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

0.3.18 (SBDL Compiler), 23.03 (SBDL Language) [#6271763]

Contents

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 control to the model (e.g. Git et al)
- Automating the generation of design output material
- Integrating design risk analysis (FMEA) with model evolution

In the near future:

- Dynamic trace model verification: verify a set of dynamic log trace elements against a model description

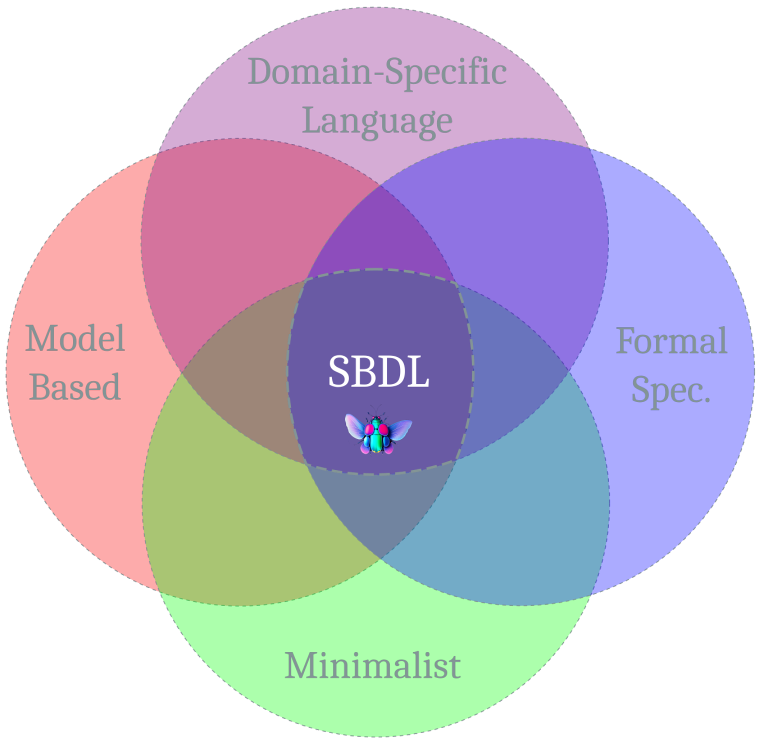


Figure 1: SBDL Venn Diagram of Influences

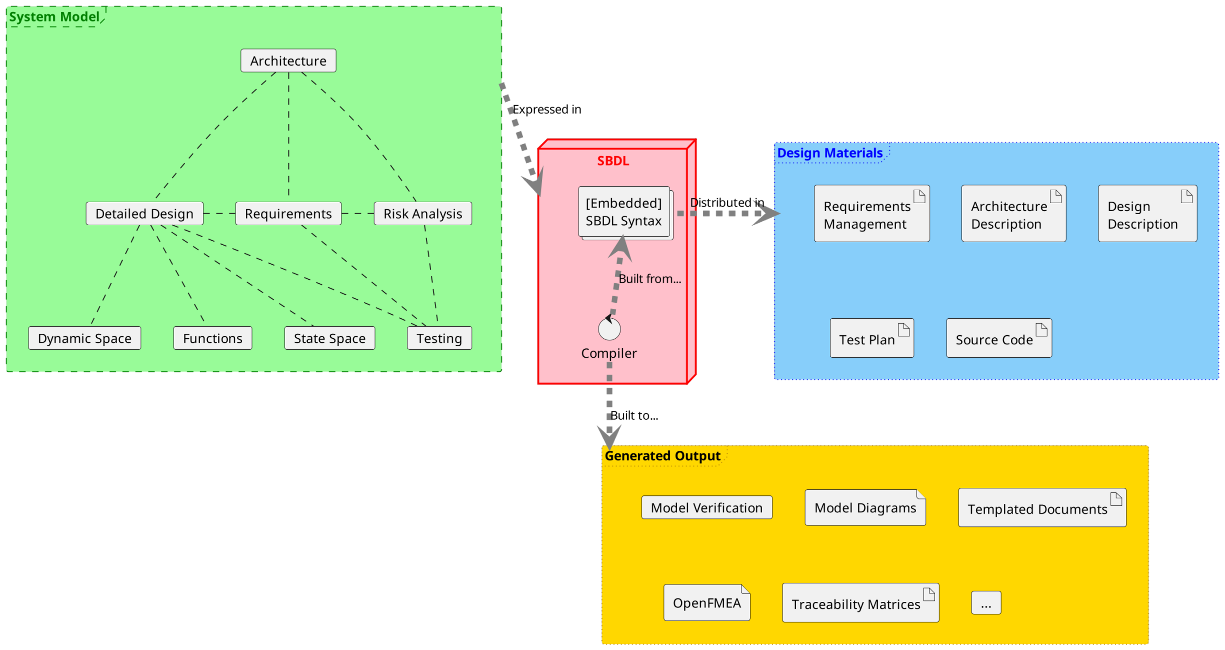


Figure 2: Overview of SBDL

- 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. When Python is installed, the SBDL compiler 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
```

External Dependencies Several of the output (UML) models use PlantUML for rendering and therfore 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
```

Syntax Highlighting To have *Visual Studio Code* syntax highlight SBDL, you need to download and install the SBDL Visual Studio Code Extension.

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 view the repository and its README 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://gitlab.com/mahicks/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 . . .).

Demonstration

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

(interactive-demo-here)

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.

Metamodel

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.

