permitted by the containing language, it is possible to create SBDL 'blocks' – multiple lines of SBDL – embedded within the containing language, for example:

```
@sbdl-begin
  test_elem    is aspect      { description is "..." }
  another_elem is requirement { description is "..." }
@sbdl-end
```

This allows for the expression of multiple SBDL statements without the requirement that each line be prefixed with '@sbdl'.

The restriction is, however, that every line in the block must be valid SBDL. This means the application of embedded blocks is mostly useful for embedding within textual formats, such as Markdown and similar.

### Directives and Cross-Referencing

**Directives** In addition to the core syntax of SBDL exists the concept of directives. A directive specifies some additional specific behaviour (performed by the compiler), often returning content to the regular syntax.

Compiler directives are specified using square brackets and the at-symbol:

```
[@DIRECTIVE_NAME:argument1,argument2,argument...]
```

A list of available compiler directives can be found here.

**Cross-Referencing** Directives also allow for cross-referencing: referring to the properties of other named elements in SBDL. For example:

```
some_requirement is requirement { description is "Defines and controls [@some_other_elem_id:description]
```

The above example will include, in the description of 'some_requirement' the description of the other named element. This works for all properties of all elements.

**Coupling to embedded definitions** Directives permit the coupling of SBDL definitions to artifacts of the containing environment/language. In the case of the example definition embedded into a C++ file, the coupling might be as follows:

```
CourseStatus SteeringFirmware::correctCourse ( ... )
{
  // Correct the course based on the firmware algorithm output ...
  // @sbdl [@CFUNC] is event { description is "[@-LINE]" }
  ...
}
```

With these modifications, the embedded definition is refactored to the use the containing function name as the SBDL identifier ([@CFUNC]) and will take the previous line as the description ([@-LINE]).

After defining this new event, it should be added to the appropriate position in the launch_protocol function. This can either be done explicitly in the function itself or by using event-tree, where an event also entails its child events:

```
correct_course  is event    { description is "Correct course trajectory"; aspect is rocket_steering; chi
```

### Custom Rule Definition

> **Work In Progress** – custom rules and the Prolog extension are experimental and under active development. The final syntax for *accessing* the Prolog subsystem may be slightly different in later releases, but the rule syntax itself will not change.

It is useful to be able to specify custom rules in addition to those provided by the base metamodel of the language. Such rules might include naming conventions, relation permissions (potentially over indirect relations), additional type property constraints, and so on.

Such additional rules can be specified through the Prolog extension of SBDL.

During compilation of a set of SBDL files, the resulting elements are exposed to a Prolog environment as a set of facts. Assertions can then be made, using the SBDL Prolog directives, to query these facts.

Combining these custom rules with custom types provides a powerful and adaptable type specification system.

**Fact Format** Facts about elements are automatically exposed to Prolog with a uniform argument cardinality. For reference, these take the Prolog form:

```
type_name(identifier('identifier'), stereotype('stereotype'), properties([description('description'), r
```

Consider the following SBDL statement:

```
base_element is aspect {
  description is "base aspect"
  child_element is aspect { description is "child aspect" }
}
test_requirement is requirement { aspect is child_element; description is "Test Requirement" }
```

In the Prolog environment, this is exposed as the following facts:

```
aspect(identifier('base_element'), stereotype('None'), properties([description('base aspect'), referenc
aspect(identifier('child_element'), stereotype('None'), properties([description('child aspect'), refere
requirement(identifier('test_requirement'), stereotype('None'), properties([description('Test Requireme
```

All parsed model elements will be automatically available in the Prolog environment as facts of this form and may subsequently be queried. It is not necessary to write the Prolog facts oneself.

**Directives** There are several compiler directives provided for interaction with Prolog, and the exposed element facts, primarily: 'PL_START','PL_ASSERT','PL_ASSERT!', and 'PL_COMMAND'. These directives are detailed in the compiler directives section.

**Rule Assertion Example** In the example SBDL snippet, it could, for example be asserted that an element with a particular identifier must exist.

Asserting such a query in Prolog, using the fact structure described previously, would look like:

```
aspect(identifier(child_element),_,_,_,_,_,_).
```

This can then be expressed in SBDL (by extending the SBDL snippet) as follows:

```
[@PL_START] # make the Prolog environment available (needed once anywhere in the input)
[@PL_ASSERT:TestAspectExists, aspect( identifier(child_element), _, _, _, _, _, _)] # assert the truth
```

The 'PL_ASSERT' directive is called with two arguments: the descriptive name of the assertion and the Prolog query itself.

