LCF 2.0

Per Larsson, Olav Bandmann

LCF 2.0

Language Definition

Per Larsson and Olav Bandmann

PROVER®

LCF-LDD 1.6 2021-05-10 Prover Technology

REVISION HISTORY

- Version 0.2 2016-10-05 PL
  Initial version.
- Version 1.0 2018-03-13 PL
  Major updates for types, tables and compact format. Updating LCF version to 1.1.
- Version 1.1 2018-10-10 PL
  Added multi line strings.
- Version 1.2 2020-07-10 PL
  Removed previous relaxed JSON syntax. Updating LCF version to 1.2.
- Version 1.3 2020-10-29 PL
  Added section on regular expressions. Added sections for all the LMC formats.
- Version 1.4 2021-03-04 PL
  Moved all LMC formats to separate document. Added examples for all LCF formats. Updating LCF version to 2.0.
- Version 1.5 2021-05-05 PL
  Various corrections and additions in response to review comments.
- Version 1.6 2021-05-10 PL
  Added document-data required for publication and corrected miscellaneous typos.

CONTENTS

LIST OF FIGURES

LISTINGS

PART I
Syntax descriptions

1 Introduction

This document contains the specification of the Layout Configuration Format (LCF), version 2.0, which is intended for specification of data for railway signaling systems. As will be explained in detail in this document, LCF consists of a set of interrelated sub-formats targeting the following type of data.

- Generic data for a class of railyard installations, sharing the same signaling principles and hardware equipment. For example, data concerning the types of signals, switches, etc. which may be used in a railyard installation.
- Specification of the railyard layout for a specific installation.
- Table-like data, typically describing relations of railyard entities in an installation.

LCF was originally created for supporting formal verification of signaling systems. However, the format may be used for a wider range of purposes, offering a consistent and succint notation designed to be easy to read both manually and with software applications.

1.1 Outline

The document is divided in two parts. The first part includes sections 1 – 7 and contains grammar listings and informal descriptions of the LCF formats. An overview of LCF is given in Section 2. LCF consists currently of four sub-formats, which are described in Sections 3 – 7.

The second part includes sections 8 – 13 and contains formal requirements for the different formats. This part is intended as a specification for software that shall read LCF files. Preliminary notations for this part are in Sections 8 – 9. Sections 10 – 13, contains the requirements for each of the formats. The final Section 14 gives symbolic names to all formal requirements.

1.2 Definitions

The following acronyms are used in this document.

AST	Abstract Syntax Tree
JSON	Java Script Object Notation
RJSON	Relaxed JSON
LCF	Layout Configuration Format
RFC	Request For Comments
UTF	Unicode Transformation Format

1.3 Contact

If you have questions or comments concerning this document, or want to report an error or omission, you are welcome to contact our support staff at:

support@prover.com

For other inquiries please refer to the contact details on our Web page:

`https://www.prover.com/about-us/contact-us/`

We welcome your comments on our products, so please don't hesitate to contact us.

2 Overview and basic concepts

This section briefly describes all the LCF formats. All these formats share the same underlying lexical format, the Java Script Object Notation (JSON). JSON is further described in Section 3.

LCF is a set of formats specifically designed to describe railyard interlocking systems. The term *package* is used to mean generic data for a class of railyard installations, sharing the same signaling principles, hardware equipment, etc. The term *project* is used for data concerning a specific railyard. A project always depends on the package it belongs to. LCF is designed to describe both generic package data and instantiated project data. Because package data is reused for several projects and because some uses of LCF only need a subset of the full LCF configuration, LCF is divided into sub-formats to enable processing subsets of the data. Some of these sub-formats are intended to be written in a single file, while others can be split into multiple files.

2.1 Railyard graphs

A *railyard graph* is described by a set of nodes and edges. Railyard objects, like signals, track circuit delimiters etc. are modeled by data objects which are placed in the nodes of the graph. Paths in the graph can be used to model routes, overlaps etc. Areas, i.e. sub-graphs, are used to model track circuits, level crossings, etc. The main purpose of the LCF formats is to describe different aspects of such graphs.

2.2 LCF formats

This document specifies four LCF sub-formats, however two of these formats are alternative notations for describing the same data. The sub-formats are shown in Figure 1 and are listed briefly below, but will be described in detail in subsequent sections. All files containing LCF formats are syntactically JSON files and shall use the `.json` extension.

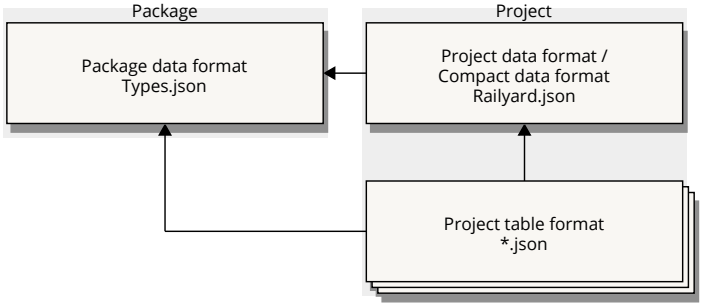

Figure 1 LCF formats overview, edges represents dependencies. The "Project data format" and the "Compact data format" are alternative formats for specifying a railyard layout.

Package data format
: A file containing the format is usually named `Types.json`. This format contains generic types for railyard objects and tables that may be used in a specific installation. It is further described in in Section 4. The format is conceptually one file per package, but reusable parts can be loaded into a main package data file using import directives. All other LCF formats depend on the types given in the package data format.

Project data format
: A file containing the format is usually named `Railyard.json`. This format describes a specific railyard, in the form of a railyard graph, containing instances of the types in the package data format. The format is described in Section 5. This format is written as one file per project.

Compact data format
: A file containing the format is usually named `Railyard.json`, as for the previous item. The project data format in its explicit form is primarily intended to be interpreted by automatic methods, hence it is not suitable for human inspection. The compact format, described in Section 7, offers an alternative syntax for railyard graphs that is more concise and adapted for human readers.

Project table format
: This format contains spreadsheet-like data conforming to the declared table types in the package data format. The format is described in Section 6. This format can be separated into several files for each project, typically one file per table.

3 JSON and RJSON

The JSON syntax is used for LCF, i.e. LCF uses JSON as its syntactic format, but adds additional constraints for the data contents. See the RFC [1] for the current version of the JSON specification. JSON is a succinct yet expressive format well adapted for representing structured data. However, it is mainly intended to be read using software and the format can be awkward to inspect manually when this is required.

The Relaxed JSON format (RJSON) extends JSON with syntactic sugar to facilitate human inspection. Note that only alternative syntax is added, RJSON can be translated to JSON without loss of data. The full specification for RJSON is in [2], only a summary of the syntax for JSON and RJSON is given in the following two subsections. In the rest of this document we will mainly use strict JSON with the exception of examples for those LCF formats that can be relevant to review.

Quite the opposite to manual reading, reading RJSON using software is more complex than the case with JSON. Translation from RJSON to JSON is therefore recommended to be performed with a single dedicated program for this task. LCF files may use RJSON extensions but applications that read LCF files can always presuppose strict JSON input. The translator is used in a preprocessing step if RJSON extensions was used for the original input.

3.1 JSON syntax

In summary, a JSON file contains literal values and collections. The literals are the Boolean constants, strings, numbers and a `null` value. There are two kinds of collections in RJSON:

1. A *list* is an ordered sequence of *elements*.
2. An *object* or *dictionary* is an unordered sequence of *members*, which are pairs of a string key and an associated value.

Elements and member values may be literals or collections, nested to arbitrary levels.

Constants The literal constants of JSON are the Boolean truth constants and the special `null` value, which is typically used as a generic place-holder for an empty or unknown value in an object.

```
true   false   null
```

Numbers Numbers are written in decimal notation and may contain a fractional part and/or an exponential part written in scientific notation.

```
25   -88   256.77   -12.34   2.19e-5   250e9
```

Strings Strings in JSON are enclosed in double quotes and must be written in a single line. Multi-line strings can be represented by inserting occurrences of `\n`, encoding a newline character. There are also such escape sequences for special characters like tabs, carriage returns, etc., and also for arbitrary Unicode characters.

```
"A string with newline \n, quote \", and tab \t character"
```

Lists Lists are enclosed in square brackets, with comma-separated elements.

```
["element", 2, null, ["nested element", true]]
```

Objects Objects are enclosed in curly braces, with comma-separated members. A colon is used to separate the member key from its value.

```
{ "key1" : 88,
  "key2" : [true, false],
  "nested object" : {"a" : 3} }
```

3.2 RJSON extensions

As previously explained, RJSON extends JSON with alternative syntactical forms for facilitating manual reading. See [2] for the full specification of RJSON, below is a quick summary.