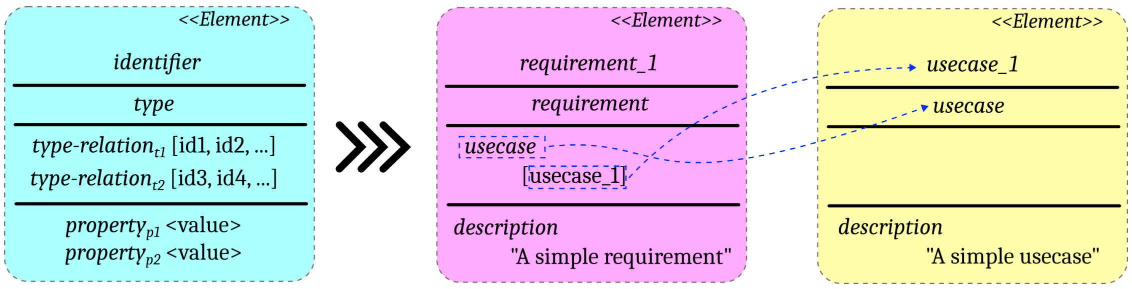


Figure 3: Simple representation of SBDL model elements and relations

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. All elements must define at least a textual description property; but different elements have various different properties which may be attached to them, depending on element 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 *(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 ro
```

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_s
booster_requirement_1  is requirement { description is "The rocket boster shall fire ...";  aspect is ro
steering_requirement_1 is requirement { description is "The rocket shall be steerable ...";  aspect is r
```

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_b
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 attribute definition lists (in this case to order to events).

Stereotyping Elements and their links 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 ro
```

Above, the relation to the rocket_booster sub-system is stereotyped to show that the rocket_steering sub-system *controls* the rocket_booster (^controls).

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').

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.

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.

Application 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
```

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.

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.