When an element with an identifier 'child_element' does not exist, the SBDL will fail to compile, citing the name of the failing assertion.

Of course, much more complex rules can be expressed in Prolog, but this is beyond the scope of this documentation. Have a look at this online introduction to Prolog as a starting point.

### Additional Syntax Notes

**Bidirectional Relation Definition** Relations between elements can, in most cases, be defined in either direction (or both!) and have the same effect:

```
some_aspect      is aspect      { ... }
some_requirement is requirement { ... aspect is some_aspect }
```

Is equivalent to:

```
some_aspect      is aspect      { ... requirement is some_requirement }
some_requirement is requirement { ... }
```

Is also equivalent to:

```
some_aspect      is aspect      { ... requirement is some_requirement }
some_requirement is requirement { ... aspect is some_aspect }
```

**Implicit Reference Property** All statement defined in SBDL automatically acquire the 'reference' property when compiled. The reference property indicates the location of definition in terms of file-path and line number. The reference property is useful in understanding where

**SBDL-Native Files vs SBDL Embedded Syntax** SBDL statements embedded within a parent file of a different format/language must always be prefixed with the SBDL statement indicator:

```
@sbdl some_id is some_type { ... }
```

SBDL statements contained within an SBDL-native file (a file containing *only* SBDL statements) do not need to be prefixed with the SBDL statement indicator. Instead, the file itself must begin with the SBDL *file* indicator:

```
#!sbdl
some_id is some_type { ... }
diff_id is some_type { ... }
```

**Multi-line Statements** SBDL statements can be broken across multiple lines in SBDL-native files. SBDL-nativefiles are essentially free-form with regards to whitespace and line breaks.

When SBDL is embedded inside another language file or format (embedded syntax), statements must be explicitly broken across multiple lines, using the line continuation symbol: backslash (\).

```
@sbdl example_definition is requirement { \
  description is "On a new line" \
}
```

**Custom Properties** Unguarded property names are only permitted when they are valid for the element type being defined. Custom properties may, however, be specified ac hoc by using the custom-property-prefix ('custom:') on a per element basis.

```
example_definition is requirement { description is "Something"; custom:my_property is "Some content" }
```

**Formal Syntax** A formal version of the SBDL syntax is captured as an EBNF, or 'Railroad', diagram.

## Metamodel Reference

### Element Types Overview

The type graph below illustrates the available definition types, their properties, and, by an edge between two types, the valid relations which may be specified between them. Each type (its relations and properties) is also elaborated upon in its own sub-section.

