System Behaviour Description Language

SBDL 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.



contact@sbdl.dev

2.1.8 (SBDL Compiler), 26.6 (SBDL Language) [#b57c0ad]

Contents

Overview . . . 2
The language… . . . 2
The compiler… . . . 4
Quick Start . . . 4
Get the compiler . . . 4
VSCode Integration . . . 5
Build the example . . . 8
Demonstration . . . 8
Language . . . 8
Semantics . . . 8
Syntax . . . 9
Model Output Generation . . . 20
SBDL and SysML v2 . . . 26
Metamodel . . . 26
Element Types Overview . . . 26
Element Types & Semantics . . . 28
Modelling Behaviour with Events . . . 37
Compiler . . . 38
Compiler Design . . . 38
Application Programming Interface (API) . . . 38
Command-Line Interface . . . 38
Compilation CLI Examples . . . 40
Compiler Directives . . . 41
License . . . 44
AI & LLMs . . . 44
MCP Server . . . 44
FAQ . . . 45
Frequently Asked Questions . . . 45
Issue Reporting . . . 45

Website · Install · VS Code extension · Example · Compiler design · PDF · DOI

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

New in 2.0

- **Native Renderer** No PlantUML/Java/Graphviz. The compiler now draws every diagram itself, as SVG, in six style packs, and adds two views: FMEA failure chains and the type-affinity matrix (more).
- **VS Code:** the Live Diagram is clickable (an element opens its definition), follows the editor's theme, and has an Options popover for diagram settings; trace verification opens as a table.
- **Faster on large models:** up to tens of times faster on models of thousands of elements.
- **An MCP server** gives AI assistants read-only access to a compiled model: search, descriptions, diagrams and exports (more).
- **The compiler documents itself:** its design is an SBDL model embedded in its own source, published as a generated page.

Overview

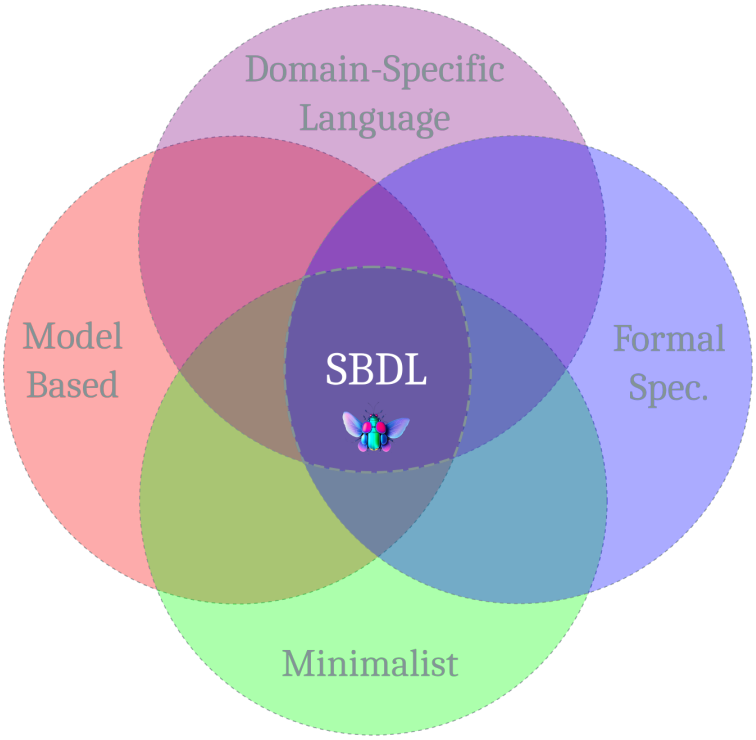


Figure 1: SBDL Venn Diagram of Influences

> 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
```

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.

Those same properties make SBDL a natural interface between engineers and AI systems: an SBDL model is a compact, verifiable specification to hand to an AI, and also a form in which an AI can be required to record its own design and decisions, close to the code it writes, so that its work is checked by the compiler and interrogable by a person (see AI & LLMs).

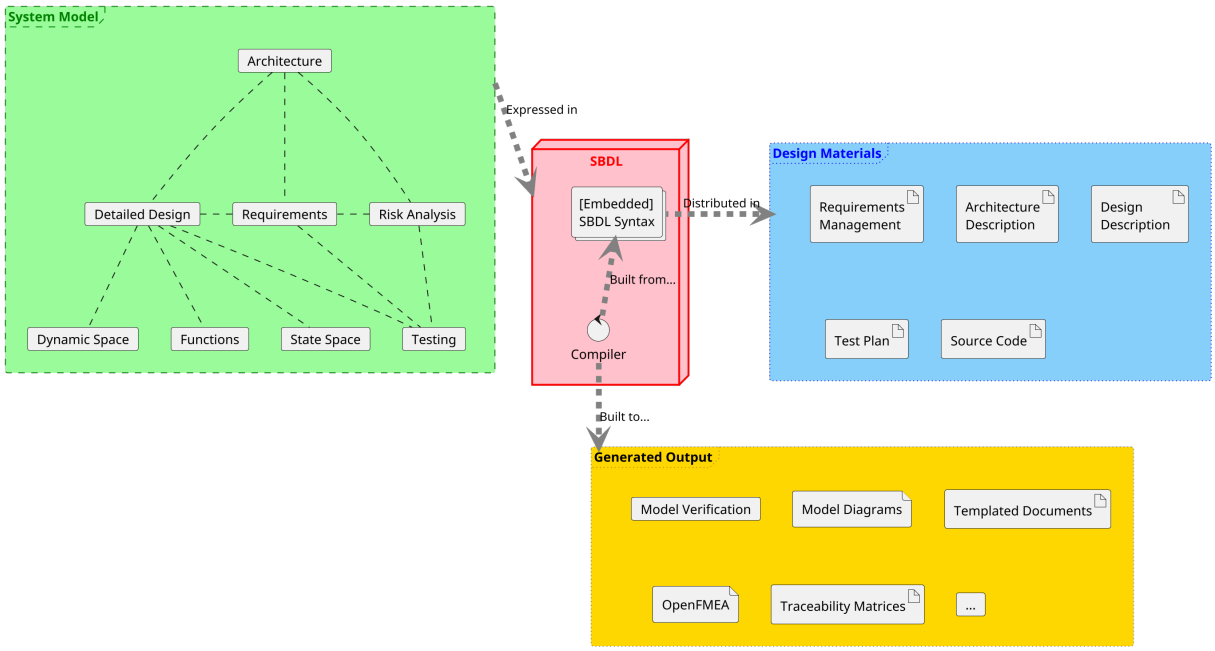


Figure 2: Overview of SBDL

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
- Specifying systems to AI agents, and recording the design structure and decisions an AI makes, as a compiler-verified, human-readable model

Extended usecases:

- Dynamic trace model verification: verify a set of dynamic log trace elements against a model description, as a readable report or as a JSON document of steps for tooling to replay
- Inclusion of more formal aspects, such as OCL, process algebras…
- Automated design review and risk analysis: describe a design in SBDL and have an AI system offer a 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. If you're using VSCode, there's a powerful extension that support syntax highlighting and language verification. 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. Nothing else is needed: the compiler draws its diagrams itself. It is cross-platform (Linux, Windows …).

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

```
python -m pip install sbdl
```

or directly from SBDL.dev:

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

This will install 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
```

Diagram Rendering The compiler draws every diagram natively, as SVG: nothing beyond Python is needed. Six styles change the look (−C native_diagram_style with classic , slate , solarized−dark, blueprint, studio or deck), and sbdl −m diagram:config −o − lists every setting a diagram reads, with its current value.

Naming a .png output file rasterises the SVG through an external tool, rsvg−convert by default (Debian/Ubuntu: librsvg2−bin).

PlantUML remains available as an alternative backend, selected with −C sbdl_renderer plantuml (or the VS Code *Renderer* setting). It must then be installed and on the path (Debian/Ubuntu: sudo apt install plantuml). Remote PlantUML rendering is possible with −C plantuml_remote_render 1, but beware: this sends content from your models to an external rendering server (though this can be configured).

VSCode Integration

A *Visual Studio Code* Extension is available that provides SBDL integration, including:

- Syntax Highlighting
- Language Server (symbol lookup, diagnostics, live semantic verification …)
- Compilation shortcuts (in GUI and via command palette)
- Live Diagram: a diagram beside the editor that redraws on save; clicking an element opens its definition, and an Options popover sets diagram properties
- Model Explorer, table views, and trace verification as a table

Download and install the SBDL Visual Studio Code Extension (or install it from the VSCode Marketplace / OVSX Marketplace).