Comments RJSON adds line comments starting with a hash-mark #.

```
# This is a comment
```

String extensions RJSON adds extra forms of string literals. A string containing only alphanumerical characters and a restricted set of punctuation characters may be written without the enclosing quotes. This is mainly for use with dictionary keys.

```
unquoted-string
```

Double quoted strings are allowed to stretch multiple lines, giving a readable layout of longer texts.

```
"A double-quoted, multi-line string. Trailing and initial spaces
 in continued lines are collapsed to one space. A backslash-\
 escaped linebreak continues the text without separating space.

 An empty line is interpreted as a newline character."
```

Layout sensitive collections Using line-breaks and indentation it is possible to omit some of the explicit separators used for JSON collections. This feature can increase the readability of a JSON file with deeply nested lists and objects.

Each element in a *layout formatted list* is prefixed with a hyphen. The elements are separated by line-breaks and shall be indented to the same level.

```
- element
- 2
- null
-
  - "nested element"
  - true
```

The members in a *layout formatted object* are, similar to list elements, separated by line-breaks and indented to the same level.

```
key1 : 88
key2 :
  - true
  - false
"nested object" :
  a : 3
```

Finally, note that RJSON style collections may be mixed with explicit JSON forms, and a layout formatted collection can contain explicitly formatted collections (but not vice versa).

3.3 JSON Schema

JSON is a generic format for storing data. A JSON *schema* or *format* defines a restricted class of JSON input files. BNF-style grammars, specially adapted for JSON, are used for describing the LCF formats, with the following notations:

`json`	JSON literals, separators and delimiters are represented verbatim
`ident`	Non-terminals are identifiers which must start with a letter (and may not be a JSON literal like `null`)
`'op'`	Single quotes are used to distinguish terminals from grammar notations
e_1`|`e_2	Choice, the first alternative may start with an initial `|`.
e_1 e_2	Concatenation, see remarks below
`expr*`	*Comma-separated* iteration zero or more times
`expr+`	*Comma-separated* iteration one or more times
`expr?`	Optional item, see remarks below

A question-mark denotes in general an optional item. When used after a member name in an object, it means that the complete member (including a preceding comma) is optional.

Grammar rules are written on the form `ident = expr`. Comments in grammar listings starts with `//`. Parentheses are used for grouping. The `|` operator has maximal scope. Operators `+`, `*` and `?` have normally minimal scope, but apply wider to expressions enclosed by parentheses, curly braces or square brackets.

Concatenation has a different meaning for object members than in a standard BNF grammar. The members in a JSON object, and hence also in LCF, are unordered. The given order of object members in grammar listings in subsequent sections is the recommended, but not required, order for printing a specified format.

3.3.1 Predefined identifiers

The following non-terminals have a predefined meaning in grammar listings.

`string`, `real`, `int`, `bool`
: These are JSON strings, reals, integers, and booleans, respectively. The JSON `null` value is not included in these types.

`nstring`
: A non-empty JSON string.

`chars`
: A sequence of characters allowed inside the double quotes in a JSON string. This non-terminal is used when specifying JSON strings with constrained content.

`id`, `ref`
: Both of these are aliases for `nstring`, non-empty strings. Using `id` is a hint that the string introduces a new name for an item. Using `ref` is a hint that the string references an item named elsewhere.

3.3.2 Object descriptions

All objects in LCF may optionally be extended with a string valued member `"descr"`. For convenience this member is not explicit in the grammar rules. This member has no effect on the interpretation of an configuration but may be used by an application for presentation purposes.

4 Package Data Format

The package data format specifies types of railyard objects that may occur in a specific installation. A package shall have a single top-level package data file, it may however be composed of multiple package data files using import directives as explained below. All the other LCF sub-formats depends on a specified package data file.

4.1 Format

The following grammar specifies the syntax of the package data format.

Listing 1 Package data format

```
package-data =
{ "format"       : "LCF-2.0-package-data",
  "package"      : id,
  "imports"?     : [nstring+],
  "node-types"   : [node-type*],
  "object-types" : [object-type*],
  "user-types"   : [user-type*],
  "union-types"  : [union-type*],
  "table-types"  : [table-type*] }

node-type =
{ "id"        : id,
  "degree"    : int,
  "traversal" : [[int, int]*] }

object-type =
{ "id"                 : id,
  "allowed-node-types" : [ref*],
  "required-attrs"     : [nstring*] }

user-type =
{ "id" : id, "base-type" : ref, "def" : string }

union-type =
{ "id" : id, "user-base-types" : [ref+] }

table-type =
{ "id"        : id,
  "primary"?  : bool,
  "signature" : signature,
  "def"       : string }

signature = [[nstring, column-type]+]

column-type =
  | nullable-atomic-type
  | [atomic-type]

atomic-type = ref | json-type

nullable-atomic-type =
  | ref
  | nullable-ref
  | alt-nullable-ref
  | json-type
  | nullable-json-type
```

```
json-type = "string"  | "int"  | "real"  | "bool"

nullable-json-type =
  | "string?" | "int?" | "real?" | "bool?"

// A type name with suffix '?', e.g. "mainsignal?"
nullable-ref = '"' chars '?' '"'

// Handling the unusual case that a type name
// ends with the '?'-character.
alt-nullable-ref = {"type" : ref, "nullable" : bool}
```

4.2 Description

The package data format is used to describe a class of possible railyard graphs. A specific railyard graph fulfilling the constraints in the package data format is described in Section 5. Below follows informal descriptions of the top level items in the format.

4.2.1 package-data

The root element of the format contains all allowed railyard types in the package. It contains the following members:

`"format"`
: This is a case-insensitive string containing the name and version of the package data format.

`"package"`
: The name of the package. A package name will be used as a reference identifier in all the other LCF sub-formats.

`"imports"`
: If this member is present it shall contain references to other package data files. The imports may not be directly or indirectly cyclic, neither is it allowed that the type names in the imported modules collide with the names in the current module.

`"node-types"`
: All node types in the package, see the description of `node-type` below.

`"object-types"`
: All object types in the package, see the description of `object-type` below.

`"user-types"`
: All user types in the package, see the description of `user-type` below.

`"union-types"`
: All union types in the package, see the description of `union-type` below.

`"table-types"`
: All table types in the package, see the description of the `table-type` below.

4.2.2 node-type

A node in a railyard graph is specified by an identifier, a number of connectors of the node and a traversal relation consisting of pairs of connectors. The traversal relation specifies the allowed passages through the node. Connectors are represented by integers. A signal, for example, could be modeled by an object which resides in a node with two connection points (0 and 1), having the traversal relation $\{ (0, 1), (1, 0) \}$ allowing passage in both directions through the node.

A node type names a class of nodes with the same number of connectors and same traversal relation. A node type has an identifier `"id"` and the following extra members:

`"degree"`
: The number of connectors of this node type. If the degree is N the connectors are represented by the numbers $0..(N-1)$. The connectors are distributed clockwise, in order, around the the node.

`"traversal"`
: The traversal relation of this node type. This is a binary relation on the set of connectors $\{ 0, ..., N-1 \}$ where (i, j) belongs to the relation whenever the node can be traversed from connector i to connector j.

4.2.3 object-type

An object in a railyard graph is a data object with a set of attributes. The attributes contain named data items. The object is typically placed in a node in the graph. An object not placed in any node is called *external*, otherwise *internal*.

An object type names a class of objects with constraints on the node associations and the attributes. An object type has an identifier `id` and the following extra members:

`"allowed-node-types"`
: A list of node types constraining which kind of nodes this object may be placed in. The list has to be empty for an external object.

`"required-attrs"`
: A list of *attribute names* which specifies a minimum set of attributes which must be included for instances of this object type. An attribute for an instantiated object is a pair consisting of a attribute name and a string value.

4.2.4 user-type

The *base types* of a package data format consists of:

- The predefined `"Path"` type, the set of all paths.
- The predefined `"Area"` type, the set of all areas.
- The object types.

User types further partition base types. A user type is the smallest building block for sets of items supported by LCF. A user type has an identifier `"id"` and the following extra members:

`"base-type"`
: The name of the base type which this user type is included in. As described above this name shall be `"Path"`, `"Area"`, or the identifier for a defined object type.

`"def"`
: This member, a text string, provides a precise description of the intended meaning of the type. Note that the optional `descr` member (see Section 3.3.2) which is available for all LCF objects shall not be used for this purpose.

4.2.5 union-type

A union type is a type consisting of the union of base types or defined user types. It has an identifier `"id"` and the following extra member:

`"user-base-types"`
: A list of references to defined user types or base types.

4.2.6 table-type

A table type shall have exactly one instance in each project. One can therefore view the table type as a declaration of a specific table. The table type specifies the name and type of each column of the table. It has an identifier `"id"` and the following extra members:

`"signature"`
: This member specifies the column names and types for the rows in the instantiated table as described above. We say that a type is *nullable* if it includes the JSON `null` value. The type for a column value can be a predefined, possibly nullable, JSON type or a reference to a union type, a user type, or a base type. A type reference, may also be nullable, this is represented either by appending `'?'` to the name of a user type, or using the explicit form: `{"type":ref,"nullable":bool}`. A column can also be typed to contain a list of single values. Such lists may not however not include `null`.