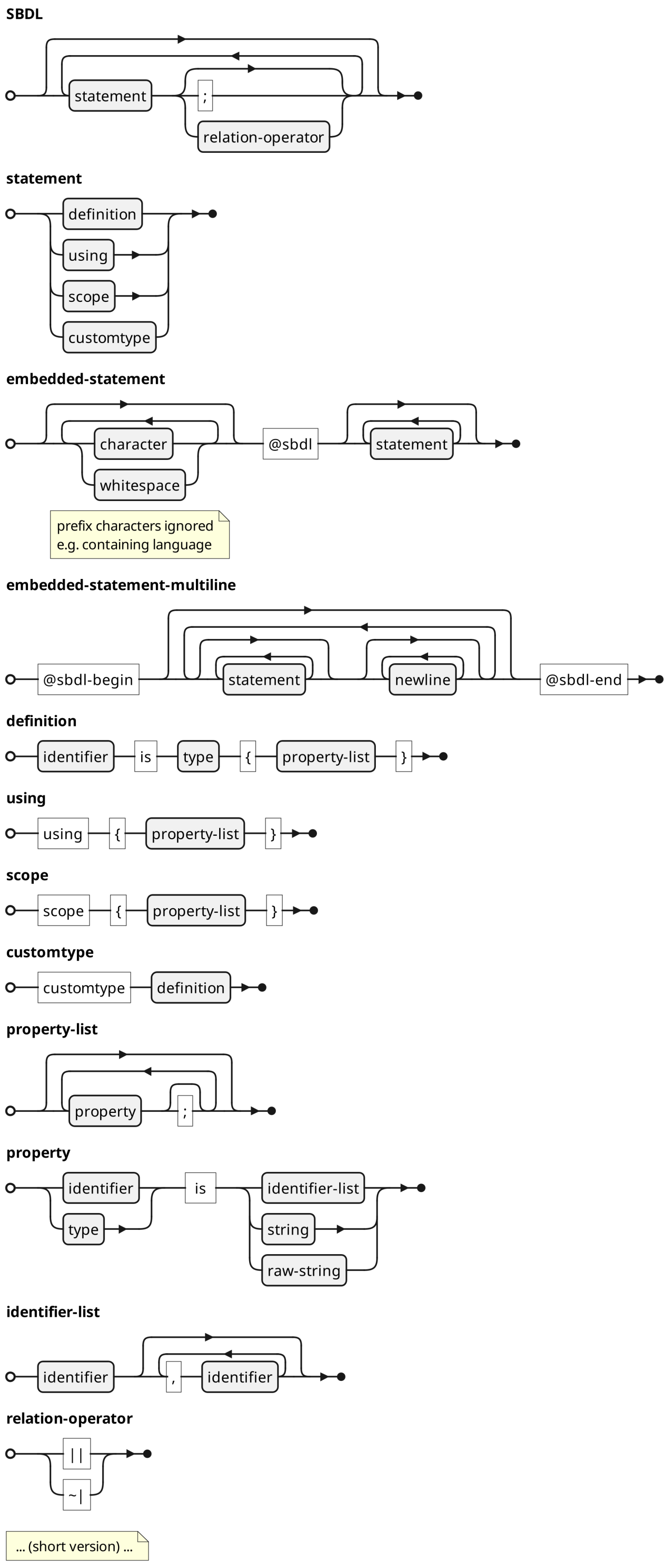
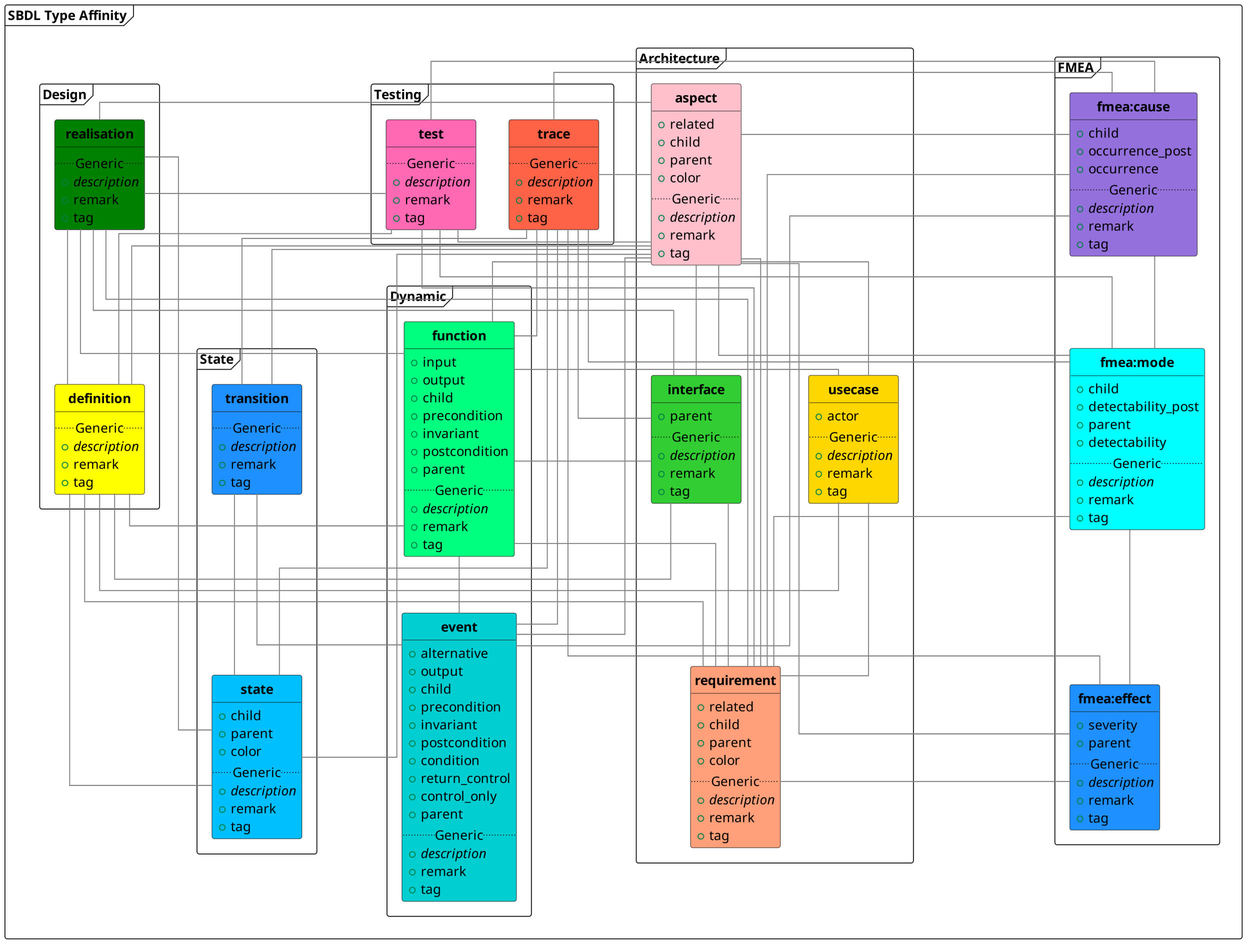