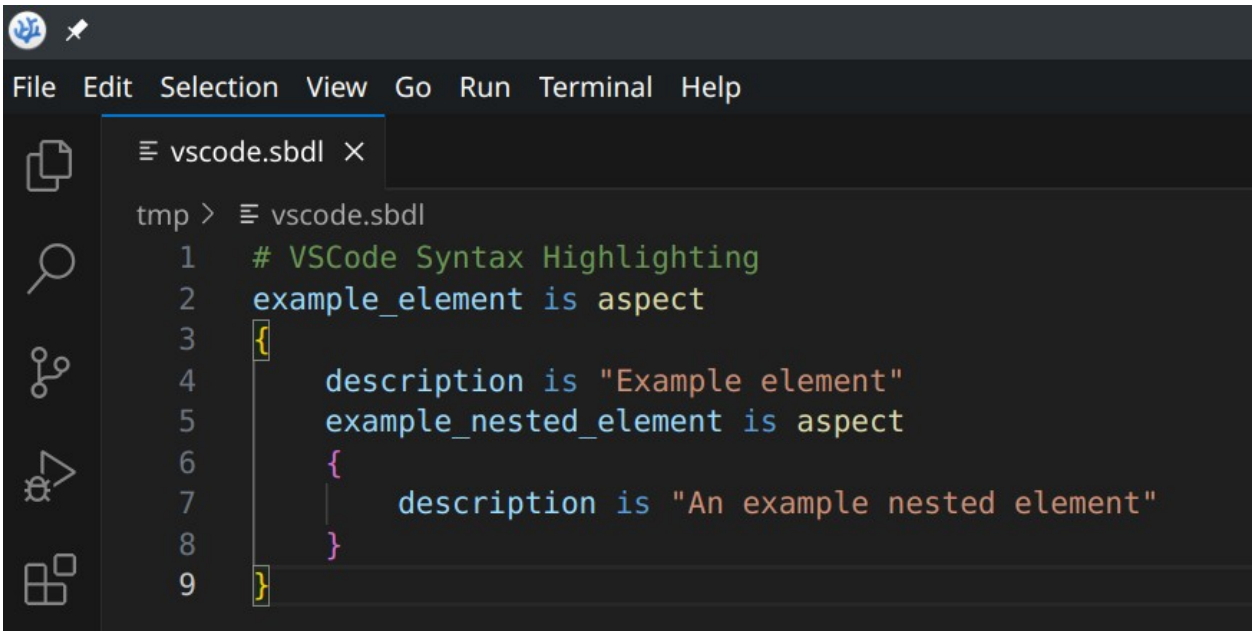


Figure 3: VSCode Syntax Highlighting

After installation, syntax highlighting is available immediately.

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

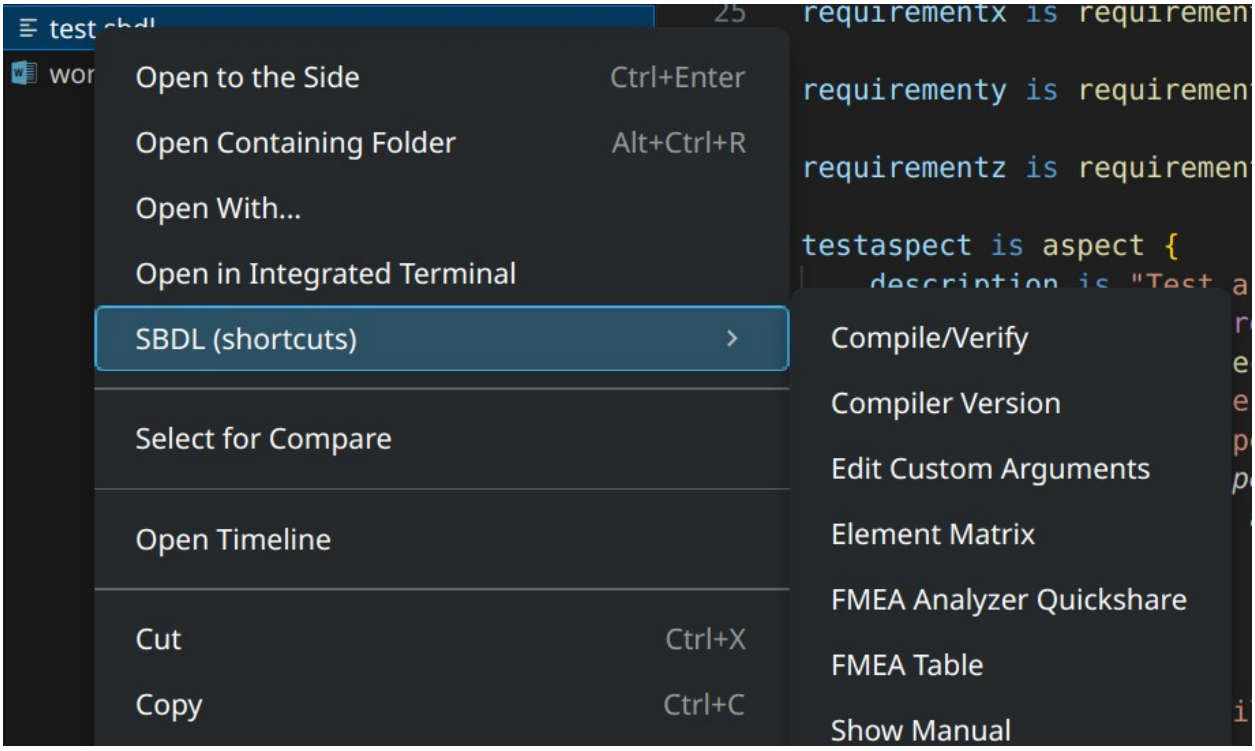


Figure 4: VSCode Explorer Context Menu



Figure 5: VSCode SBDL Button

… or by pressing the SBDL button (top-right of interface) / keyboard shortcut (default: alt+shift+s, or the chord ctrl+k alt+s where that is taken).

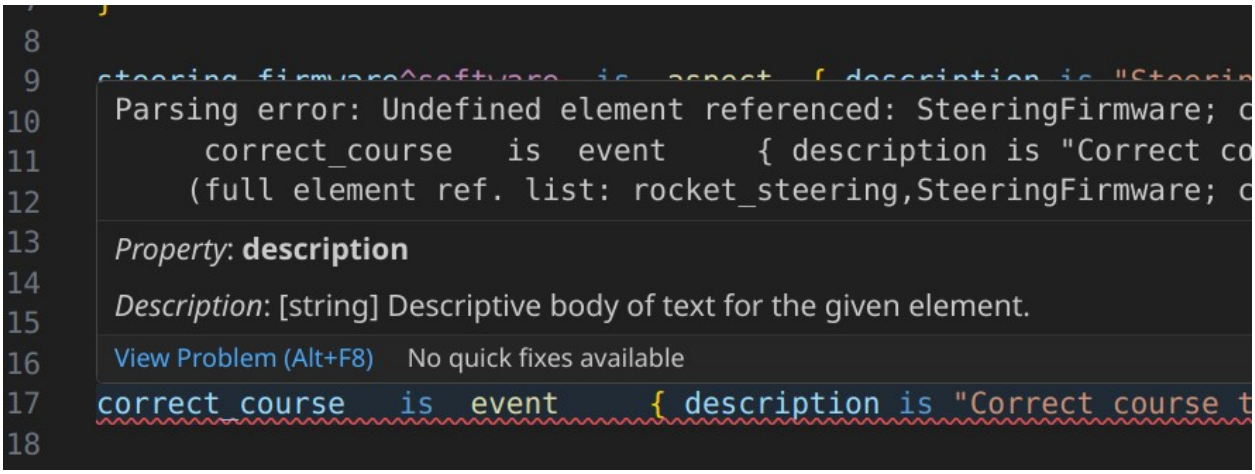


Figure 6: VSCode Language Server

The Language Server supports:

- Definition / Reference lookup
- Diagnostics (red-squigly underline for issues)
- Hover-hints (move cursor over symbols for additional information)

Extension settings can be customized using the VSCode configuration, which can be found in: File->Preferences->Settings->Extensions->SBDL

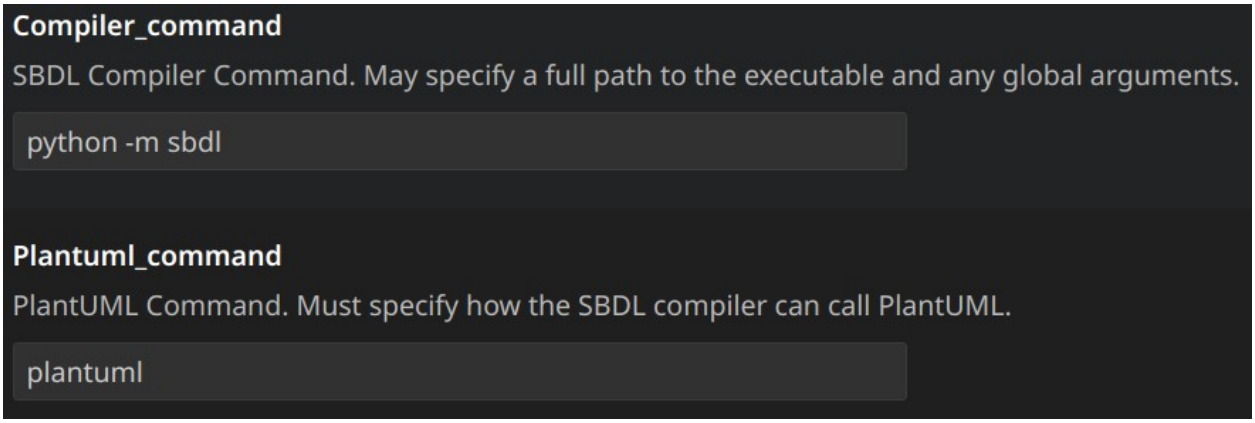


Figure 7: VSCode Extension Configuration

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'.