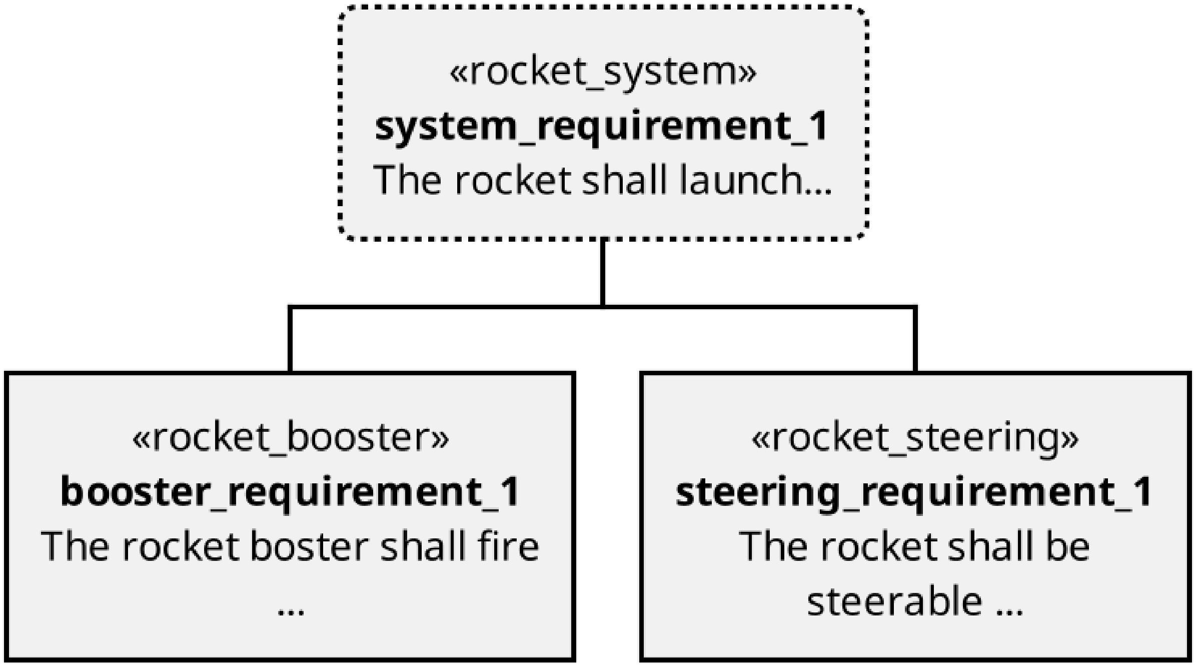


Figure 4: Requirements Model View

Additional Syntax Notes

Bidirectional Link Definition Links 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.