Figure 11: SBDL Syntax Diagram (short version)



Simplified Metamodel View (click for larger size)

### Element Types & Semantics

The follow subsections describe the available definition types, their meaning, and their properties.

Each element type is intended for relations with entities of affiliated types: for example, all elements may relate to an aspect, and failure modes may relate to a requirement, and so on. Together, these relations constitute a typed graph which describes the system structure and behaviour. This typed graph of definitions and relations can be used to generate many of output formats provided by the SBDL compiler.

**Architecture Elements** Architectural elements are used for specifying system decomposition. Such decompositions may be according to different views/schemes. for example: logical/structural, functional ...

aspect Element

Description:

*Unit of system decomposition. May associate with a variety of perspectives: logical, functional ...*

Relations:

aspect • requirement • fmea:mode • fmea:effect • fmea:cause • fmea:control • fmea:detection • fmea:action-control • fmea:action-detection • test • definition • realisation • function • event • state • transition • usecase • interface • trace • grou

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

color: [string] Color associated with element.

child: [identifier] Hierarchical child of the given element.

related: [identifier] Indicates a related element.

parent: [identifier] Hierarchical parent of the given element.

requirement Element

Description:

*Requirement (or acceptance criteria) definition. Describes an expected behaviour at the associated level of abstraction.*

Relations:

fmea:control • fmea:mode • fmea:cause • fmea:effect • aspect • test • definition • realisation • function • usecase • interface •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

color: [string] Color associated with element.

child: [identifier] Hierarchical child of the given element.

related: [identifier] Indicates a related element.

parent: [identifier] Hierarchical parent of the given element.

usecase Element

Description:

*Definition of a usecase within a particular abstraction of a the system.*

Relations:

requirement • aspect • definition • function •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

actor: [string] Indicates the name of an actor affiliated with the given element.

interface Element

Description:

*Definition of an interface exposing behaviour externally from the given abstraction.*

Relations:

requirement • interface • aspect • trace • definition • realisation • function • interface •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

parent: [identifier] Hierarchical parent of the given element.

group Element

Description:

*Syntactic group of model elements. Used only to structure model representation and facilitate filtering; has no semantic implications for the model itself. May parent (contain) any other element type.*

Relations:

aspect •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

**FMEA Elements** Failure Mode and Effect Analysis (FMEA) elements are used to describe risks present in a system, and the controls and actions mitigating them.

fmea:mode Element

Description:

*Failure Mode. Describes the way in which a failure may occur, from the perspective of a requirement.*

Relations:

requirement • fmea:effect • fmea:control • fmea:detection • test • fmea:action-control • fmea:action-detection • fmea:cause • aspect •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

detectability_post: [number] Numerical rating on the interval [1(best)..10(worst)] indicating quality of a failure detection AFTER realisation of an improvement action.

child: [identifier] Hierarchical child of the given element.

detectability: [number] Numerical rating on the interval [1(best)..10(worst)] indicating quality of a failure detection.

parent: [identifier] Hierarchical parent of the given element.

fmea:effect Element

Description:

*Failure Effect defintion. Describes the consequence of a failure mode.*

Relations:

requirement • fmea:mode • aspect • trace •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

parent: [identifier] Hierarchical parent of the given element.

severity: [number] Numerical rating on the interval [1(least)..10(most)] indicating the severity of a failure effect.

fmea:cause Element

Description:

*Failure Cause. An underlying technical mechanism, scenario or sequence of events that may result in a failure mode.*

Relations:

requirement • fmea:mode • fmea:control • fmea:detection • test • fmea:action-control • fmea:action-detection • event • aspect • trace

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

child: [identifier] Hierarchical child of the given element.

occurrence: [number] Numerical rating on the interval [1(infrequent)..10(always)] indicating the probability of a failure cause occurring.

occurrence_post: [number] Numerical rating on the interval [1(infrequent)..10(always)] indicating the probability of a failure cause occurring AFTER realisation of an improvement action.

fmea:control Element

Description:

*Existing controls which are present to prevent a failure cause either from occurring or leading to its associated failure mode.*

Relations:

fmea:mode • fmea:cause • aspect • trace • requirement •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

fmea:detection Element

Description:

*Existing detections (tests) which are present to measure (before release) the occurence of a failure mode.*

Relations:

fmea:mode • fmea:cause • aspect • test •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

fmea:action-control Element

Description:

*Necessary steps that remain to be taken in order to prevent the occurance of a failure cause or mode.*

Relations:

fmea:mode • fmea:cause • aspect •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

fmea:action-detection Element

Description:

*Necessary steps that remain to be taken in order to increase the detectability of a failure mode.*

Relations:

fmea:mode • fmea:cause • aspect •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

**Testing Elements** Testing elements describe the implementation and coverage of tests (including dynamic traces).

test Element

Description:

*Instance of a test for a particular part of a design.*

Relations:

requirement • definition • realisation • fmea:detection • fmea:mode • fmea:cause • aspect •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

trace Element

Description:

*A dyanmic occurance of element instance (for example, an event or failure cause). Intended to be embedded within log files. Can be used to build and validate dynamic behaviour against the statically defined behaviour model.*

Relations:

fmea:cause • fmea:mode • fmea:effect • fmea:control • function • transition • event • state • interface • aspect •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

**Design Elements** Design elements elaborate on the prescription and description of design decisions and their technical concepts.

definition Element

Description:

*Prescriptive definition of a particular aspect of design.*

Relations:

requirement • function • state • usecase • interface • aspect • test • realisation •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

realisation Element

Description:

*Final realisation of a particular aspect of prescriptive design.*

Relations:

requirement • definition • function • state • interface • aspect • test •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

**Dynamic Elements** Dynamic elements describe and structure dynamic system behaviours, such as functions and events.

function Element

Description:

*Definition of a function.*

Relations:

event • function • usecase • interface • requirement • aspect • trace • definition • realisation • function •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

invariant: [string] Conditions unaffected by the behaviour of the given element.

child: [identifier] Hierarchical child of the given element.

input: [string] Input consumed by the given element.

postcondition: [string] Condition resulting from the given element's behaviour.

precondition: [string] Precondition for correct behaviour of the given element.

output: [string] Output produced by the given element.

parent: [identifier] Hierarchical parent of the given element.

event Element

Description:

*Definition of a dynamic event. May be a step within a broader function or cause a transition between states. Events may be composed as trees, with an event also entailing all of its children. Decisions can be expressed with the condition property; unmet conditions entail the alternative (property) children instead of default children.*

Relations:

fmea:cause • aspect • trace • function • transition •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

return_control: [*] Presence indicates immediate return of control flow

invariant: [string] Conditions unaffected by the behaviour of the given element.

child: [identifier] Hierarchical child of the given element.

postcondition: [string] Condition resulting from the given element's behaviour.

precondition: [string] Precondition for correct behaviour of the given element.

output: [string] Output produced by the given element.

control_only: [*] Presence indicates control flow only

condition: [string] Element is conditional on the specified binary decision.

parent: [identifier] Hierarchical parent of the given element.

alternative: [identifier] Specifies an alternative for an unmet condition.

**State Elements** State elements capture the stateful behaviour of the system by offering a stateful view of dynamic elements.

state Element

Description:

*Definition of a state.*

Relations:

aspect • trace • definition • realisation • transition •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

child: [identifier] Hierarchical child of the given element.

color: [string] Color associated with element.

parent: [identifier] Hierarchical parent of the given element.

transition Element

Description:

*Definition of a transition between states. Takes an ordered pair of states (from,to).*

Relations:

state • event • aspect • trace •

Properties:

description: [string] Descriptive body of text for the given element.

remark: [string] General remark or more extensive information.

tag: [string] Comma separated list of tags which indicate some property of the given element.