Renderer: The diagram backend. Empty (the default) uses the compiler's own setting, which draws diagrams natively.

Plantuml_command: Command to invoke PlantUML, used only when the PlantUML backend is selected (see Diagram Rendering). This might be simply 'plantuml' or 'java -jar /path/to/jar'.

Enable_LSP: Enable use of the Language Server (live semantic analysis, error diagnostics etc)

Build the example

The SBDL example is the model of an office elevator, spread across the materials an engineering team keeps (SBDL files, a requirements spreadsheet, a Word document, C++ sources, captured logs) and compiled by its build into diagrams, an FMEA, other exports and a generated report. It exercises most of the language and its README maps each material to the features it shows. It resides in its own repository: you can find the repository here.

To build the SBDL example you will need the SBDL compiler and CMake; the diagrams and the report need nothing more. A C++ compiler builds the optional (but recommended) example program and its tests.

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, the 'build' directory holds the verified model, the diagrams, the FMEA and other exports, and the generated report (Elevator_Report.html). You can view the pre-compiled output here and the report here.

Demonstration

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

(interactive-demo-here)

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.

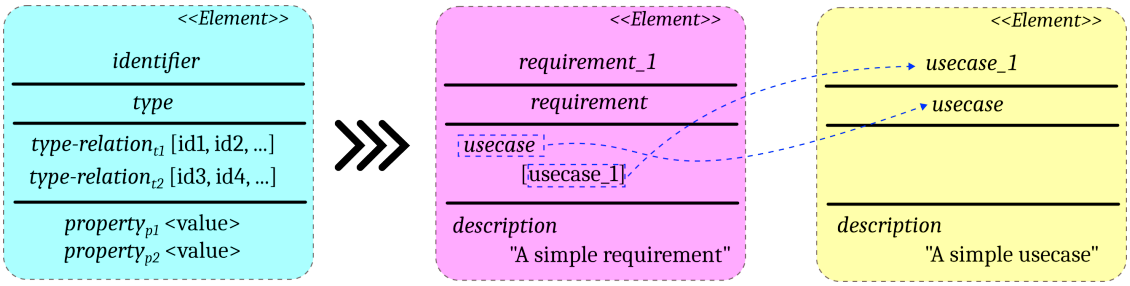


Figure 8: Simple representation of SBDL model elements and relations

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.

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

Syntax at a Glance

Construct	Example	Section
Element definition	some_id is aspect { description is "..."; related is a,b }	Core Language
Raw string	description is [[[multi—line "raw" text]]]	Strings
Stereotype (element, relation, property)	fw^software is aspect { ... }, related is other^controls, width is 20^mm	Stereotyping
Reserved stereotype	retry^loop is event { condition is "n < 3" ... }, related is other^inherit, related is maybe^*weak	Reserved Stereotypes

Construct	Example	Section
Control flow (events)	check^condition is event { condition is "ok"; alternative is my_fail; }	Modelling Behaviour with Events
Custom property	custom:my_property is "..."	Custom Properties
Relation from either side	req is requirement { aspect is a } is equivalent to a is aspect { requirement is req }	Bidirectional Relation Definition
Nesting (implicit parent)	outer is aspect { inner is aspect { ... } }	Nested Statements
Relation operators || and ~|	a is aspect { ... } followed, on its own line, by || and then b is requirement { ... } relates b to a; ~| relates back to the start of the chain	Relationship Operators
Element copy	new_id is copy of source_id { description is "..." }	Element Copying
Custom type	customtype Name is aspect { ... } with relation:required is requirement<1..3>	Custom Types
Embedded statement	// @sbdl some_id is event { ... } (in any host file)	Embedding Statements
Native file marker	#!sbdl as the first line of an SBDL-only file	Native Files vs Embedded Syntax
Embedded block	@sbdl−begin … @sbdl−end	Embedding Blocks
Line continuation (embedded)	@sbdl some_id is requirement { \	Multi-line Statements
File dependency	depends { Ref is "file_a.sbdl" }	File/Unit Content Distribution
Default relations	using { aspect is some_aspect }	Unit Scope Management
Namespace prefix	scope { identifier is PackageName }	Unit Scope Management
Relation hash check	requirement is some_requirement~41575	Relationship Hash Checks
Directive	[@DIRECTIVE_NAME:argument1,argument2]	Directives
Cross-reference	[@other_id:description], optional form [@other_id?default:severity?default]	Cross-Referencing
Coupling to host code	[@CFUNC] is event { description is "[@−LINE]" }	Coupling to Embedded Definitions

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 42;
   some_relation_type 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.

There are two broad categories of attributes specified within an SBDL element definition:

- Properties: strings, values etc attached to the element itself
- Type Relations: identifiers of other other elements to be related by their type
- Parent/Child: identifiers of elements of the same type, serving as a nested parent/child

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;
```

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_system}
booster_requirement_1  is requirement { description is "The rocket boster shall fire ...";
aspect is rocket_booster; parent is system_requirement_1}
steering_requirement_1 is requirement { description is "The rocket shall be steerable ...";
aspect is rocket_steering; parent is system_requirement_1}
```

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; ev
fire_booster    is event    { description is "Fire boosters"; aspect is rocket_booster }
correct_course  is event    { description is "Correct course trajectory"; aspect is rocket_
```

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,rocke
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, their relations, and properties 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;
```

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
```

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

Properties may also be annotated with a stereotype. This often serves as an indication of the unit used, for example:

```
some_part is aspect { max_width is 20^mm }
```

This associated the stereotype unit of 'mm' with the value '20' for 'max_width'. (see customtype to define new types with arbitrary properties)

Reserved Stereotypes Most stereotypes are informative: they are shown on diagrams (as <<type::stereotype>> on an element, as a label on a relation, as a unit on a value) and can be required by a custom type. The following stereotypes are recognised by the compiler and change its behaviour:

Stereotype	On	Types	Effect
^loop	element	event	Child events repeat while the 'condition' holds; 'condition' is required.
^condition	element	event	Child events run if the 'condition' holds, the 'alternative' events otherwise; 'condition' is required.
^parallel	element	event	Child events run concurrently.
^port	element	interface	Drawn as a port on its aspect rather than as a provided interface.
^inherit	relation	aspect	Generalisation: rendered with an inheritance arrow between aspects.
^compose	relation	aspect	Composition: rendered with a composition arrow between aspects.
^*weak	relation	any	Suffix: the target may be undefined; an unresolved weak relation is skipped instead of reported.

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 "Som
```

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 }
```

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_boo
}
```

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.

Element Copying One element may be defined as a copy of another element. In such a case, the new element has a new identifier but copies the type, properties and relations of the source element. The new element may then extend the content of the source element.

```
source_element is aspect {
    description is "This is the source"
    custom:testprop is "Test property"
}
new_element is copy of source_element {
    description is "This is the new element"
}
```

In the above example, 'new_element' is a copy of 'source_element'. It retains the type of aspect and copies both the description and custom:testprop properties. The description, however, is override in 'new_element'.

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:

- 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:optional is requirement;
    relation:required is aspect;
    property:required is UID;
    property:optional is Rationale;
    default_property  is "Default text here";
}
new_functional_element is FunctionalElement {
  requirement is System_Requirement1
  aspect is System_Block1
  UID is "XXXX:YYYY"
}
```

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

- derives from base type 'aspect'
- instances are valid in place of an 'aspect' instance
- may be linked to an element of type 'requirement'
- *must* be linked to an element of type 'aspect'
- must have a property 'UUID'
- may have a property 'Rationale'
- by default has a property 'default_property' with a default string assigned (which may be overwritten)

Multiple Base Types New custom types may specialise more than one base type.

For example:

```
customtype FunctionalElement is aspect, requirement {
    ...
}
```

Above, the custom type 'FunctionalElement' specialises both the aspect *and* requirement types. Instances of 'FunctionalElement' will therefore be valid in place of instances of both aspect and requirement.

Relation Cardinality New custom types may specify that relations adhere to a certain cardinality.

For example:

```
customtype NewType is aspect {
  relation:required is requirement<1..10>;
  relation:optional is aspect<1..3>;
}
```

The above asserts that all elements of type 'NewType' must have a relation to at least one requirement, up to a maximum of 10. In addition, the new element may optionally specify a relation to an aspect; if it does, there must be at least one, and a maximum of 3 such relations.

Required Stereotypes New custom types may require that defined relations and properties adhere to specified stereotype constraints.

For example:

```
customtype NewType is aspect {
  relation:required is requirement<1..5>^complies_with;
  property:required is max_width^mm;
}
```

In the above, new elements of type 'NewType' may *only* make relations to requirement elements using the stereotype 'complies_with' – relations not using this stereotype are invalid. In addition, the property 'max_width' is required, and must be of stereotype 'mm':

```
some_new_elem is NewType {
  requirement is some_requirement_element^complies_with;
  max_width   is 10^mm;
}
```

Verifiers Custom types may also define custom verifiers. Such verifiers check additional compliance rules with the definition of the type and instances thereof, and are written in an embedded language (Python is supported by default).