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 { ... }
```

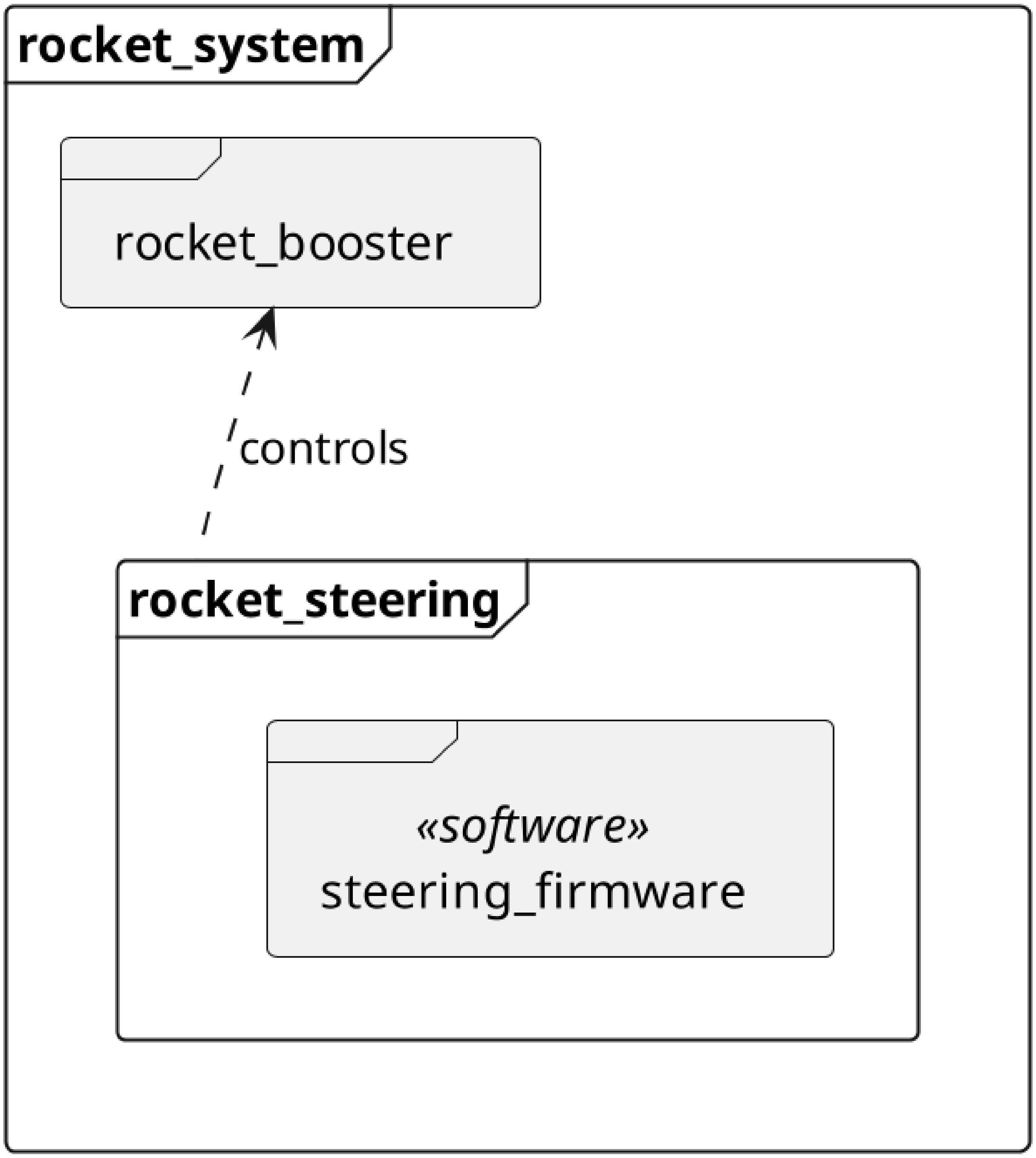


Figure 5: Aspect Model View

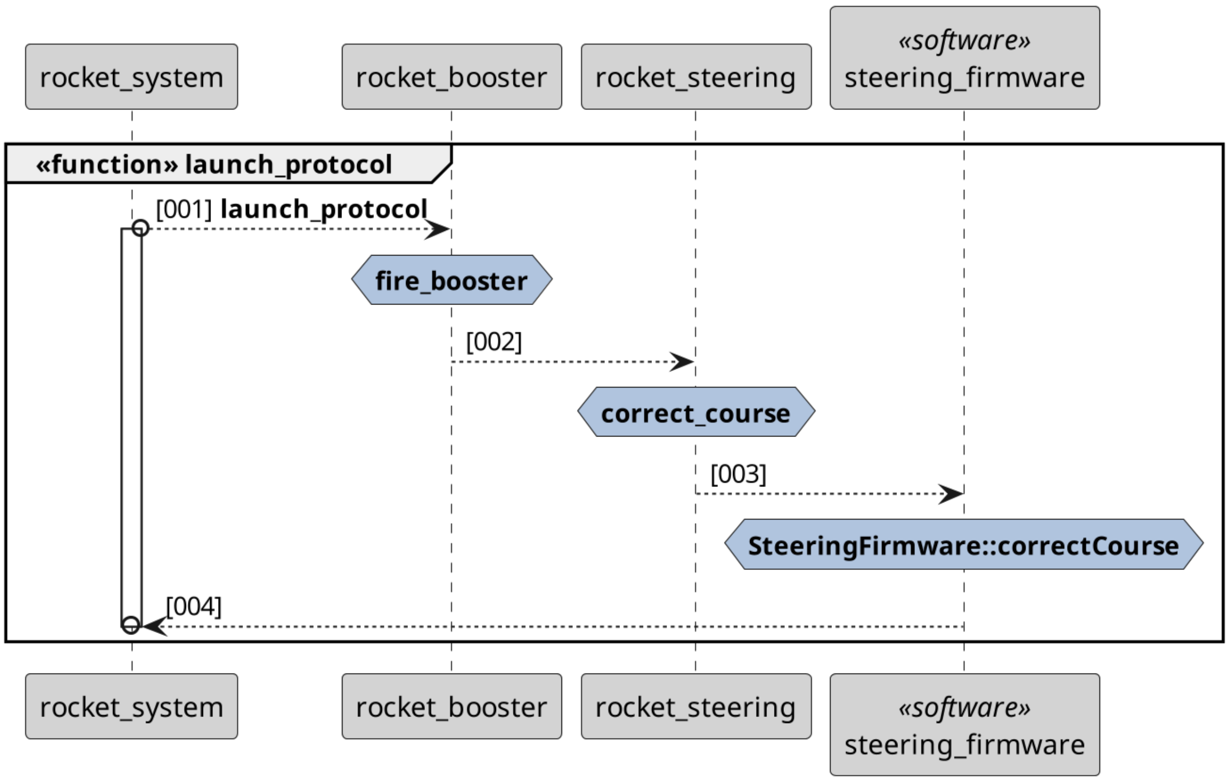


Figure 6: Function Model View

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 statement can be broken across multiple lines using the like continuation symbol: backslash (\).