## SBDL Compiler

### Command-Line Interface

```
usage: sbdl [-h] [-m operating_mode] [-o output_file] [--version] [-W {normal,all}] [--hidden] [-i] [-s
            [--manual] [--dump-config config_file] [--load-config config_file] [--list-config] [--set-c
            [--trace [trace_files ...]] [--template template_file] [-fc element_identifier] [-fl elemen
            [-fpa element_identifier] [-fd filter_depth] [-ft element_type] [-fi element_identifier] [-
            [-fg group_identifier] [--custom-directive [compiler_definitions ...]] [--custom-mode [mode
            [source_files ...]
SBDL Version 1.13.1 (DSL Version 25.6). System Behaviour Description Language (SBDL) compiler.
WWW: https://sbdl.dev. Author: contact@sbdl.dev.

Base Arguments:
  source_files          List of files to compile ["-" implies stdin]

Optional Arguments:
  -h, --help            show this help message and exit
  -m operating_mode, --mode operating_mode
                        Specify the mode of operation
  -o output_file, --output output_file
                        Specify the name of the output file
  --version             Print the current version
  -W {normal,all}, -w {normal,all}, --warning {normal,all}
                        Set warning level
  --hidden              Include hidden files when recursing
  -i, --id              Include element identifiers in applicable output formats
  -s, --source          Include source reference in applicable output formats
  -r, --recurse         Recurse on directories specified in the input list
  --skip-errors         Do not stop for errors (emit warning instead)
  --title TITLE         Provide a default title for certain output formats
  -v, --verbose         Enable verbose output during execution
  --manual              Show extensive SBDL manual page
  --dump-config config_file, --dumpconfig config_file
                        Dump the internal configuration to a named JSON file
  --load-config config_file, --loadconfig config_file
                        Load the internal configuration from a named JSON file
  --list-config, --listconfig
                        List internal configuration options
  --set-config config_option config_value, --setconfig config_option config_value
                        Set a named configuration option
  -D name value, --define name value
                        Specify a named global definition
  --trace [trace_files ...]
                        Provide a trace file to be processed
  --template template_file
                        Specify a template file for the 'template-fill' mode
  -fc element_identifier, --filter-connected element_identifier
                        Filter everything but those elements with a direct or indirect connection to the
                        parents/children]
  -fl element_identifier, --filter-linked element_identifier
                        Filter everything but those elements with a direct or indirect connection to the
                        parents/children]
  -fch element_identifier, --filter-children element_identifier
                        Filter everything but those elements which are children of the specified element
  -fpa element_identifier, --filter-parents element_identifier
                        Filter everything but those elements which are parental ancestors of the specifi
  -fd filter_depth, --filter-depth filter_depth
                        Maximum depth for filters which pursue links (natural number)
  -ft element_type, --filter-type element_type
                        Filter everything but those elements which are of the specified element type (re
  -fi element_identifier, --filter-identifier element_identifier
                        Filter everything but those elements whose identifiers match the specified stri
  -fpr property_name property_value, --filter-property property_name property_value
                        Filter everything but those elements possessing a named property matching the s
  -fg group_identifier, --filter-group group_identifier
                        Shortcut filter for everything but those elements which are children of the spe
                        group element itself
  --custom-directive [compiler_definitions ...]
                        Specify a file path defining custom compiler directives
  --custom-mode [mode_definitions ...], --custom_mode [mode_definitions ...]
                        Specify a file path containing custom compiler modes
  --rpc RPC             Remote Procedure Call to be used by RPC-based modes

e.g. "sbdl <file 1> <file 2> <file n>"

---------------
Operating Modes
---------------
             compile: Parse all specified input files, gather SBDL elements, perform semantic checks, a
               query: Compile inputs, then pretty print the results (after filtering)
          csv-matrix: Compile inputs, then write a CSV-formatted representation of the SBDL elements to
           json-tree: Compile inputs, then write a JSON-formatted representation of the SBDL elements t
           yaml-tree: Compile inputs, then write a YAML-formatted representation of the SBDL elements t
     from:csv-matrix: Read SBDL-schema CSV-matrix inputs and write SBDL-formatted output
      from:json-tree: Read SBDL-schema JSON-tree inputs and write SBDL-formatted output
      from:yaml-tree: Read SBDL-schema YAML-tree inputs and write SBDL-formatted output
            openfmea: Compile inputs, then write the FMEA-related content to an OpenFMEA-formatted oupu
  openfmea-portfolio: Compile inputs, then write the FMEA-related content to an OpenFMEA Portfolio-form
       from:openfmea: Read OpenFMEA-formatted input and write SBDL-formatted output
        openfmea-csv: Compile inputs, then write the FMEA-related content to a CSV-formatted ouput
     network-diagram: Compile inputs, then write a PNG-formatted output, visually representing the netw
 requirement-diagram: Compile inputs, then write a SysML-style requirements diagram to rendering-backen
      aspect-diagram: Compile inputs, then write a SysML-style block diagram to rendering-backend-forma
     element-diagram: Compile inputs, then write a SysML-style block diagram to rendering-backend-forma
    function-diagram: Compile inputs, then write a SysML-style sequence diagram to rendering-backend-fo
       state-diagram: Compile inputs, then write a SysML-style state diagram to rendering-backend-forma
     usecase-diagram: Compile inputs, then write a SysML-style use-case diagram to rendering-backend-fo
       template-fill: Compile inputs, then provide an object, 'sbdl', in a Jinja parsing environment an
                 rpc: Compile inputs, then transmit to the RPC server for processing by the specified R
```