`"def"`
: This member, a text string, provides a precise description of the intended meaning of the table. The description typically uses the column names from the signature to describe the table. Note that the optional `descr` member (see Section 3.3.2) which is available for all LCF objects shall not be used for this purpose.

`"primary"`
: If this Boolean member is present and true, or omitted, it means that projects using this package-data specification must provide the actual table (see Section 6) as input. A table type in which the member is present and false is called *secondary*. Such tables are instead intended to be computed, deriving their content from actual primary tables and railyard data in the project data file (see Section 5) or in the compact data file (see Section 7).

4.3 Example

The following listing contains fragments of data from an example package data file. Because a package data file may be relevant to review manually, it is formatted using RJSON extensions.

Listing 2 Package data example

```
format:  "LCF-2.0-package-data"
package: "Some Package"
descr:   "A small example"

node-types:
  - id: CrossingNode
    descr: "Standard Crossing Node Type"
    degree: 4
    traversal: [[0, 2], [1, 3], [2, 0], [3, 1]]
  - id: EndNode
    descr: "Standard End Node Type"
    degree: 1
    traversal: []

object-types:
  - id: DirectedInsideObject
    descr: "Standard Signal Object Type"
    allowed-node-types: [PassageNode]
    required-attrs: [DirectionLeg]
  - id: DirectionObject
    descr: "Standard Direction Object Type"
    allowed-node-types: []
    required-attrs: [Positive]

user-types:
  - id: NodeUserType
    base-type: GenericInsideType
    def: "Internal type"
  - id: GSWITCH
    base-type: SwitchObject
    def: "3-legged switch"

union-types:
  - id: BaseType
    user-base-types:
      - Area
      - Path
      - GenericOutsideType

table-types:
  - id: axle_counter_section
    primary: true
    signature:
      - [AXLE_COUNTER_SECTION, g_area_axle_counter]
      - [TCI, bool]
      - [HasTurnback, bool]
    def: "Input table for the static attributes of
          axle counter section."
```

5 Project Data format

The project data format contains the specification of a specific railyard installation. It describes a railyard graph with nodes, edges, node objects, paths, and areas. The contained items must be instances of types declared in a package data file. The data for a railyard is written to a single file per project.

5.1 Format

The following grammar specifies the project data format.

Listing 3 Project data format

```
project-data =
{ "format"  : "LCF-2.0-project-data",
  "package" : ref,
  "project" : id,
  "nodes"   : [node*],
  "edges"   : [edge*],
  "objects" : [object*],
  "paths"   : [path*],
  "areas"   : [area*] }

node =
{ "id"        : id,
  "node-type" : ref }

edge =
{ "id"   : id,
  "edge" : [[ref,int], [ref,int]] }

object =
{ "id"        : id,
  "user-type" : ref,
  "attrs"     : attrs,
  "node"      : (ref|null) }

path =
{ "id"        : id,
  "user-type" : ref,
  "attrs"     : attrs,
  "start"     : ref,
  "edges"     : [ref+] }

area =
{ "id"        : id,
  "user-type" : ref,
  "attrs"     : attrs,
  "nodes"     : [ref*],
  "edges"     : [ref*] }

attrs = {(nstring : string)*}
```

5.2 Description

Below follows informal descriptions of the top level items in the format.

5.2.1 project-data

The root element contains all railyard objects in the project. It contains the following members:

`"format"`
: This is a case-insensitive string containing the name and version of the project data format.

`"package"`
: This member is a reference to the package data, which the objects in the railyard shall instantiate.

`"project"`
: This member gives the name of this railyard specification.

`"nodes"`
: The list of nodes in the railyard, see description of `node`.

`"edges"`
: The list of edges in the railyard, see description of `edge`.

`"objects"`
: The list of railyard objects, see description of `object`.

`"paths"`
: The list of railyard paths, see description of `path`.

`"areas"`
: The list of railyard areas, see description of `area`.

5.2.2 node

A railyard node has an identifier `"id"`. The member `"node-type"` is a reference to a node type defined in the package data format to which this project belongs.

5.2.3 edge

A railyard edge has an identifier `"id"`. The `"edge"` member contains a pair $[[n_1, i_1], [n_2, i_2]]$ where each element identifies a node and a connector on the node. The edge is directed from the first node n_1 at connector i_1, to the second node n_2 at connector i_2.

5.2.4 object

A railyard object has an identifier `"id"` and a list of attributes `"attrs"`. The member `"user-type"` is a reference to a user-type defined in the package data format to which this project belongs. The member `"node"` is a reference to the node in the railyard where the object is placed. If the value is `null`, the object is external, i.e. does not have a specific location in the railyard. The object must be in agreement with its object type, which means it must possess the required attributes and only be placed in a node having an allowed node type.

5.2.5 path

A railyard path has an identifier `"id"` and a list of attributes `"attrs"`. The member `"user-type"` is a reference to a user-type defined in the package data format to which this project belongs. The members `"start"` and `"edges"` specifies the path by containing references to the start node and a list of references for the edges contained in the path.

5.2.6 area

A railyard area has an identifier `"id"` and a list of attributes `"attrs"`. The member `"user-type"` is a reference to a user-type defined in the package data format to which this project belongs. A railyard area specifies a sub-graph in a railyard graph. The member `"nodes"` contains all nodes in the area. The member `"edges"` specifies a set of edges included in the area. The two endpoint nodes of an edge in the area must be in the `"nodes"` member.

5.3 Example

The following listing contains fragments of data from an example project data file. Because the project data format is intended as a serialization format to be read by applications only, it is formatted using strict JSON syntax.

Listing 4 Project data example

```
{ "format":  "LCF-2.0-project-data",
  "package": "Some Package",
  "project": "Some Project",
  "descr":   "A small example",
  "nodes": [
    { "id" : "node10",
      "node-type" : "PassageNode" },
    { "id" : "node10-12*14-16",
      "node-type" : "CrossingNode" },
    { "id" : "node97",
      "node-type" : "PassageNode" }
  ],
  "edges": [
    { "id" : "node10-12*14-16:1-node370:1",
      "edge" : [["node10-12*14-16", 1], ["node370", 1]] },
    { "id" : "node10-12*14-16:2-node372:2",
      "edge" : [["node10-12*14-16", 2], ["node372", 2]] },
    { "id" : "node97:1-node201:0",
      "edge" : [["node97", 1], ["node201", 0]] }
  ],
  "objects": [
    { "id" : "1",
      "user-type" : "g_switch",
      "node" : "node257",
      "attrs" : { "BentLeg" : "2" } },

    { "id" : "__d__:node81",
      "user-type" : "g_track_circuit_delimiter",
      "node" : "node81",
      "attrs" : {} }
  ],
  "paths": [
    { "id" : "D1-D17:D",
      "user-type" : "g_shunting_route",
      "attrs" : {},
      "start" : "node267",
      "edges" : ["node267:1-node260:2", "node260:0-node259:2"] }
  ],
  "areas": [
    { "id" : "1-7DG",
      "user-type" : "g_track_circuit",
      "attrs" : {},
      "nodes" : ["node256", "node257"],
      "edges" : ["node256:0-node279:0", "node257:0-node256:1"] }
  ]
}
```

6 Project Table Format

Table instances for a project are written in a separate JSON format (see Section 5). Typically, there will be a separate LCF file for each table, but it is allowed to have multiple tables for a project in the same LCF file.

6.1 Format

The following grammar specifies the project table format.

Listing 5 Project table format

```
project-table =
{ "format"  : "LCF-2.0-project-table",
  "package" : ref,
  "project" : ref,
  "tables"  : [table*] }

table = { "type" : ref, "header"? : [nstring+], "rows" : [[cell+]*]  ⟷
    }

cell =
  | json-literal
  | null
  | [json-literal*]

json-literal = string | int | real | bool
```

6.2 Description

Below follows descriptions of the items in the format.

6.2.1 project-table

The root element contains all tables in the project.
It contains the following members:

`"format"`
: This is a case-insensitive string containing the name and version of the project table format.

`"package"`
: This member is used to reference the package data types which the tables in the format shall instantiate.

`"project"`
: This member is used to reference the project data (or compact data, see Section 7) to which the tables are associated. The referenced data and the tables must share the same `"package"` reference.

`"tables"`
: A list of tables, see description of `table`.

6.2.2 table

A table is a list of rows conforming to a referenced table type. For each table type in the package data file there shall be exactly one corresponding table in the set of project table files for a project. The table contains the following members:

`"type"`
: This member shall reference a table type in the corresponding package data file.

`"header"`
: The order of the columns in the table shall by default be as in the signature for the table given by the table type. Another order may be chosen by listing the columns in the optional header row. If present, the header must be a permutation of the column names in the signature.

`"rows"`
: This member includes the actual contents of the table. The rows of the table must be of correct length and type according to the signature for the table as given by the table type.