```
customtype CustomRequirementType is requirement {
    verifier is [[[#!python
        # two variables are available:
        # self — this SBDL element
        # elements — all SBDL elements (dict)
        if "requirement" in self.identifier().lower():
            return "Word 'requirement' not allowed in element identifier"
        else:
            return True
        # return: True if everything is valid
        # or: a string containing a description of the error
    ]]]
}
```

In the above example:

- The verifier is specified in Python (#!python perfix required)
- Two variables are exposed by the SBDL compiler: self (current element instance) and elements (dict of all SBDL elements)
- The identifier of the current element is checked to see if it contains the string 'requirement'
- … if it does, an error is thrown by returning a descriptive string
- … otherwise, True is returned (valid)

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_fir
  ...
}
```

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

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 { ... }
```

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.

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" \
}
```

File/Unit Content Distribution Content maybe distributed in different files and depended upon by others. This is achieved by using the 'depends' keyword.

File A ('file_a.sbdl')

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

File B

```
depends {
  RefForFileA is "file_a.sbdl"
}

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

The depends statement above indicates that File B is dependent upon File A. After the depends statement, all of the elements of File A are available in File B.

The property name 'RefForFileA' is arbitrary and currently informative only.

A depends statement may cite multiple dependencies:

```
depends {
  RefForFileA  is "file_a.sbdl"
  SomeOtherRef is "file_b.sbdl"
}
```

Or be simplified using the raw-string syntax:

```
depends { [[[path/to/some/file.sbdl]]] }
```

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
PackageName::example_requirement2 is requirement { aspect is example_aspect; description is
```

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_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.

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
```

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.

A missing element or a missing property is a compilation error. Where the absence is legitimate rather than a mistake, a question mark after the element or the property (or both) makes that part optional; whatever follows the question mark, up to the next separator, is substituted instead. Nothing after the question mark substitutes an empty string:

```
req_a is requirement { description is "Severity: [@some_other_elem_id:severity?not rated]"
req_b is requirement { description is "Mitigated by [@some_control:description?]" }
# missing property, empty string
req_c is requirement { description is "See [@optional_elem?none:description]" }
# missing element, default text
```

The element default applies when the element does not exist; the property default when the element exists but lacks the property. Defaults are literal text (not expanded further) and an element default cannot contain the ':' separator. The plain form without a question mark remains the recommended default, as it catches mistyped identifiers.

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_
```

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 an element was defined, and underpins the traceability of every element back to its material of origin.

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

Model Output Generation

Using the SBDL compiler, the SBDL statements shown in the preceding sections can be aggregated from distributed sources and verified for correctness. Several different model output views are available; a selection of which are shown below.

Requirements Diagram Model requirements content can be automatically rendered by the compiler into a requirements diagram (−m diagram:requirement), as depicted here. This is an adapted form of a SysML-style requirement diagram.

Aspect Diagram Model aspect content can be automatically rendered by the compiler into an aspect diagram (−m diagram:aspect), as depicted here. This is an adapted form of a SysML-style block diagram.

Function Diagram Model function content can be automatically rendered by the compiler into a function diagram (−m diagram:function), as depicted here. This is an adapted form of a SysML-style sequence diagram.

Function-Process Diagram Model function content can also be automatically rendered by the compiler into a function-process diagram (−m diagram:function−process), as depicted here. This is an adapted form of a SysML-style activity diagram.

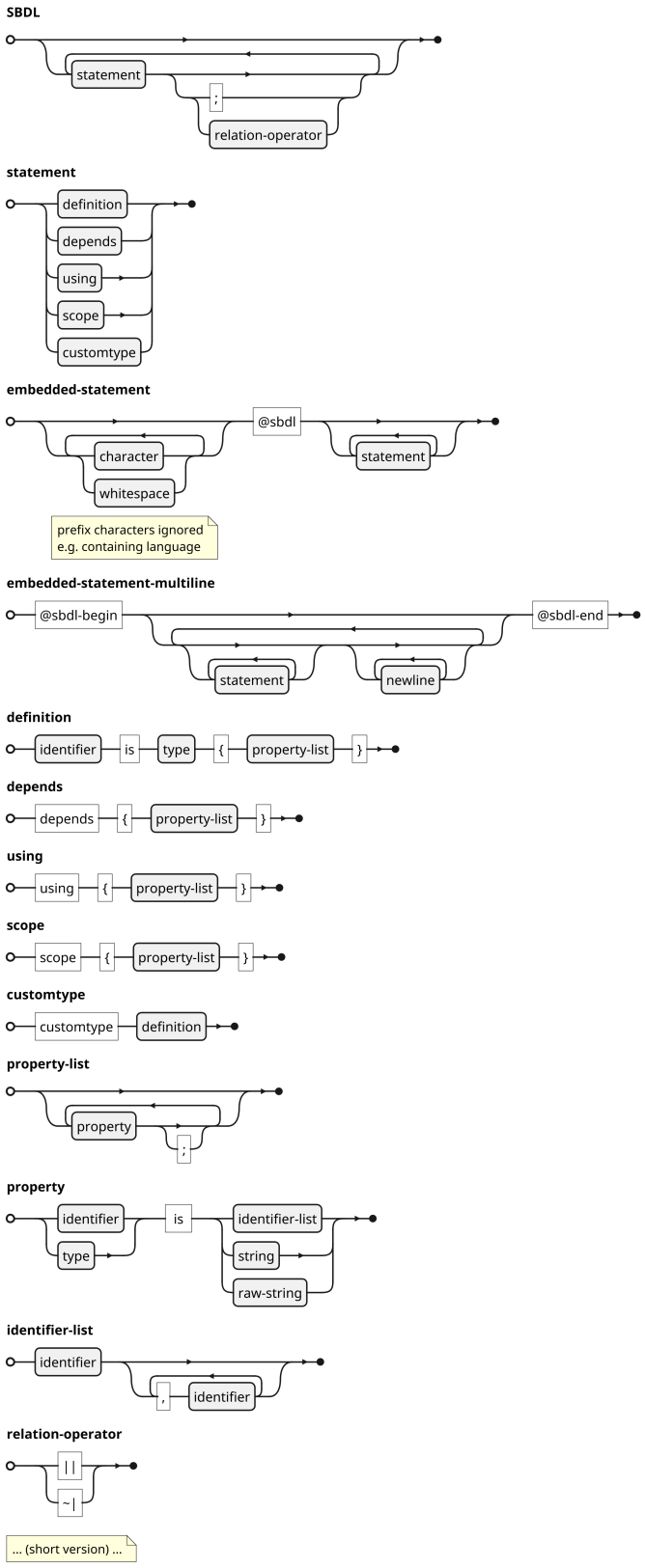


Figure 9: SBDL Syntax Diagram (short version)