### Compilation CLI Examples

The full list of the SBDL compiler's modes of operation can be found in the Command-Line Interface section; this section will demonstrate some examples of how to use these modes.

**Compile Inputs** The core mode of the compiler is to *compile* its input arguments. This consists of the following steps:

1. Identify all input files (potentially recursively)
2. For each input file: extract its SBDL statements
3. Parse all extracted SBDL statements into a model representation
4. Verify the correctness of the model (including relations)

5. Write a single SBDL-native output file containing the *aggregation* of all inputs

This can be achieved with the following command-line:

```
sbdl -m compile -r input-file input-dir -o output.sbdl
```

The above command is useful as a basis for general executions of the SBDL compiler. In the example command there are the following arguments:

**sbdl**: Command to invoke the compiler.

**-m**: Switch to specify a mode.

**compile**: Name of the desired mode.

**-R**: Recurse on directories when identifying input files.

**input-(file|directory)**: Name one or more input files or directories.

**-o**: Switch to specify and output file.

**output.sbdl**: Name of a output file.

Because compile is the default mode of the SBDL compiler, and STDOUT is the default output, the command above can be simplified to the following:

```
sbdl -r input-file input-dir
```

(compiled SBDL will then be written to STDOUT)

Importantly, the compiled output can then be used as an input for further SBDL compiler modes, to simplify further invocations and also avoid repeating the aggregation process.

**Generate Model Diagram** Most invocations of the SBDL compiler follow the same form. The compile inputs example demonstrated this form; this section shows a variation of that form for generating an aspect diagram.

```
sbdl -m aspect-diagram output.sbdl -o aspect-diagram-output.png
```

The mode switch has been changed to specify 'aspect-diagram' and the output file has been adjusted to the desired PNG target. 'output.sbdl' is used (from the previous compile command) as input.

**Templating Output** Some SBDL compiler invocations require additional parameters, for example the templated output:

```
sbdl -m template-fill output.sbdl -o output.html --template template.html
```

In this case, the '–template' switch is used to specify to which template the parsed SBDL should be applied.

For a concrete exploration of using SBDL models in document templates, see the worked example.

### Compiler Directives

Available compiler directives:

(Format: Name: [arg1,arg2,… -> return] Description)

- *SELF*: [-> SBDL_Element] Element object for the current element (for internal cross-referencing)
- *SELF_ID*: [-> string] Identifier of the current element (when embedded within an element property)
- *SELF_PROP*: [-> string] Identifier of the current element's property (when embedded within an element property)
- *ABORT*: [string ->] Abort compilation with error