6.3 Example

The following listing contains fragments of data from an example table data file. Because a table data file may be relevant to review manually, it is formatted using RJSON extensions.

Listing 6 Project table example

```
format:  "LCF-2.0-project-table"
package: "Some Package"
project: "Some Project"
descr:   "A small example"
tables:

  - type: g_associated_direction
    rows:
      - ["1"            , Up  ]
      - ["85"           , Down]
      - ["87"           , Down]
      - ["89"           , Down]
      - ["9"            , Down]
      - ["D1"           , Down]
      - ["D1-D17:D"     , Up  ]
      - ["D1-D19:D"     , Up  ]
      - ["D1-D21:D"     , Up  ]
      - ["D1-D23:D"     , Down]
      - ["D1-D35:C"     , Down]
      - ["D1-D37:C"     , Down]
      - ["D1-D39:C"     , Up  ]
      - ["D1-D41:C"     , Down]
      - ["D1-D43:C"     , Down]

  - type: g_inhibited_by_lodk
    rows:
      - ["6005(BM)"     , "5210"]
      - ["6007(BM)"     , "5210"]
      - ["6021(AM)"     , "5209"]
      - ["6035(AM)"     , "5217"]
      - ["6038(BM)"     , "5209"]
      - ["6040(BM)"     , "5209"]
      - ["6041(AM)"     , "5209"]
      - ["6041(AM)"     , "5217"]
      - ["6041(BM)"     , "5218"]
      - ["6042(AS(NP))" , "5210"]
      - ["6046(AM)"     , "5210"]
      - ["6086(AM)"     , "5222"]
      - ["6086(BM)"     , "5217"]
      - ["6086(BM)"     , "5221"]
      - ["6088(AM)"     , "5222"]
      - ["6088(BM)"     , "5217"]
      - ["6088(BM)"     , "5221"]
```

7 Compact Data Format

LCF is designed to represent a wide range of possible railyard layouts. Reconsider the project data format as described in Section 5. The basic idea in this representation is to describe a railyard graph in the form of nodes and edges. On top of this graph various railyard objects like signals, switches, etc. are placed on the nodes of the graph. Paths in the graph are represented as a start node and a list of edges. Areas in the graph are represented as a set of nodes and edges. The project data format is also called *explicit* since it does not contain implied data, everything is explicitly given. This makes the explicit format easy to read and manipulate using software tools, but since it contains a lot of redundant data it is not well suited for manual review.

The current section introduces an alternative to the explicit format using a new representation of the railyard graph, objects, paths and areas. This new *compact* representation dispenses with the explicit graph nodes and edges and instead describes the graph directly in terms of the railyard objects. Areas and paths in the compact representation are also made more concise by only including the minimal data for uniquely identifying the intended part of the graph. As can be seen below the same data can be written in several ways in the compact representation, enabling the choice of a more readable alternative for a specific application.

Some explicit configurations can not be entirely represented using the compact format, but every consistent, compact configuration can be replaced with an explicit configuration.

The term *entity* is used to refer to railyard objects like signals and switches but also for paths and areas in the railyard graph.

7.1 Format

Below the extended project data format is listed, with the new members for compact representation. Please compare this grammar with the grammar in Section 5.

Listing 7 Compact data format

```
xproject-data =
{ "format"     : "LCF-2.0-xproject-data",
  "package"    : ref,
  "project"    : id,
  "entities"   : [entities*],
  "attributes" : [attributes*],
  "segments"   : [segment*],
  "paths"      : [paths*],
  "areas"      : [areas*] }

entities =
{ "user-type" : ref, "entities" : [entity*] }

entity = id | {"id" : id}

attributes =
{ "user-type" : ref, "attrs" : [entity-attrs]* }

entity-attrs =
{ "entity" : ref, "attrs" : attrs }

attrs = {(nstring : string)*}

segment = [segment-item, segment-item+]
```

```
segment-item =
  | generic-item
  | [object-item+]
  | single-item

generic-item =
  | {"in"? : int,  "objects" : [object+], "out"? : int}
  | index-decorated-string

// A JSON string with prefix N^ or suffix ^N or both, where N
// is a natural number, e.g. "1^OBJECT^2".
index-decorated-string = '"' (int '^')? chars (int '^)? '"'

object-item = object | directed-item

object = ref | {"object" : ref}

single-item =
  | object
  | directed-item
  | switch-item

directed-item =
  | {"dir" : dir, "object" : ref}
  | direction-decorated-string

dir = "<" | ">"

// A JSON string with <dir> as prefix, e.g. ">SIGNAL".
direction-decorated-string = '"' ('<'|'>') chars '"'

switch-item =
  | {"in"? : leg, "object" : ref, "out"? : leg}
  | switch-decorated-string

leg = "~" | "-" | "/" | "\\"

// A JSON string with <leg> as prefix, suffix or both,
// e.g. "~SWITCH/".
switch-decorated-string = '"' lchar? chars lchar? '"'

lchar = '~' | '-' | '/' | '\\'

paths = {"user-type" : ref, "paths" : [path*]}

path = {"id" :id, "path" : [ref, ref+]}

areas = {"user-type" : ref, "areas" : [area*]}

area =
  | {"id" : id, "delimiters" : [ref*], "objects" : [ref*]}
  | {"id" : id, "include" : [ref*], "exclude"? : [ref*]}
  | {"id" : id, "paths" : [[ref+]*]}
  | {"id" : id, "union" : [ref*]}
```

7.2 Description of top-level members

The top-level members for the compact data format are:

`"format"`
: This is a case-insensitive string containing the name and version of the compact data format.

`"package"`
: This member is used to reference the package data types specification which the objects in the format should instantiate.

`"project"`
: This member gives the name of this railyard specification.

`"entities"`
: This member contains declarations of all entities, i.e. objects, paths and areas in the compact format. The entities are grouped per user-type. Most objects occurring in this section will also appear in some *segment*, see below. An object which is not part of any such segment is an *external* object which is not located in the railyard but used to model abstract objects which affect the railyard model but has no physical correspondence. An entity is declared using a single identifier, or with the notation `{"id": id}`. The latter form is useful if one also want to add a description of the entity using the `descr` tag.

`"attributes"`
: Attributes are defined on entities. For improved readability, the attributes are grouped per entity user-type. An attribute specification refers to a declared entity together with its attributes.

`"segments"`
: This member is used for specifying the graph in terms of the railyard objects. Every segment is a list representing a path in the graph. The given segments shall describe the complete graph. Nodes can be repeated, but not edges. The case where multiple objects are positioned at the same implicit graph node in a segment is simply handled in the notation by enumerating them inside a JSON array. See Section 7.4 for further information of the components in a segment.

`"paths"`
: This member contains the definitions of the paths in the compact format. A path is represented by a list of objects of minimal length two, where the first element is the start object and the last element is the end object. The object list must uniquely describe a path in the graph.

`"areas"`
: This member contains the definitions of the areas in the compact format, as given in Section 7.3 below.

7.3 Description of areas

There are four different ways to specify a compact area:

Delimited areas
: An item `{"id":id, "delimiters":[ref*], "objects":[ref*]}` is used for specifying an area using delimiters. The member `"delimiters"` is a list of user types. An object having any of these user types is called a *delimiter*. The member `"objects"` is a list of non-external objects.

  The specified area consists of the sub-graph containing all objects reachable from the given objects without *strictly* passing a delimiter, i.e. without going beyond the delimiter. Thus, delimiter objects can also be reachable, provided that no (other) delimiter objects are passed earlier on the way.

  Note that reachable here shall be interpreted without regards to the direction of the edges in the original graph, and without regard to the traversal relation.

Hull areas
: An item `{"id":id, "include":[ref*], "exclude"?:[ref*]}` denotes the union

of all possible paths that start and end in objects listed in the `include` member but do not pass any object listed in the `exclude` member.

Path sets
: An item `{"id":id, "paths":[[ref+]*]}` denotes the union of a set of explicitly given paths and singleton nodes. A path is represented as a list of objects in the same way as in the top member `"paths"` (implying a list-length of at least two). The singleton nodes are represented by lists of length one.

Unions
: Finally, an item `{"id":id, "union":[ref*]}` is used for specifying an area as the union of already given areas.

7.4 Description of segments

A segment describes a path in the railyard graph. This means that it has to respect the traversal relation. Two consecutive entities in a segment represents an edge in the graph, with the same direction as in the segment, read from left to right. The complete set of segments describes the entire graph. A single segment is a list of *segment items*, corresponding to nodes in the explicit format. The compact format allows several alternatives for writing a segment item with the intention to optimize the readability for a specific project.

Names referencing objects in a segment item may in some places contain *decorations*. These are extra prefixes or suffixes for giving concise information for signals and switches. For example `"~SWITCH17/"` can be used to denote a segment item consisting of a switch which is entered from the common leg (the ~ prefix) and exited along the leg bent to the left (the / suffix). Decorations are further described in Section 7.4.4 and Section 7.4.5.