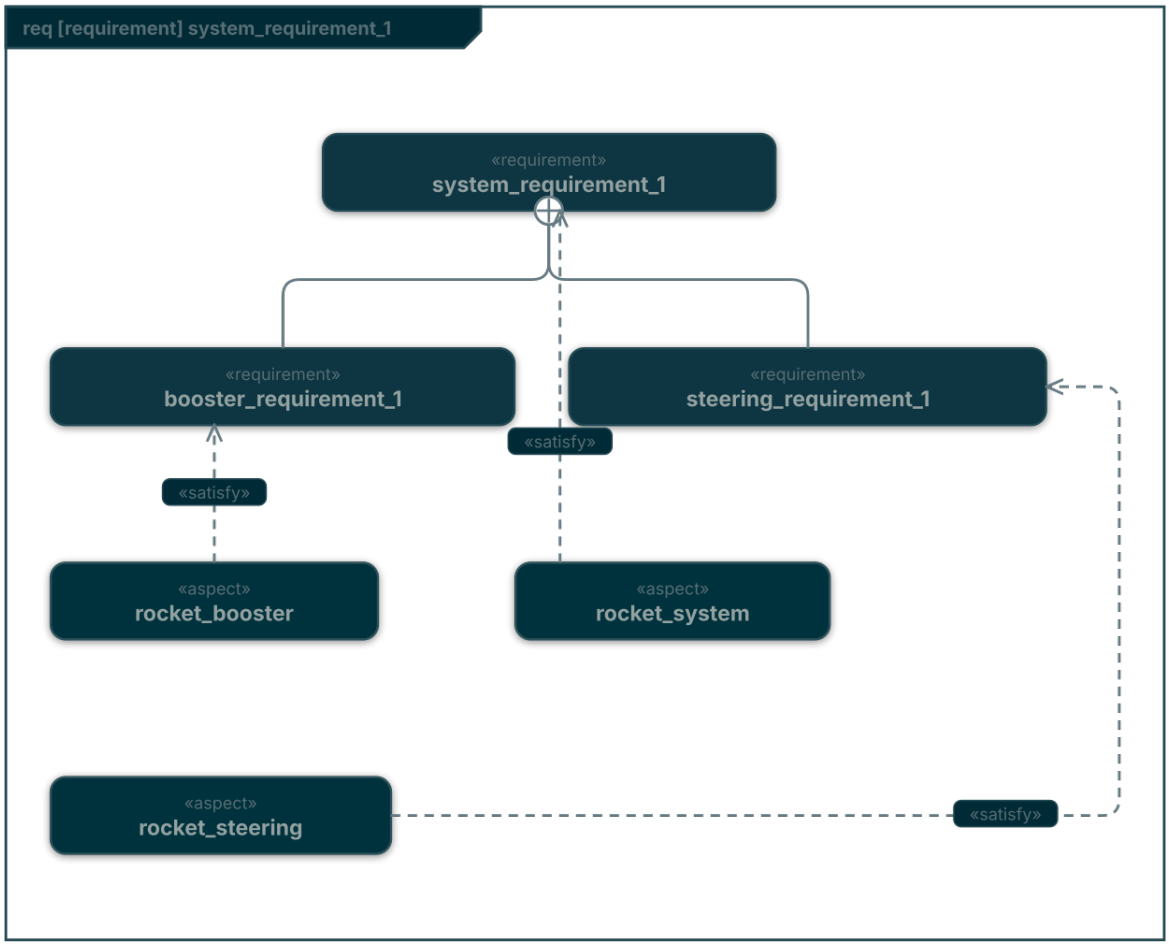


Figure 10: Requirements Model View

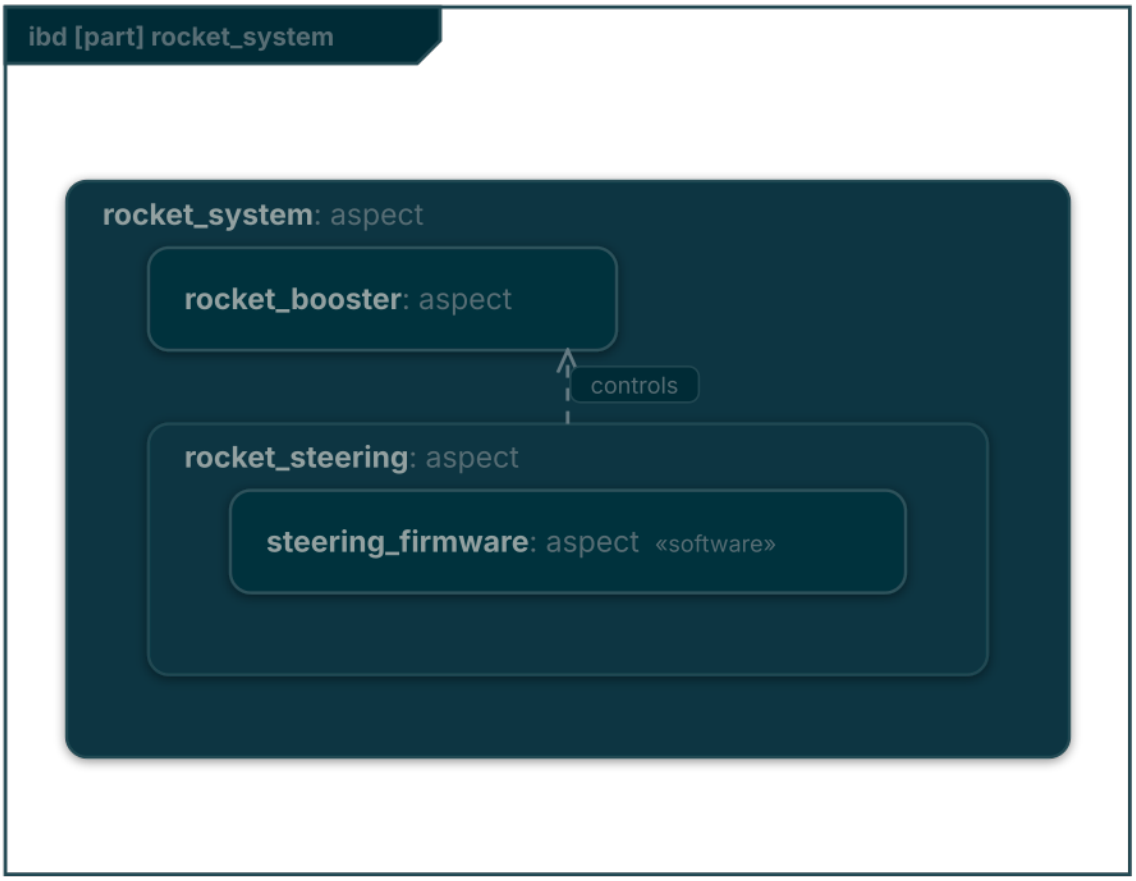


Figure 11: Aspect Model View

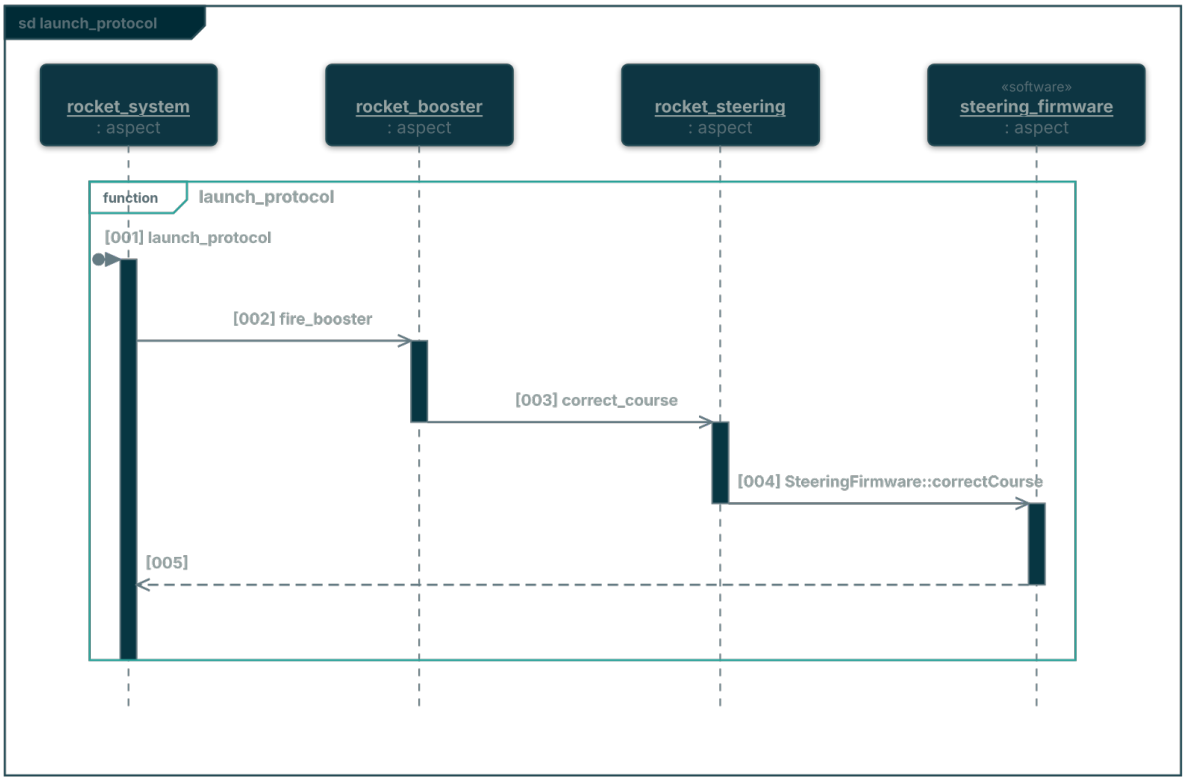


Figure 12: Function Model View

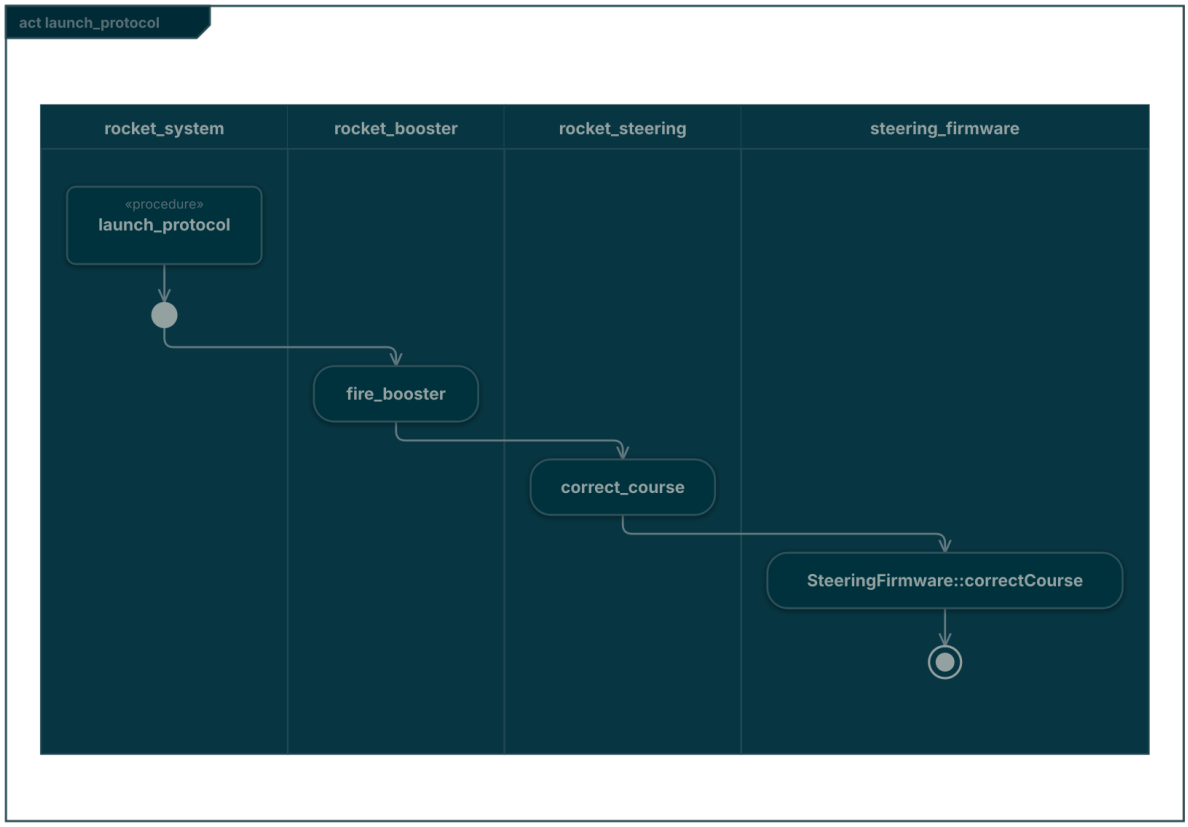


Figure 13: Function-Process Model View

State Diagram Model state content can be automatically rendered by the compiler into a state diagram (−m diagram:state), as depicted here. This is an adapted form of a SysML-style state diagram.

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.

SBDL and SysML v2

Both SBDL and SysML v2 are textual, declarative languages that model a system as typed elements joined by typed relations. Where they diverge is in *scope* and *workflow*:

Scope SysML v2 is a comprehensive OMG standard: a large, layered metamodel (KerML + SysML), a standard library, a repository and API, and a growing tool ecosystem. SBDL is deliberately minimal — a small set of element types and a terse surface you can read and write without training. SBDL trades expressive completeness for low ceremony; SysML v2 makes the opposite trade.

Locality A SysML v2 model is authored as its own artefacts. SBDL is designed to be *embedded* — definitions can live as annotations inside the very materials they describe: source code, markdown, requirement documents, issue trackers. Each definition is anchored to its point of definition, so the model stays close to (and honest about) the thing it models:

```
CourseStatus SteeringFirmware::correctCourse ( ... )
{
  // @sbdl correct_course is event { description is "Correct course"; aspect is steering_fi
  ...