```
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 by using the custom-property-prefix ('customer:').

```
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 attributes, and, by an edge between two types, the valid links which may be specified between them. Each type (its relations and properties) is also elaborated upon in its own sub-section.

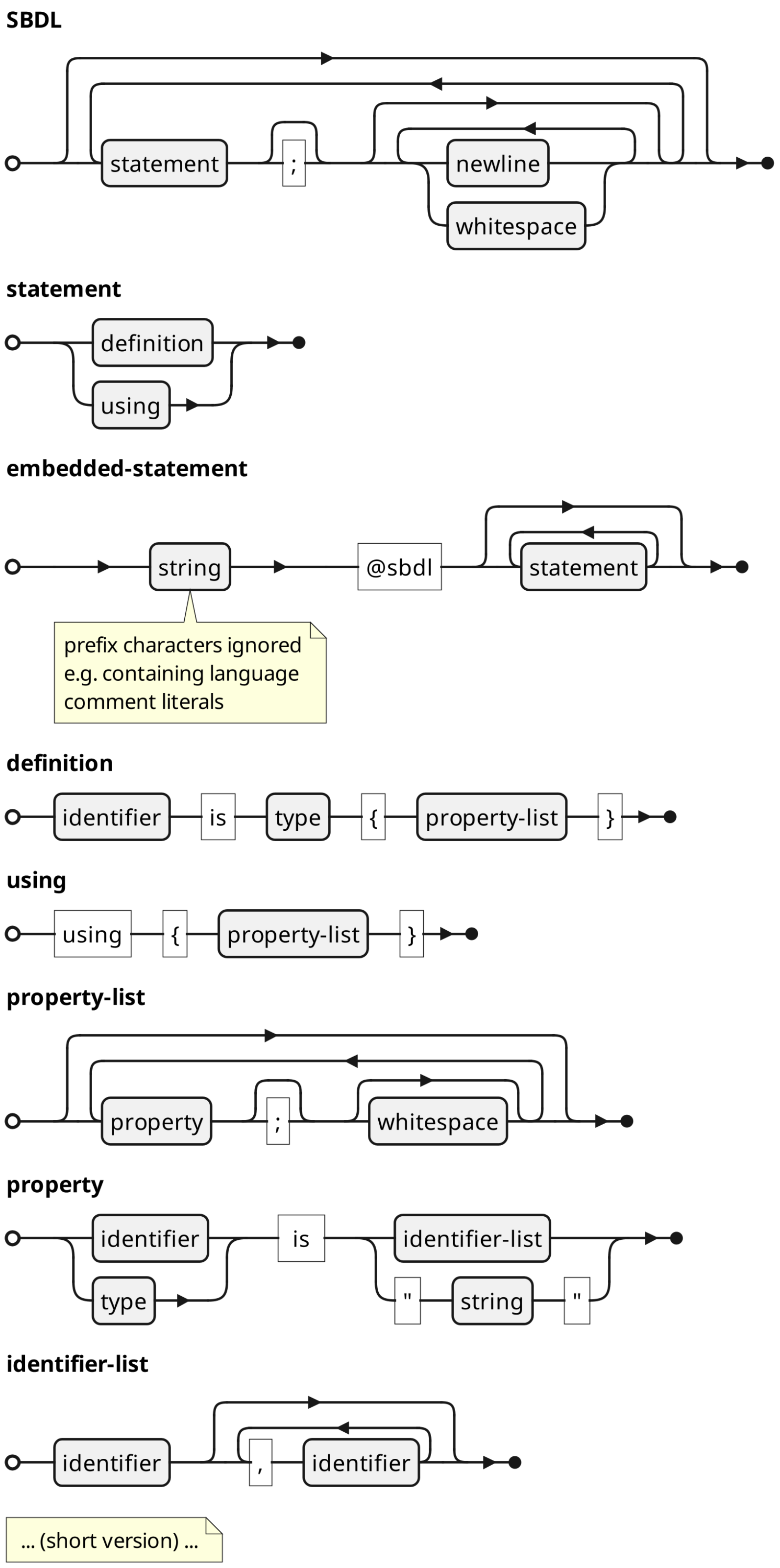
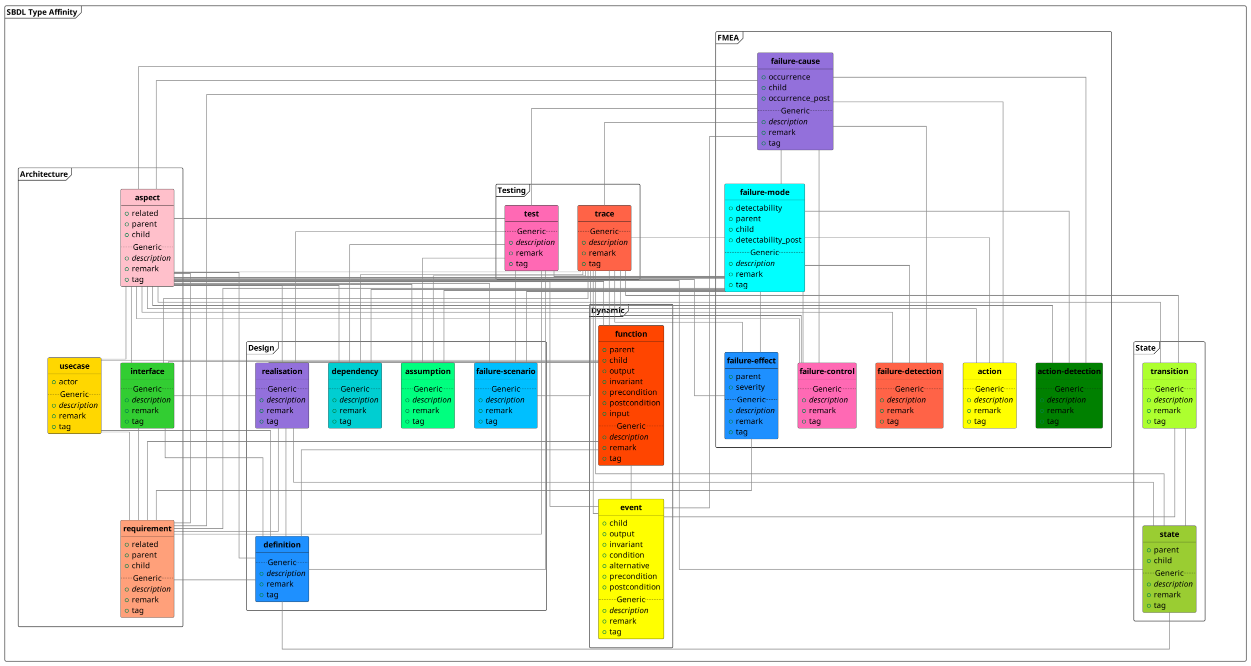


Figure 7: SBDL Syntax Diagram (short version)



(click for larger version)

Element Types & Semantics

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

Each element type is intended for linking 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 links 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 • failure-mode • failure-effect • failure-cause • failure-control • failure-detection • action • action-detection • assumption • dependency • failure-scenario • test • definition • realisation • function • event • state • transition • usecase •
mode • failure-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.

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

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

related: [identifier] Indicates a related element.

requirement Element

Description:

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

Relations:

failure-mode • failure-cause • failure-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.

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

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.

failure-mode Element

Description:

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

Relations:

requirement • aspect • failure-effect • failure-control • failure-detection • test • action • action-detection • assumption • dependency • scenario • failure-cause • 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.

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.

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

failure-effect Element

Description:

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

Relations:

requirement • failure-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.

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

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

failure-cause Element

Description:

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

Relations:

requirement • aspect • failure-mode • failure-control • failure-detection • test • action • 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_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.

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

failure-control Element

Description:

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

Relations:

failure-mode • failure-cause • 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.

failure-detection Element

Description:

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

Relations:

failure-mode • failure-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.

action Element

Description:

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

Relations:

failure-mode • failure-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.

action-detection Element

Description:

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

Relations:

failure-mode • failure-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.

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

assumption Element

Description:

Assumption made by a particular part of a design. Maybe be a source for a failre mode.

Relations:

failure-mode • aspect • trace • 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.

dependency Element

Description:

Dependency required by a particular part of a design. Maybe be a source for a failre mode.

Relations:

failure-mode • aspect • trace • 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.

failure-scenario Element

Description:

A particular scenario of use which would potentially result in a failure mode.

Relations:

failure-mode • aspect • trace • 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.

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.

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 • failure-scenario • assumption • dependency • failure-mode • failure-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:

failure-cause • failure-mode • failure-effect • failure-control • assumption • dependency • failure-scenario • function • transition • eve

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.

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

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

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

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

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

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

output: [string] Output produced by 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:

failure-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.

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

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

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

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

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

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

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

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.

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] [--dumpconfig config_file] [--loadconfig config_file] [--listconfig] [--setconfi
            [-D macro_name macro_value] [--trace [trace_files ...]] [--template template_file] [--filte
            [--filter_linked element_identifier] [--filter_children element_identifier] [--filter_paren
            [--filter_depth filter_depth] [--filter_type element_type] [--filter_identifier element_ide
            [--filter_property property_name property_value] [--rpc RPC]
            [source_files ...]

SBDL Version 0.3.18 (DSL Version 23.03). 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
  --skiperrors          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
  --dumpconfig config_file
                        Dump the internal configuration to a named JSON file
  --loadconfig config_file
                        Load the internal configuration from a named JSON file
  --listconfig          List internal configuration options
  --setconfig config_option config_value
                        Set a named configuration option
  -D macro_name macro_value
                        Define a named global macro
  --trace [trace_files ...]
                        Provide a trace file to be processed
  --template template_file
                        Specify a template file for the 'template-fill' mode
  --filter_connected element_identifier, -fc element_identifier
                        Filter everything but those elements with a direct or indirect connection to th
                        parents/children]
  --filter_linked element_identifier, -fl element_identifier
                        Filter everything but those elements with a direct or indirect connection to th
                        parents/children]
  --filter_children element_identifier, -fch element_identifier
                        Filter everything but those elements which are children of the specified elemen
  --filter_parents element_identifier, -fpa element_identifier