7.4.1 Generic segment item

In most cases a segment item is a single reference to a declared railyard object. The generic format for a segment item is however:

```
{"in"?:int, "objects":[object+], "out"?:int}
```

The items are references to declared objects. When there are more than one item, it means that the referred objects are placed together in the underlying node in the graph. If the node type of an object is known or can be inferred one may add indexes for the node's connector points. A start segment item can only have an `out` index and an end segment item can only have an `in` index. The object names in a generic segment item may not contain any decorations. When there is a single object in the object list a generic item may instead be written on the shorter form: `"IN^NAME^OUT"`, which is equivalent to the explicit form `{"in":IN, "objects":[NAME], "out":OUT}`.

7.4.2 Co-located segment item

The second form for segment items is written

```
[object-item+]
```

which allows for one, or possibly more co-located railyard objects. Compared to the generic notation, this form does not allow explicit node indices. The implicit

node of the objects shall have at most two connectors, which excludes switches or other sorts of crossings. As can be seen from the format, an `object-item` is either simply a name of an object or a decorated name typically referring to signals, see Section 7.4.4 below.

7.4.3 Single segment item

The third and last form for segment items can only be used when there is a single object occupying the implicit node. In this case the object can be of any type and all decorations are allowed. See subsequent subsections.

7.4.4 Signal decorations

A reference to a signal in a segment item may be prefixed by decorations < or >. The type of such a prefixed object shall be `DirectedInsideObject`, and its associated node type shall be `PassageNode`. Both these types must be defined in the package data file with the following definitions:

```
directed-inside-object =
{ "id" : "DirectedInsideObject",
  "allowed-node-types" : ["PassageNode"],
  "required-attrs" : ["DirectionLeg"]}

passage-node =
{ "id" : "PassageNode",
  "degree" : 2,
  "traversal" : [[0,1],[1,0]]}
```

The prefixes "<" and ">" are used with the following informal meaning for segment elements:

- ">" means that the directed object is directed forward (from left to right)
- "<" means that the directed object is directed backward (from right to left)

The specification can also be written as a JSON object with the prefix separated from the name, e.g. `{"dir":">", "object":"SIG"}`.

The required attribute `DirectionLeg` for a `DirectedInsideObject` shall be a value in $\{0, 1\}$. It specifies the direction of the object, by assigning the value of the node index that gives the direction. The value for `DirectionLeg` is inferred from the decorations.

7.4.5 Switch decorations

A reference to a switch in a segment item may be prefixed and/or suffixed by strings `-`, `~`, `/` or `\`. The type of such a decorated object shall be `SwitchObject` and its associated node type shall be `SwitchNode`. Both these types must be defined in the package data file with the following definitions:

```
switch-object =
{ "id" : "SwitchObject",
  "allowed-node-types" : ["SwitchNode"],
  "required-attrs" : ["BentLeg"] }

switch-node =
{ "id" : "SwitchNode",
  "degree" : 3,
```

```
  "traversal" : [[0,1],[1,0],[0,2],[2,0]] }
```

The extra symbols have the following informal meaning:

- A hyphen - refers to the straight leg in the switch.
- A tilde ~ refers to the common leg in the switch.
- A slash / refers to the bent leg in a left-handed switch.
- A backslash \ refers to the bent leg in a right-handed switch.

For example, given a switch reference VX the segment

```
["A", "~VX/", "B"]
```

means that we start from object A, entering the common leg of the left-handed switch VX and leave the switch via it's bent leg to reach object B.

A switch specification can also be written as a JSON object with codes separated from the name, e.g. {"in":"~", "object":"VX", "out":"/"}.

The required attribute BentLeg for a SwitchObject shall be a value in {"1", "2"}. When interpreting a switch specification, a left handed switch gets attribute value "1" and a right handed switch get value "2".

7.5 Example

The following listing contains fragments of data from an example compact data file. Because a compact data file may be relevant to review manually, it is formatted using RJSON extensions.

Listing 8 Compact data example

```
format:  "LCF-2.0-xproject-data"
package: "Some Package"
project: "Some Project"
descr:   "A small example"

entities:
  - user-type:"GSWITCH"
    entities:
      - "2403bV"
      - "2405aV"
      - "GSWITCH-19"
  - user-type: "g_direction"
    entities:
      - "negdir"
      - "posdir"

attributes:
  - user-type: "g_direction"
    attrs:
      - entity: "negdir"
        attrs: {Positive: "False"}

segments:
  - ["N_1", ">2403", "<2559", "N_3"]
  - ["N_9", "<2406", "N_10"]
  - ['-SWP_STK_22~', "N_234", "g_delim_track_circuit-2"]

paths:
  - user-type: "g_fl_path_si"
    paths:
      - id: "2403bV.$g_end_safe-7-0"
        path: ["br_2403bV__0-1", "N_15"]

areas:
  - user-type: "g_area_track_circuit"
    areas:
      - id: "CDV_2401"
        paths: [["2403", "2559"]]
```

PART II
Formal requirements

8 Common Notations

We use standard concepts from predicate logic and naïve set theory for defining common data structures and operators.

Functions For sets A and B, let $f : A \to B$ denote a function f with domain A and co-domain B. For such an f and $C \subseteq A$, let

- $\mathrm{dom}(f) = A$ be the domain of f
- $f[C] = \{ f(x) \mid x \in C \}$, the *image of C under f*
- $f|_C = \{ (x, f(x)) \mid x \in C \}$, the *restriction of f to C*
- $\mathrm{ran}(f) = f[A]$, the *range* of f.

Given functions $f : A \to B$ and $g : B \to C$ the *composition* $g \circ f : A \to C$ is defined by $(g \circ f)(x) = g(f(x))$.

Given any term t, the notation $t\langle x_0, x_1, \ldots, x_n \rangle$ asserts that the set of free variables in t is a subset of $\{x_0, x_1, \ldots, x_n\}$. For terms t and v and variable x the notation $t\langle v/x \rangle$ denotes the result of substituting all free occurrences of x in t by v.[1]

For a term $t\langle x \rangle$ and a set A, the term $\lambda(x : A)\, t$ denotes a function f with domain A defined as $f(x) = t$. The shorter notation $\lambda x\, t$ is used when the intended domain is obvious from the context.

Choice operator For a predicate $p\langle x \rangle$, $\varepsilon x\, p$ denotes a value v such that $p\langle v/x \rangle$, if such a v exists, otherwise any value. Formally, for all p, $\exists x p \to p(\varepsilon x p)$. The value $\varepsilon x\, p$ is *uniquely defined* if $\exists! x p$. The chosen value is the same for extensional equal predicates, i.e. for all predicates p and q, $\forall x (p(x) \leftrightarrow q(x)) \to \varepsilon x p = \varepsilon x q$. Let $\bot$ denote the chosen value for an empty predicate, i.e. $\bot = \varepsilon x\, \mathit{false}$.

Quotient sets Let X be any set and let R be an equivalence relation on X. For an element $e \in X$, the notation $[e]_R = \{ x \in X \mid x\, R\, e \}$ denotes the *equivalence class* of e under R. The set of all such classes, $X/R = \{ [e]_R \mid e \in X \}$, is the *quotient set* of X induced by R.

List operations For $i, j \in \mathbb{N}$, let $i\,..\,j = \{ n \mid i \le n \le j \}$ and let $\overline{n} = 0\,..\,n-1$. For any set X a function $l : \overline{n} \to X$, is called a *list of X*, where $|l| = n$ is its length. Let X^* be the set of all lists of X. The range of a list is also called its *elements*. Elements in a list l are by convention denoted by indexing, l_i, instead of $l(i)$. For nested lists, $l_{i,j}$ abbreviates $l(i)(j)$. A list l may also be denoted by the notation

$$[l_0, l_1, \ldots, l_{|l|-1}]$$

enumerating its elements in order. For a list l let

- $\mathrm{uniq}(l)$ be true iff l is injective
- $\mathrm{set}(l)$ be a synonym for $\mathrm{ran}(l)$
- x in l mean $x \in \mathrm{ran}(l)$.

[1] It may be necessary to first rename bound variables in t to avoid unintended captures of free variables in a substitution $t\langle v/x \rangle$.

For a finite set X, let list(X) be $\varepsilon l\,(\mathrm{ran}(l) = X \wedge |l| = |X|)$.

A list l *reorder* a list m iff there is a bijection $\sigma\colon \mathrm{dom}(l) \to \mathrm{dom}(l)$ such that $m = l \circ \sigma$. For any function f with a finite domain $D \subseteq \mathbb{N}$ we define squash(f) as the list $f \circ \sigma$ where σ is the unique order preserving map from $\overline{|D|}$ to D. Given a list l and $J \subseteq \mathrm{dom}(l)$, let sub$(l, J)$ be the list squash$(l|_J)$. The relation $l \sqsubseteq m$ of lists is true iff there exists an J such that $l = \mathrm{sub}(m, J)$.