}
```

This in-place, distributed definition is SBDL's central idea, and the point at which the two languages diverge most sharply.

Failure Analysis Failure Mode and Effect Analysis is a first-class part of the SBDL metamodel — modes, effects, causes, controls, detections and actions are native element types, woven into the same graph as requirements and behaviour. In SysML v2, FMEA is something you construct *on top of* the language rather than something the language provides.

Complementary, not competing Crucially, SBDL *targets* SysML rather than replacing it. The compiler can render the captured model as SysML-style requirement, block, sequence, activity and state views, and can also emit SysML v2 textual notation directly (see the sysmlv2 output mode). The two are natural partners: capture behaviour tersely and in-place with SBDL, close to where the work happens, and generate standard SysML v2 — along with FMEA and documentation — for downstream tools and stakeholders.

> In short: reach for SysML v2 when you need the full expressive power of the standard and its ecosystem; reach for SBDL when you want a verified behavioural model captured with minimal ceremony, embedded in the materials you already maintain — and let the compiler bridge to SysML when you need it.

Metamodel

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.

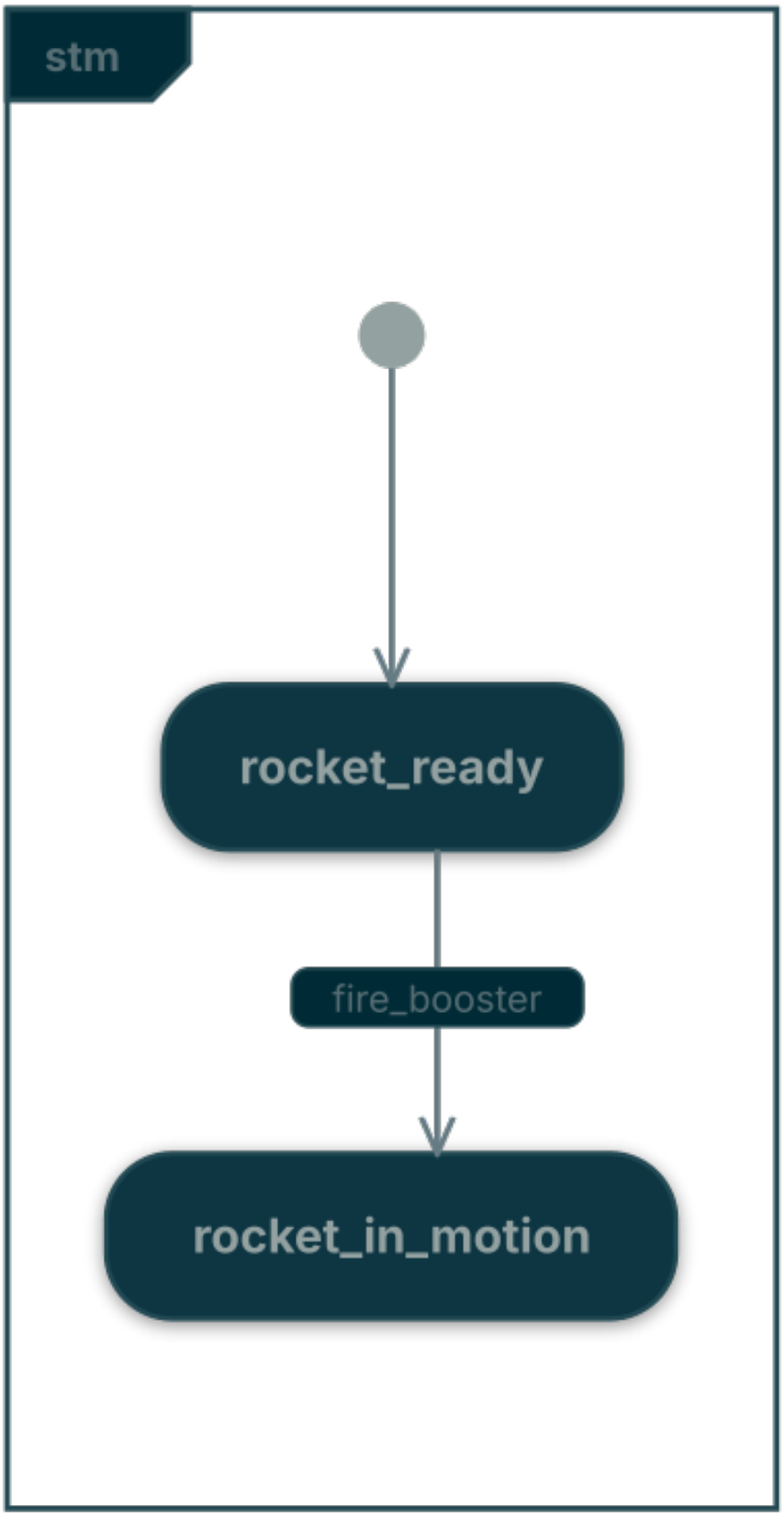
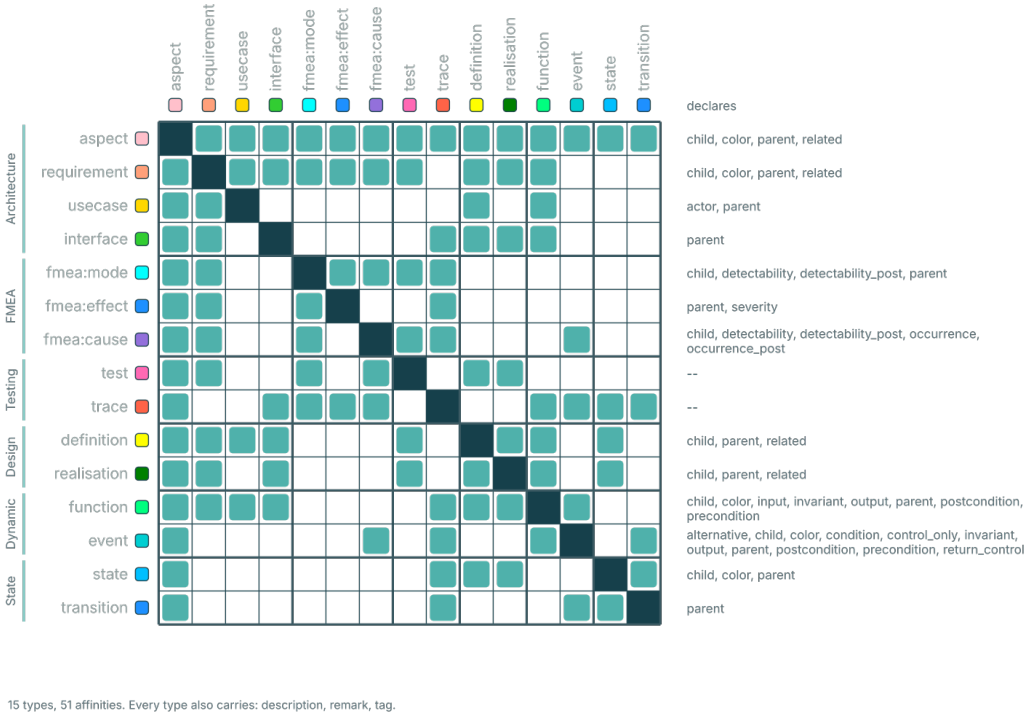


Figure 14: State Model View



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 • gr

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.

related: [identifier] Indicates a related element.