- *MESSAGE*: [string ->] Show a message (on stdout) during compilation
- *MSG*: [string -> string] Show a message (on stdout) during compilation and replace occurence with the message inline
- *DATE*: [-> string] Today's date
- *USER*: No Description
- *ADD*: [int, … -> int] Sum of arguments
- *SUB*: [int, … -> int] Subtraction of subsequent arguments from the first argument
- *EQUAL*: [val, val ->] Raise a compiler error if two value arguments are not equal
- *CONCAT*: [string, … -> string] Concatentate string arguments
- *INSTLI*: [string, … -> string] Index a list of string terms separated by whitespaces
- *SHOW_ALL*: [->] Show a message (on stdout) displaying all defined macros
- *RMCOM*: [string -> string] Remove comments from a string
- *MKID*: [string -> string] Make a given string a valid SBDL identifier
- *DFP*: [string -> string] Return a string defining a description as the previous line
- *REQUIRE_DSL_VERSION*: [string ->] Raise a compiler error if the DSL version is not at least equal to the argument
- *REQUIRE_COMPILER_VERSION*: [string ->] Raise a compiler error if the current compiler version is not at least equal to the argument
- *REQUIRE_DSL_VERSION_EXACT*: [string ->] Raise a compiler error if the DSL version is not exactly equal to the argument
- *PATH*: [-> string] Path of the current file
- *FILE*: [-> string] Name of the current file
- *DIR*: [-> string] Name of the current directory
- *CONTEXT*: [-> string] Context string (embedded statement)
- *=LINE*: [-> string] Current line (embedded statement)
- *-LINE*: [-> string] Previous line (embedded statement)
- *+LINE*: [-> string] Next line (embedded statement)
- *LINE*: [-> string] Current line number
- *IMPORT*: [string ->] Import contents of another SBDL file
- *DEFINE*: [string, string->] Create a key,value pair compiler definition
- *DEFINE_APPEND*: [string, string ->] Append to a named definition
- *DEFUNC*: [string, string, string ->] Create a definition from the result of a named compiler function application
- *DEFIND*: [string, string, string ->] Create a definition from the result of indexing another name definition
- *DEFINEF*: [string, string ->] Create a definition from the contents of a named file
- *DEFINEFH*: [string, string ->] Create a definition from the hash of the contents of a named file
- *EXPAND*: [string -> string] 'Expand' and sanitize a named definition
- *PARSED_ELEMENTS*: [->] Set of parsed elements (useable only programatically)
- *GENERATED_ELEMENTS*: [->] Set of output elements generated by other directives (useable only programmatically)
- *SYNTHETIC_ELEMENT*: [->] Synthetic object class reference, used to generate elements dynamically (usable only programmatically)
- *CROSS_REFS_AVAILABLE*: [-> bool] Indicates whether cross references are available
- *EXTEND_CAUSE*: [string, int -> string] In a failure cause to indicate it extends a name higher-level failure cause, args: [higher-cause, detectability] return [failure-mode-identifier]
- *CPPCLASS*: [-> string] Most recently defined C++ Class (embedded statement)
- *PYCLASS*: [-> string] Most recently defined Python Class (embedded statement)
- *CFUNC*: [-> string] Most recently defined C/C++ function (embedded statement)
- *PYFUNC*: [-> string] Most recently defined Python function (embedded statement)
- *PL_START*: [->] Startup the Prolog environment (required at least once to use Prolog directives;

indempotent)
- *PL_ASSERT*: [string ->] Assert the truth of a Prolog query (throw a compiler error if not)
- *PL_ASSERT!*: [string ->] Assert a Prolog query as false (throw a compiler error if not)
- *PL_COMMAND*: [string ->] Issue a Prolog statement
- *PL_TRACE_FILE*: [string->] Write all generated Prolog facts, assertions and commands to a named file (for external use in Prolog)
- *PL_RESULT_OUTPUT*: [->] Output the result of Prolog assertions to stdout

## License

The SBDL compiler and tools are provided under a modified BSD-3-Clause-based license.

## FAQ and Issue Reporting

### Frequently Asked Questions

**Q:** *What is that bug-looking mascot-thing?*

That's a generative-AI interpretation of a 'speedy-beetle', or 'speedle' (SBDL).

**Q:** *Who is the creator/maintainer of SBDL?*

SBDL is created and maintained by Michael Hicks.

**Q:** *How can I use the SBDL compiler as a Python library to interact with the SBDL language directly?*

A fuller SBDL Compiler API will follow the first stable release.

For now, have a look at this code example of using the SBDL compiler API to read a set of SBDL files and interact with the parsed and verified elements.

(HINT: make use of Python's introspection facilities to see what can be done with element objects)

**Q:** *I'm having problems with certain diagrams rendering improperly with the PlantUML backend. How can I fix them?*

First try installing a tested version of PlantUML.

If the problems persist, report an issue (below).

### Issue Reporting

Issues can be raised by sending an email to issues@sbdl.dev. Upon receipt, a trackable issue is automatically created, and a confirmation reference is returned to the sender.