For a list $l = [x_0, x_1, \ldots, x_n]$ of X, a function $f\colon Y \times X \to Y$, and $e \in Y$, fold(f, e, l) is a value in Y defined as

$$f(\ldots f(f(e, x_0), x_1), \ldots), x_n)$$

Given lists l and m their *concatenation* $l + m$ is the list

$$l \cup \{(i + |l|, m_i) \mid i \in \mathrm{dom}(m)\}.$$

Let $x :: l = [x] + l$ and for a list l of lists, let conc(l) = fold$(+, [], l)$.

Let l be a list of X and $p : X \to \mathrm{bool}$, then filter(p, l) is the list sub$(l, \{\, i \in \mathrm{dom}(l) \mid p(l_i)\,\})$. For a list l, term $f\langle x\rangle$, and predicate $p\langle x\rangle$, the notation

$$[\, f \mid x \text{ in } l \text{ if } p\,] = (\lambda x\, f) \circ (\mathrm{filter}(\lambda x\, p, l))$$

is a *list comprehension* where the if-clause is optional with *if true* as default.

Strings and dictionaries A *character* is a Unicode value, an integer in range `U+0000 .. U+D7FF` or `U+E000 .. U+10FFFF`. Characters are generally denoted by their Unicode value, but printable ASCII-letters may be written enclosed in single quotes.

A *string* is a list of characters. A string of printable ASCII-letters can be written as a word enclosed in double quotes or simply as an unquoted word when this is transparent.

Given a set S of strings and any set X, a function $d : S \to X$ is called a *dictionary of* X. Let X^D denote the set of all dictionaries of X.

9 Extra JSON Notations

The JSON syntax is used for LCF. See the RFC [1] for the current version of the JSON specification. This section contains notations for specifying and accessing JSON input.

9.1 Semantic restrictions

An LCF parser shall adhere to the syntax for JSON as described in the JSON standard [1]. In addition, the parser shall have a stricter interpretation of semantics than the standard in the following cases.

Encodings The JSON standard recommends that source files are encoded in UTF-8, which is mandatory for LCF sources. The standard leaves it to the implementation how to handle encoding errors, however an LCF parser shall reject any malformed encoding. Note also that LCF sources shall not start with a Byte Order Mark (BOM).

Object members The JSON standard recommends, but do not enforce, that the members in a JSON object are interpreted as unordered and that the member names in an object are unique. An LCF parser shall always disregard the order of the members when interpreting the contents of the file and also report duplicated member names in an object as an error.

Numbers The JSON standard does not distinguish integers and real numbers. An LCF parser shall however interpret a JSON number with a fractional part or exponent as a real number and otherwise as an integer. Neither does the JSON standard mandate a specific interpretation of real numbers for computerized processing, e.g. as 64-bit floating point numbers. It is left to an application reading JSON data to specify such an interpretation and give further constraints regarding size and precision. This approach is shared by LCF, delegating to readers of LCF to specify the final interpretation.

9.2 JSON AST

Let int, real, bool and string be the sets of integers, real numbers, Boolean constants and strings, respectively. A valid JSON input, fulfilling the restrictions in Section 9.1, has an obvious translation to an instance of the type *json*, defined inductively as

$$\text{json} = \text{int} \cup \text{real} \cup \text{string} \cup \text{bool} \cup \{\text{null}\} \cup \text{json}^* \cup \text{json}^D$$

Let a *JSON file* mean both the actual file, its contents, and its interpretation as a json value. We rely on context for distinguishing these different meanings.

The equivalence relation $u \equiv^u v$ is true for json values u and v iff they are equal disregarding the ordering of contained lists in u and v, formally

$$\begin{aligned}
&u \equiv^u v \Leftrightarrow\ u = v \vee \\
&(u \in \text{json}^* \wedge v \in \text{json}^* \wedge |u| = |v|\ \wedge \exists z\,(z \text{ reorder } v\ \wedge \forall i\colon \text{dom}(u)\,(z_i \equiv^u u_i)) \vee \\
&(u \in \text{json}^D \wedge v \in \text{json}^D \wedge\ \text{dom}(u) = \text{dom}(v) \wedge \forall x\colon \text{dom}(u)\,(u(x) \equiv^u v(x)))
\end{aligned}$$

9.3 Path expressions

A (JSON) *path expression* has the form $vq_1q_2\ldots q_n$ where $v \in$ json and each q_i is a *qualifier* with the following informal meaning:

[∗] Selects all elements in a dictionary or list. The alternative form .∗ is also allowed.

$[i]$	Selects the element with index i in a list. A negativ index selects an element from the end of a list.
$[i{:}j]$	Selects elements from l_i to l_{j-1} in a list l. The second index may be omitted, in which case all remaining elements from l_i in l are selected.
$[?(c)]$	Selects all values satisfying a condition c. The symbol @ is used in the condition for referencing the currently evaluated value.
.key	Selects the specified member value in a dictionary. Member keys in qualifiers may be unquoted if not containing white space.
..key	Recursively searches for the specified key and selects all member values with this key.
p, q	Selects the union of p and q, this may also be written as $[p', q']$ where p' and q' are the results of removing square brackets in p and q.

Some preliminary definitions are needed to give a formal meaning for path expressions. The function jran returns all elements in a list or dictionary

$$\text{jran}(v) = \begin{cases} v & \text{if } v \in \text{json}^* \\ \text{list}(\text{ran}(v)) & \text{if } v \in \text{json}^D \\ [\,] & \text{otherwise} \end{cases}$$

The function jsub returns the list of all json values contained in a json value and is inductively defined by

$$\text{jsub}(v) = [v] + \text{conc}(\text{jsub} \circ (\text{jran}(v)))$$

Let $i, j \in \mathbb{N}$ below. For a $L \in \text{json}^*$ and a qualifier q, the function $\varphi(L, q)$ returns a new json list:

$$\begin{aligned}
\varphi(L, [*]) &= \text{conc}(\text{jran} \circ L) \\
\varphi(L, [i]) &= [\, l_i \mid l \text{ in } L \text{ if } l \in \text{json}^* \wedge i \in \text{dom}(l) \,] \\
\varphi(L, [-i]) &= [\, l_{|l|-i} \mid l \text{ in } L \text{ if } l \in \text{json}^* \wedge |l| - i \in \text{dom}(l) \,] \\
\varphi(L, [i{:}j]) &= \text{conc}([\, \text{sub}(l, i\,..\,j-1 \cap \text{dom}(l)) \mid l \text{ in } L \text{ if } l \in \text{json}^* \,]) \\
\varphi(L, [i{:}]) &= \text{conc}([\, \text{sub}(l, i\,..\,|l|-1) \mid l \text{ in } L \text{ if } l \in \text{json}^* \,]) \\
\varphi(L, [?(c)]) &= [\, v \mid v \text{ in } \text{conc}(\text{jran} \circ L) \text{ if } c\langle v/@\rangle \,] \\
\varphi(L, .key) &= [\, d(key) \mid d \text{ in } L \text{ if } d \in \text{json}^D \wedge key \in \text{dom}(d) \,] \\
\varphi(L, ..key) &= [\, d(key) \mid d \text{ in } \text{jsub}(L) \text{ if } d \in \text{json}^D \wedge key \in \text{dom}(d) \,] \\
\varphi(L, (p, q)) &= \varphi(L, p) + \varphi(L, q)
\end{aligned}$$

Finally, the formal meaning of a path expression is given as

$$\text{eval}(vq_1q_2 \ldots q_n) = \text{fold}(\varphi, [v], [q_1, q_2, \ldots, q_n])$$

The condition $c\langle@\rangle$ in a qualifier $[?(c)]$ (or $[c]$) is not formalized in this presentation, it shall be a Boolean term containing standard or defined operators.

Note that in the rest of the document, the *eval* function is implicit, when using path expressions we always mean the evaluated result.

Path expressions are typically used in the context of a schema. The symbol $ is used in path expressions for denoting an arbitrary root instance of the schema under consideration.

Given the definition of *eval*, a path expression always denotes a list l of json values. An empty result indicates a non-matching expression. When the result is guaranteed to be a singleton, the operator

$$l' = \varepsilon x\,(|l| = 1 \wedge l_0 = x)$$

is used to denote the single element.

10 Requirements for Package Data Format

Let a *package data file* denote a file satisfying the grammar in Section 4.1. This section contains additional semantic requirements for a package data file $. In later sections, when referring to a construction defined in another LCF file F, the construction is indexed with F.
The following are definitions for sets of built-in type identifiers of LCF.

JsonTypes = {"string", "int", "real", "bool"}
JsonNullTypes = {"string?", "int?", "real?", "bool?"}
GeoTypes = {"Path", "Area"}
BuiltinTypes = JsonTypes ∪ JsonNullTypes ∪ GeoTypes

The following are definitions for sets of type identifiers contained in $.