Notes:

- 'parent'/'child' is containment; 'related' is a plain association, stereotyped 'inherit' or 'compose' for generalisation or composition.
- An element stereotype is shown as <<aspect::stereotype>>; 'color' fills the box.

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 • group • 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.

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

color: [string] Color associated with element.

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

related: [identifier] Indicates a related element.

usecase Element

Description:

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

Relations:

requirement • aspect • group • 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.

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

Notes:

- ‘actor’ names the actor (default ‘User’); use cases are drawn per aspect with their functions.

interface Element

Description:

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

Relations:

requirement • interface • aspect • group • 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.

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

Notes:

- Related to the aspect exposing it; a relation stereotype becomes the connector label.
- Stereotype ‘port’ draws it as a port instead of a provided interface.

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 • aspec

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.

detectability: [number] Numerical rating on the interval [1(best)..10(worst)] indicating quality of a failure detection. On a cause, replaces its mode’s for that cause’s FMEA rows.

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

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

Notes:

- Joined with its effects, causes, controls and detections into FMEA rows; ratings are 1 (best) to 10 (worst).

fmea:effect Element

Description:

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

Relations:

requirement • fmea:mode • aspect • group • 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 • gr

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.

detectability: [number] Numerical rating on the interval [1(best)..10(worst)] indicating quality of a failure detection. On a cause, replaces its mode's for that cause's FMEA rows.

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

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.

Notes:

- An optional 'detectability' (and 'detectability_post') replaces the mode's on this cause's FMEA rows; without one the mode's applies.

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 • group • trace • definition • 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 • group • test • definition •

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 • group • definition •

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 • group • definition •

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 • group •

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 verify dynamic behaviour against the statically defined behaviour model.

Relations:

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

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.

Notes:

- Written into logs at run time and checked against the model by '-m trace:verify'.

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 • fmea:control • fmea:detection • fmea:action-control • fmea:action-detection • aspect • group • 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.

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

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

related: [identifier] Indicates a related element.

realisation Element

Description:

Final realisation of a particular aspect of prescriptive design.

Relations:

requirement • definition • function • state • interface • aspect • group • 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.

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

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

related: [identifier] Indicates a related 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 • group • trace • definition • 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.

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

color: [string] Color associated with element.

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

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

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

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

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

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

Notes:

- Child functions nest as frames; related events are the steps, in definition order.
- 'input' and 'output' become the entry and exit gates; 'precondition', 'postcondition' and 'invariant' are carried into outputs as documentation.

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. Events may be stereotyped as "loop" or "parallel" if children should be interpreted as such.

Relations:

fmea:cause • aspect • group • 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.

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

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

color: [string] Color associated with element.

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

control_only: [*] Presence indicates control flow only

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

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

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

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

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

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

Notes:

- A step of a function; child events are its sub-steps. Stereotypes 'loop', 'condition' and 'parallel' with the 'condition' and 'alternative' attributes express control flow, see Modelling Behaviour with Events.
- Without an aspect relation a step stays on the aspect of the previous step.

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 • group • 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.

Notes:

- Related transitions form the state machine; 'parent'/'child' nests composite states.

transition Element

Description:

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

Relations:

state • event • aspect • group • 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.

Notes:

- 'state' takes exactly one ordered pair: state is from,to (verified).
- 'event' names the trigger shown on the transition.

Model management Elements Model management elements organise the model itself rather than describe the system. They carry no system semantics: removing one changes what can be selected or presented, not what the system is.

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) or be related to any other element type; containment may be declared from either side.

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 • gr

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.

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

related: [identifier] Indicates a related element.

Notes:

- No system semantics: a group only structures the model, and '-fg' filters by it. Membership may be declared from either side.

Modelling Behaviour with Events

A function is performed as a sequence of events, each attributed to the aspect that performs it. Events nest: the children of an event are its sub-steps, and a few stereotypes and properties turn plain nesting into control flow. The function and function-process diagrams render these as sequence and activity constructs.

```
retry_login is function {
  description is "Log in, retrying on failure"
  aspect      is client
  event       is attempt, verify
}
attempt^loop is event {
  description is "Try up to three times"
  aspect      is client
  condition   is "attempts < 3"
  send_credentials is event { description is "Send credentials"; aspect is server }
}
verify^condition is event {
  description is "Check the answer"
  aspect      is server
  condition   is "token valid"
  alternative is reject
  grant is event { description is "Open the session"; aspect is server; output is "session
}
reject is event { description is "Refuse and report"; aspect is client }
```

Construct	Meaning
child events	sequential sub-steps of the parent
condition is "..." alone	guard on the step
condition with child events	optional block, entered when the condition holds
^condition with alternative is e	if/else: children when the condition holds, the alternative events otherwise
^loop with condition	children repeat while the condition holds
^parallel	children run concurrently
control_only is true	control passes to the aspect, no step is shown
return_control is true	control stays with the previous aspect after the step
output is "..."	labels the arrow leaving the step; on a function, its exit gate (input its entry gate)

Construct	Meaning
no aspect relation	the step stays on the aspect of the previous step
color is <name>	colours the step, its lane or its block

The compiler verifies that ^loop and ^condition events carry a condition, and that every alternative is an event.

Compiler

Compiler Design

The compiler's own architecture, design principles and behaviour are modelled in SBDL, with most elements embedded in its source code adjacent to what they describe. The compiler design page is generated from that model.

Application Programming Interface (API)

The SBDL compiler can be used as a Python package to parse SBDL files into native Python objects for progammatic interaction, and to generate SBDL output from progammatically created objects.

API documentation overview | Small example use of the API

Command-Line Interface

```
usage: sbdl.py [-h] [-m operating_mode] [-o output_file] [--version] [--license] [-W {norm
               [--dump-config config_file] [--load-config config_file] [--list-config] [--
               [-fpa element_identifier] [-fd filter_depth] [-ft element_type] [-fi elemen
               [--rpc RPC]
               [source_files ...]

SBDL Version 2.1.8 (DSL Version 26.6). System Behaviour Description Language (SBDL) compil
WWW: https://sbdl.dev. Author: contact@sbdl.dev. License: BSD 3-Clause License (https://sb

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
  --license             Print the license
  -W {normal,all}, -w {normal,all}, --warning {normal,all}
                        Set warning level
  --hidden              Include hidden files and directories when recursing
  -i, --identifier      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)
  --skip-verification   Do not very model semantics
```

```
  --already-processed   Disables certain processing actions (implies --skip-verification).
  --title TITLE         Provide a default title for certain output formats
  --diagram-root element_identifier, --diagram_root element_identifier
                        Root applicable diagram output modes at the named element(s) instea
  -v, --verbose         Enable verbose output during execution
  --progress            Show a progress bar for long compilation stages (terminal only)
  --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
  --list-modes, --listmodes
                        List the operating modes (JSON with --debug)
  --set-config config_option config_value, -C config_option config_value, --setconfig confi
                        Set a named configuration option
  -D name value, --define name value
                        Specify a named global definition
  --template template_file
                        Specify a template file for the 'template:fill' mode
  -fr element_identifier, --filter-related element_identifier
                        Filter everything but those elements with a direct or indirect rela
  -frx element_identifier, --filter-related-x element_identifier
                        Filter everything but those elements with a direct or indirect con
  -fch element_identifier, --filter-children element_identifier
                        Filter everything but those elements which are children of the spe
  -fpa element_identifier, --filter-parents element_identifier
                        Filter everything but those elements which are parental ancestors
  -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 ele
  -fi element_identifier, --filter-identifier element_identifier
                        Filter everything but those elements whose identifiers match the s
  -fpr property_name property_value, --filter-property property_name property_value
                        Filter everything but those elements possessing a named property m
  -fg group_identifier, --filter-group group_identifier
                        Shortcut filter for everything but those elements which are childre
  --sql-filter sql_expr, --sqlfilter sql_expr
                        [from:sql only] Filter elements by raw SQL: either a WHERE expressi
  --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.py <file 1> <file 2> <file n>"
```