                        Filter everything but those elements which are parental ancestors of the specif
  --filter_depth filter_depth, -fd filter_depth
                        Maximum depth for filters which pursue links (natural number)
  --filter_type element_type, -ft element_type
                        Filter everything but those elements which are of the specified element type (r
  --filter_identifier element_identifier, -fi element_identifier
                        Filter everything but those elements whose identifiers match the specified stri
  --filter_property property_name property_value, -fpr property_name property_value
                        Filter everything but those elements possessing a named property matching the s
  --rpc RPC             Remote Procedure Call to be used by RPC-based modes

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

---------------
Operating Modes
---------------
           aggregate: Parse all specified input files, gather SBDL elements, perform semantic checks, a
               query: Aggregate inputs, then pretty print the results (after filtering)
           matrixcsv: Aggregate inputs, then write a CSV-formatted representation of the SBDL elements
          matrixjson: Aggregate inputs, then write a JSON-formatted representation of the SBDL elements
            openfmea: Aggregate inputs, then write the FMEA-related content to an OpenFMEA-formatted ou
  openfmea-portfolio: Aggregate inputs, then write the FMEA-related content to an OpenFMEA Portfolio-fo
        fromopenfmea: Read OpenFMEA-formatted input and write SBDL-formatted output
             fmeacsv: Aggregate inputs, then write the FMEA-related content to a CSV-formatted ouput
     network-diagram: Aggregate inputs, then write a PNG-formatted output, visually representing the ne
 requirement-diagram: Aggregate inputs, then write a SysML-style requirements diagram to PlantUML-forma
      aspect-diagram: Aggregate inputs, then write a SysML-style block diagram to PlantUML-formatted ou
    function-diagram: Aggregate inputs, then write a SysML-style sequence diagram to PlantUML-formatted
       state-diagram: Aggregate inputs, then write a SysML-style state diagram to PlantUML-formatted ou
     usecase-diagram: Aggregate inputs, then write a SysML-style use-case diagram to PlantUML-formatted
       template-fill: Aggregate inputs, then provide an object, 'sbdl', in a Jinja parsing environment
      remote-process: Aggregate inputs, then transmit to the RPC server for processing by the specific
```

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.

Aggregate Inputs The core mode of the compiler is to *aggregate* 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 aggregate -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.

aggregate: 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 aggregate 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
```

(aggregated SBDL will then be written to STDOUT)

Importantly, the aggregated 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 aggregate 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 aggregate 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:

- SELF
- SELF_ATTR
- ABORT
- MESSAGE
- MSG
- DATE

- ADD
- SUB
- CONCAT
- INSTLI
- SHOW_ALL
- RMCOM
- MKID
- DFP
- REQUIRE_DSL_VERSION
- REQUIRE_DSL_VERSION_EXACT
- PATH
- FILE
- DIR
- CONTEXT
- =LINE
- -LINE
- +LINE
- LINE
- IMPORT
- DEFINE
- DEFINE_APPEND
- DEFUNC
- DEFIND
- PARSED_ELEMENTS
- GENERATED_ELEMENTS
- SYNTHETIC_ELEMENT
- CROSS_REFS_AVAILABLE
- EXTEND_CAUSE
- CPPCLASS
- PYCLASS
- CFUNC
- PYFUNC

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: *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)