NodeTypes = set($.node-types..id)
ObjectTypes = set($.object-types..id)
UserTypes = set($.user-types..id)
UnionTypes = set($.union-types..id)
TableTypes = set($.table-types..id)
CoreTypes = set($..id)
BaseTypes = GeoTypes ∪ ObjectTypes
UserBaseTypes = BaseTypes ∪ UserTypes
EntityTypes = BaseTypes ∪ UserTypes ∪ UnionTypes
ListColumnTypes = EntityTypes ∪ JsonTypes
ColumnTypes = ListColumnTypes ∪ { s + "?" | $s \in$ ListColumnTypes }

REQUIREMENT 1 (Type identifiers)
Let L = $..id, the list of identifiers in $, it is required that:

1. The list L is unique.
2. No identifier is the name of a built-in type, i.e. set(L) ∩ BuiltinTypes = ∅.
3. No identifier ends with '?', i.e. ∀s in L ($s_{|s|-1} \neq$ '?').

REQUIREMENT 2 (Traversal relation)
For each x in $.node-types[*], let $T = x$.traversal′, it is required that:

1. Indices in T respects the degree of x, i.e. ∀i in T[*][*] ($0 \leq i < x$.degree′).
2. T is irreflexive, i.e. ∀e in T ($e_0 \neq e_1$).
3. T is symmetric, i.e. ∀e in T ($[e_1, e_0]$ in T).

REQUIREMENT 3 (Column names)
For each x in $.table-types[*].signature, the list $x[*][0]$ of column names is unique.

Let a *valid file identifier* be a string that uniquely references an existing file. For a valid file identifier p, let $\mathcal{F}(p)$ be the file referenced by p. A package data file u is *reachable* from a package data file v iff there is an s in v.imports[*] such that either $u = \mathcal{F}(s)$ or u is reachable from $\mathcal{F}(s)$. Note that imports are optional, if not used the following requirements for imports are vacuously satisfied.

REQUIREMENT 4 (Import paths)
An application of LCF making use of import directives must specify the notion of a valid file identifier and the function $\mathcal{F}$ from such identifiers to files on the implemented platform.

REQUIREMENT 5 (Import validity)
Each p in $.imports[*] shall be a valid file identifier and $\mathcal{F}(p)$ shall be a package data file satisfying all requirements in this section.

The import graph may not contain cycles, but note that the same file may be reachable in more than one way from $.

REQUIREMENT 6 (Imports non-cyclic)
The file $ may not be reachable from itself.

REQUIREMENT 7 (Import collisions)
Let T be a package data file reachable from $, then the type identifiers in the files must be disjoint, formally

$$\text{CoreTypes}_{\$} \cap \text{CoreTypes}_{T} = \varnothing$$

Let Q be the set consisting of $ and all package data files reachable from $.

REQUIREMENT 8 (Referenced types)
The references to types in $ shall belong to the following categories of names.

1. set($.object-types..allowed-node-types[*]) $\subseteq \bigcup_{t \in Q} \text{NodeTypes}_t$
2. set($.user-types..base-type) $\subseteq \bigcup_{t \in Q} \text{BaseTypes}_t$
3. set($.union-types..user-base-types[*]) $\subseteq \bigcup_{t \in Q} \text{UserBaseTypes}_t$
4. set($.table-types..signature[*][1][0]) $\subseteq \bigcup_{t \in Q} \text{ListColumnTypes}_t$
5. set($.table-types..signature..type) $\subseteq \bigcup_{t \in Q} \text{ColumnTypes}_t$
6. set($.table-types..signature[@(1) ∈ string][1]) $\subseteq \bigcup_{t \in Q} \text{ColumnTypes}_t$

Note that Q = {$} for an application of LCF that disallows imports.

10.1 Auxiliary definitions

All the following LCF formats described in this document are dependent on a dedicated package data file, describing the types for used entities. This section contains extra definitions needed for the formal requirements of subsequent formats. First are definitions for sets of type identifiers contained in $ or in any of its imported files.

$$\begin{aligned}
\text{AllNodeTypes} &= \bigcup_{t \in Q} \text{NodeTypes}_t \\
\text{AllObjectTypes} &= \bigcup_{t \in Q} \text{ObjectTypes}_t \\
\text{AllUserTypes} &= \bigcup_{t \in Q} \text{UserTypes}_t \\
\text{AllTableTypes} &= \bigcup_{t \in Q} \text{TableTypes}_t \\
\text{AllCoreTypes} &= \bigcup_{t \in Q} \text{CoreTypes}_t \\
\text{AllBaseTypes} &= \bigcup_{t \in Q} \text{BaseTypes}_t \\
\text{AllUnionTypes} &= \bigcup_{t \in Q} \text{UnionTypes}_t \\
\text{AllEntityTypes} &= \bigcup_{t \in Q} \text{EntityTypes}_t
\end{aligned}$$

Given a type identifier s in AllCoreTypes the requirements in the previous sections guarantees that there is a unique associated json object in \$ or in an imported package data file. Let $\mathrm{id}(s)$ denote this object, formally

$$\mathrm{id}(s) = \varepsilon x\,(\exists t : \mathrm{Q}\,(x \text{ in } \mathrm{jsub}(t) \wedge x.\mathrm{id} = [s]))$$

Given a name s in AllEntityTypes, $\mathrm{tc}(s)$ is the set of user types included in s:

$$\mathrm{tc}(s) = \begin{cases} \bigcup_{t \in \mathrm{Q}} \mathrm{set}(t.\text{user-types}[@(\text{base-type}) = s].\mathrm{id}) & s \in \text{AllBaseTypes} \\ \bigcup \{\, \mathrm{tc}(j) \mid j \text{ in } \mathrm{id}(s).\text{user-base-types}[*] \,\} & s \in \text{AllUnionTypes} \\ \{\, s \,\} & s \in \text{AllUserTypes} \end{cases}$$

11 Requirements for Project Data Format

Let a *project data file* denote a file satisfying the grammar in Section 5.1. This section contains additional semantic requirements for a project data file $. The requirements are dependent on a package data file, providing the types for the entities in $.

REQUIREMENT 9 (Package data)
There must exist a package data file for $, that is a valid package data file T such that T.package = $.package.

In subsequent requirements let T be the package data file for $. The definitions in Section 10.1, applied to T, are used in the requirements below.

The following are definitions for sets of identifiers in $.

Nodes = set($.nodes..id)
Edges = set($.edges..id)
Objects = set($.objects..id)
Paths = set($.paths..id)
Areas = set($.areas..id)
Ident = set($..id)
Entities = Objects ∪ Paths ∪ Areas

REQUIREMENT 10 (Item identifiers)
The list of identifiers is unique, i.e. uniq($..id).

REQUIREMENT 11 (Referenced names)
The references to items and types in $ shall satisfy the following restrictions:

1. set($.nodes..node-type) ⊆ AllNodeTypes_T
2. set($.objects..user-type) ⊆ AllUserTypes_T
3. set($.objects[@(node) ≠ null].node) ⊆ Nodes
4. set($.edges..edge[*][0]) ⊆ Nodes
5. set($.paths..user-type) ⊆ AllUserTypes_T
6. set($.paths..start) ⊆ Nodes
7. set($.paths..edges[*]) ⊆ Edges
8. set($.areas..user-type) ⊆ AllUserTypes_T
9. set($.areas..nodes[*]) ⊆ Nodes
10. set($.areas..edges[*]) ⊆ Edges

For an identifier s in $, let pid($s$) be the associated json object in $, formally

$$\text{pid}(s : \text{Ident}) = \varepsilon x\ (x \text{ in jsub}(\$) \wedge x.\text{id} = [s])$$

which is uniquely defined in view of previous requirements.

The following functions are used to retrieve type information for named nodes and entities in $.

$\text{nt}(s : \text{Nodes}) = \text{id}_T(\text{pid}(s).\text{node-type}')$
$\text{ut}(s : \text{Entities}) = \text{id}_T(\text{pid}(s).\text{user-type}')$
$\text{bt}(s : \text{Entities}) = \text{id}_T(\text{ut}(s).\text{base-type}')$

REQUIREMENT 12 (Entity base types)
Entities shall have the expected base types.

1. $\{ \text{bt}(s).\text{id}' \mid s \in \text{Objects} \} \subseteq \text{AllObjectTypes}_T$
2. $\forall s : \text{Paths}\ (\text{bt}(s).\text{id}' = \text{"Path"})$
3. $\forall s : \text{Areas}\ (\text{bt}(s).\text{id}' = \text{"Area"})$

Let E = \$.edges[*].edge in the following two requirements.