Operating Modes

```
                  compile: Parse all specified input files, gather SBDL elements, verify m
          compile:formal: As 'compile', written in the formal grammar
                   query: Compile inputs, then pretty print the results (after filtering)
              matrix:csv: Compile inputs, then write a CSV-formatted representation of the
              tree:json: Compile inputs, then write a JSON-formatted representation of th
              tree:yaml: Compile inputs, then write a YAML-formatted representation of th
                sysmlv2: Compile inputs, then write the model in SysML v2 textual notatio
                    sql: Compile inputs, then write the SBDL elements into a normalised S
               from:sql: Read an SBDL-schema SQL database and write SBDL-formatted output
        from:matrix:csv: Read SBDL-schema CSV-matrix inputs and write SBDL-formatted outp
         from:tree:json: Read SBDL-schema JSON-tree inputs and write SBDL-formatted outpu
         from:tree:yaml: Read SBDL-schema YAML-tree inputs and write SBDL-formatted outpu
                    omi: Compile inputs, then write the model as an OMI (Open Model Inter
               from:omi: Read OMI (Open Model Interchange) documents and write SBDL-form
           openfmea: Compile inputs, then write the FMEA-related content to an OpenFM
  openfmea:portfolio: Compile inputs, then write the FMEA-related content to an OpenFM
           openfmea:csv: Compile inputs, then write the FMEA-related content to a CSV-for
         from:openfmea: Read OpenFMEA-formatted input and write SBDL-formatted output
        template:fill: Compile inputs, then provide an object, 'sbdl', in a Jinja parsi
          trace:verify: Compile inputs, then check the trace elements they contain agai
     trace:verify:json: As 'trace:verify', written as a JSON document: one record per tr
           type-info: Write the metamodel (types, relations, properties) as JSON
         list-import: List the files the inputs depend on or import
                 rpc: Compile inputs, then transmit to the RPC server for processing
               batch: Compile inputs, then dispatch a set of batch jobs specified by
                 lsp: Serve the Language Server Protocol over standard input and outpu
                 mcp: Compile inputs, then serve the model read-only over the Model C
         graph:sigma: Compile inputs, then write Sigma interactive HTML output
diagram:requirement: Compile inputs, then write a SysML-style requirements diagram to
      diagram:aspect: Compile inputs, then write a SysML-style block diagram to render
     diagram:element: Compile inputs, then write a SysML-style block diagram to render
    diagram:function: Compile inputs, then write a SysML-style sequence diagram to ren
diagram:function-process: Compile inputs, then write a SysML-style activity diagram to ren
       diagram:state: Compile inputs, then write a SysML-style state diagram to render
     diagram:usecase: Compile inputs, then write a SysML-style use-case diagram to ren
       diagram:fmea: Compile inputs, then write the FMEA failure chain as staged caus
    diagram:affinity: Write the SBDL type system — types, categories and their affin
      diagram:config: List the native renderer's theme keys, each with its value and
```

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

The mode switch has been changed to specify 'diagram:aspect' 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.

An example template, generic enough to be suitable for use with any model input, can be found here (Markdown version) and here (Word equivalent).

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

Compiler Directives

Available compiler directives, with their arguments and return value in the form [arg1,arg2 ,... -> return]:

Directive	Arguments -> Return	Description
+LINE	−> string	Next line (embedded statement)
−LINE	−> string	Previous line (embedded statement)
=LINE	−> string	Current line (embedded statement)
ABORT	string −>	Abort compilation with error
ADD	int , ... −> int	Sum of arguments
ARG_SEP	−> string	Directive argument separator string (‘,’)
BATCH	string , ... −>	Specify a batch compiler job, of the form: [@BATCH:-m,compile,...]
BATCH&	string , ... −>	Specify a background batch compiler job
BATCH∗	string , ... −>	Specify a batch compiler job, and wait for background jobs before executing
CFUNC	−> string	Most recently defined C/C++ function (embedded statement)
COMPILER_VERSION	−>	Compiler version string
CONCAT	string , ... −> string	Concatentate string arguments
CONTEXT	−> string	Context string (embedded statement)
CPPCLASS	−> string	Most recently defined C++ Class (embedded statement)
CROSS_REFS_AVAILABLE	−> bool	Indicates whether cross references are available
DATE	−> string	Today’s date
DEFIND	string , string , string −>	Create a definition from the result of indexing another name definition
DEFINE	string , string−>	Create a key,value pair compiler definition
DEFINE_APPEND	string , string −>	Append to a named 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
DEFUNC	string , string , string −>	Create a definition from the result of a named compiler function application
DFP	string −> string	Return a string defining a description as the previous line
DIR	−> string	Name of the current directory
DSL_VERSION	−>string	DSL version string
EQUAL	val , val −>	Raise a compiler error if two value arguments are not equal
EXPAND	string −> string	‘Expand’ and sanitize a named definition
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]
FILE	−> string	Name of the current file
FILTER_IDS	string −> string	Takes a regex and return an identifier list of all matching element identifiers
GENERATED_ELEMENTS	−>	Set of output elements generated by other directives (useable only programmatically)
GIT_COMMIT_HASH	string −> string	Return the current (or stated path) Git repository’s short commit hash
GIT_COMMIT_LOG	int , string −> string	Take a log depth and optional path, and return the Git log

Directive	Arguments -> Return	Description
GIT_FILE_DATE	string -> string	Take a file path (relative to the current file; defaults to the current file) and return the ISO date (YYYY-MM-DD) of the last Git commit that modified it
ID_FROM_PROP	string , string -> string	Takes a property name and value. Returns the identifier list of elements with matching properties (which may be an empty string)
IMPORT	string ->	Import contents of another SBDL file
IMPORT_DIRECTIVE	string ->	Import a custom directives file
INSTLI	string , ... -> string	Index a list of string terms separated by whitespaces
LINE	-> string	Current line number
LOAD_CONFIG	string ->	Load a compiler configuration file
LOOKUP	string [, string	-> string] Takes a definition name and an optional default. Returns the value of definition when defined, otherwise the default
MESSAGE	string ->	Show a message (on stdout) during compilation
MKID	string -> string	Make a given string a valid SBDL identifier
MSG	string -> string	Show a message (on stdout) during compilation and replace occurence with the message inline
PARSED_ELEMENTS	->	Set of parsed elements (useable only programatically)
PATH	-> string	Path of the current file
PATH_SEP	-> string	Path separator of current OS
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_RESULT_OUTPUT	->	Output the result of Prolog assertions to stdout
PL_START	->	Startup the Prolog environment (required at least once to use Prolog directives; indempotent)
PL_TRACE_FILE	string ->	Write all generated Prolog facts, assertions and commands to a named file (for external use in Prolog)
PYCLASS	-> string	Most recently defined Python Class (embedded statement)
PYFUNC	-> string	Most recently defined Python function (embedded statement)
REQUIRE_COMPILER_VERSION	string ->	Raise a compiler error if the current compiler version is not at least equal to the argument
REQUIRE_DSL_VERSION	string ->	Raise a compiler error if the DSL 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
RMCOM	string -> string	Remove comments from a string
SELF	-> SBDL_Element	Element object for the current element (for internal cross-referencing)

Directive	Arguments -> Return	Description
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)
SHOW_ALL	−>	Show a message (on stdout) displaying all defined macros
SUB	int , ... −> int	Subtraction of subsequent arguments from the first argument
SYNTHETIC_ELEMENT	−>	Synthetic object class reference, used to generate elements dynamically (usable only programmatically)
TITLE	−> string	Active compilation title
USER	−> string	Current OS user

License

The SBDL compiler and tools are provided under the BSD 3-Clause License. The license text ships with every package and is printed by sbdl −−license.

AI & LLMs

SBDL works in two directions with AI. *As an instruction*, a model is a terse, unambiguous specification an agent can be given to guide implementation. *As a record*, an agent can be required to emit its design (aspects, requirements, decisions, failure modes) as SBDL embedded in, or along side, the code it writes; the compiler verifies that model like any other, and diagrams, traceability and FMEA output can be generated from it, making the AI's design scrutable after the fact.

Give the 'SBDL Expert System' a go!

Prompt your favourite LLM to do the following:

```
Please follow the instructions at: https://sbdl.dev/sbdl_llm_prompt.md
```

(or open the prompt in your browser and copy/paste the content in the LLM interface)

The prompt is designed to work well with Claude and Grok.

Try asking the expert for an SBDL example of a system of your choice.

MCP Server

The compiler can serve a compiled model to AI assistants over the Model Context Protocol, read-only and with no extra dependencies:

```
sbdl model.sbdl -m mcp
```

This listens on http://127.0.0.1:8765/mcp (−C mcp_host, −C mcp_port), and −C mcp_transport stdio suits a client that starts the compiler itself. Five tools are provided: an overview of the model, element search, element descriptions, diagrams and exports. With Claude Code, for example:

```
claude mcp add --transport http sbdl http://127.0.0.1:8765/mcp
```

FAQ

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

Since 2.0 the compiler draws diagrams itself by default, so first try the native backend (−C sbdl_renderer native). If you need PlantUML, try installing its latest version.

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.