REQUIREMENT 13 (Edge consistency)

1. Edge indices must be valid w.r.t. the node type,
   $\forall n \text{ in } E[*][*]\ (0 \leq n_1 < \text{nt}(n_0).\text{degree}')$
2. Edges are unique and irreflexive,
   $\text{uniq}(E) \wedge \forall e \text{ in } E\ (e_{0,0} \neq e_{1,0})$
3. A node index can not be attached to multiple edges,
   $\forall d, e \text{ in } E\ (d \neq e \Rightarrow \text{set}(d) \cap \text{set}(e) = \varnothing)$
4. There may not be multiple edges between nodes,
   $\forall d, e \text{ in } E\ (d \neq e \Rightarrow \text{set}(d[*][0]) \neq \text{set}(e[*][0]))$

Let p be a list of edges, then p is an *edge path*, $\text{epath}(p)$, iff

1. each edge in p is in the symmetric closure of E
   $\forall d \text{ in } p\ (\exists e \text{ in } E\ (d \equiv^u e))$
2. consecutive edges in p are connected and respects the traversal relation
   $\forall i, j : \text{dom}(p)\ (j = i + 1 \Rightarrow p_{i,1,0} = p_{j,0,0} \wedge [p_{i,1,1}, p_{j,0,1}] \text{ in } \text{nt}(p_{i,1,0}).\text{traversal}')$
3. p has no cycles
   $\forall i, j : \text{dom}(p)\ (j > i \Rightarrow p_{i,0,0} \notin \text{set}(p_j[*][0]))$

REQUIREMENT 14 (Path consistency)
Each path in \$ shall be a valid path, disregarding the order of each edge pair:

$\forall p \text{ in } \$.\text{paths}[*]\ (\exists q\ (|p| = |q| \wedge \text{epath}(q) \wedge\ (p.\text{start}' = q_{0,0,0} \wedge$
$\forall i \in \text{dom}(q)(\text{set}(q_i) = \text{set}((\text{pid}(p.\text{edges}[i]')).\text{edge}')))))$

REQUIREMENT 15 (Object consistency)
The node of an internal object must be an allowed node type for the object, and the object must have the the required attributes mandated by its type. This is expressed formally below.

For each object o in \$.objects[*], let

$n = o.\text{node}'$
$ot = \text{bt}(o.\text{id}')$
$a = ot.\text{allowed-node-types}'$

ot is guaranteed to be an object type in T by previous requirements. The following shall apply:

1. if $n \neq \mathit{null}$ then $\text{nt}(n).id'$ in a
2. if $n = \mathit{null}$ then a = []
3. $\text{set}(ot.\text{required-attributes}') \subseteq \text{dom}(o.\text{attrs}')$

12 Requirements for Project Table Format

Let a *table file* be a file satisfying the project table format. This section contains additional semantic requirements for a table file $.

The project table format is dependent on both a package data file and a project data file. The latter file might be in the *compact* version of the project data format which is described in Section 7.

REQUIREMENT 16 (Package and project data.)
There must exist package and project data for $. That is, a valid package data file T and a file P that is either a valid project data file or a valid compact data file satisfying:

$.package = T.package

P.package = T.package

$.project = P.project

In subsequent requirements let T be the package data file for $, and let P be the (compact) project data file for $.

REQUIREMENT 17 (Referenced table type)
The table identifier for a table must be in T, formally

$$\text{set}(\$.\text{tables..type}) \subseteq \text{AllTableTypes}_T$$

Let the predicate altnull be true for a json term of the form `alt-nullable-ref` in the grammar in Section 4.1. The types of table cells are checked using the relation $C(t, u)$ where t is the expected type of the cell value u, it is inductively defined by:

$$\begin{aligned}
&C(t : \text{json}, u : \text{json}) \Leftrightarrow \\
&\quad t = \text{"int"} \wedge u \in \text{int} \vee \\
&\quad t = \text{"real"} \wedge u \in \text{real} \vee \\
&\quad t = \text{"bool"} \wedge u \in \text{bool} \vee \\
&\quad t = \text{"string"} \wedge u \in \text{string} \vee \\
&\quad \exists p\ (t = p + \text{"?"} \wedge (u = \mathit{null} \vee C(p, u))) \vee \\
&\quad t \in \text{json}^* \wedge u \in \mathit{json}^* \wedge \forall i : \text{dom}(u)\ (C(t_0, u_i)) \vee \\
&\quad t \in \text{AllEntityTypes}_T \wedge u \in \text{Entities}_P \wedge \text{ut}_P(u) \in \text{tc}_T(t) \vee \\
&\quad \text{altnull}(t) \wedge\ \ (u = \text{null} \wedge t.\text{nullable}' = \text{true} \vee C(t.\text{type}', u))
\end{aligned}$$

REQUIREMENT 18 (Table header)
If present, the `header` member of a table must be a reordering of the column names in the signature of the table. Formally, for each t in $.tables[*] such that header $\in \text{dom}(t)$, let first $a = \text{id}_T(t.\text{type}').\text{signature}[*][0]$ and $b = t.\text{header}[*]$. It is now required that there is a permutation σ of $\text{dom}(b)$ such that $a = b \circ \sigma$.

For a table with a header we let σ be the permutation guaranteed by the previous requirement. If no header was given for the table, σ is the identity function. We can now state the main requirement for this section:

REQUIREMENT 19 (Types of table cells)
Each row in a table must respect the signature of the table, formally

$$
\begin{array}{l}
\forall t \text{ in } \$.\mathrm{tables}[*] \ (\\
\quad \text{let } r = t.\mathrm{rows}' \text{ in} \\
\quad \text{let } s = \mathrm{id}_T(t.\mathrm{type}').\mathrm{signature}[*][1] \text{ in} \\
\quad \forall i \colon \mathrm{dom}(r) \ (|s| = |r_i| \wedge \forall j \colon \mathrm{dom}(r_i) \ (C(s_j, r_{i,\sigma(j)})))
\end{array}
$$

13 Requirements for Compact Data Format

This section contains additional semantic requirements that a compact data file $ shall satisfy. The requirements are dependent on a package data file, providing the types for the entities in $.

REQUIREMENT 20 (Package data)
There must exist a package data file for $, that is a valid package data file *T* such that T.package = $.package.

In subsequent requirements let *T* be the package data file for $. The definitions in Section 10.1, applied to *T*, are used in the requirements below.[2]

In order to reduce the many forms of segment items we presuppose that $ has been rewritten to *normal form* as described in the following sub-section.

13.1 Normal form

The many alternative forms in the compact format enables a concise representation of a railyard, but complicates reasoning of the format. The *normal form* of the compact format consists of the following restrictions regarding segment items w.r.t. the grammar listing in Section 7.1.

```
segment-item =
  | generic-item
  | switch-item
  | co-located-item

generic-item  = { "in"? : int, "objects" : [ref+], "out"? : int }

switch-item   = { "in"? : leg, "object" : ref, "out"? : leg }

co-located-item = [(object | directed-item)+]

object = {"object" : ref}

directed-item = { "dir": dir, "object": ref }

entity = {"id" : id}
```

The original full format can be rewritten to the reduced normal form, with preserved meaning, using the following steps

1. The string encoded variants for `entity`, `object`, `generic-item`, `directed-item` and `switch-item` in the full format are represented with their object form given by the grammar above.
2. Each `single-item` *x* of type `object` or `directed-item` in the full format is instead represented as a singleton list `[x]` in the normal form.

13.2 Types of referenced names

REQUIREMENT 21 (Mandatory types)
The package data file *T* shall contain the object and node types:

- `SwitchObject`
- `SwitchNode`

[2]Note that some names for concepts defined in this section are overloaded w.r.t. definitions in Section 11 for the project data format.

- `DirectedInsideObject`
- `PassageNode`

as defined in Sections 7.4.4 and 7.4.5.

All items used in the compact format are listed in the entity section.

Entities = set($.entities..id)

REQUIREMENT 22 (Entity declarations)

1. Entity identifiers shall be unique.
   uniq($.entities..id)
2. Entities shall be grouped using unique user types.
   uniq($.entities..user-type)
3. Referenced entity user types shall be in the type file.
   set($.entities..user-type) $\subseteq$ AllUserTypes$_T$

Given an entity name s, the following two functions returns the identifiers for the user type and base type of s, respectively.

$$\mathrm{ut}(s : \mathrm{Entities}) = \epsilon x\ (\exists y \text{ in } \$.\mathrm{entities}[@(\text{user-type}) = x]\ (s \text{ in } y..\mathrm{id}))$$
$$\mathrm{bt}(s : \mathrm{Entities}) = \mathrm{id}_T(\mathrm{ut}(s)).\text{base-type}'$$

which are both uniquely defined in view of the previous requirement.
Next the entity declarations are partitioned according to their base types.

$$\mathrm{Paths} = \{\, s \in \mathrm{Entities} \mid \mathrm{bt}(s) = \text{"Path"} \,\}$$
$$\mathrm{Areas} = \{\, s \in \mathrm{Entities} \mid \mathrm{bt}(s) = \text{"Area"} \,\}$$
$$\mathrm{Objects} = \{\, s \in \mathrm{Entities} \mid \mathrm{bt}(s) \in \mathrm{AllObjectTypes}_T \,\}$$

For a name s of an object, the following function returns its type.

$$\mathrm{ot}(s : \mathrm{Objects}) = \mathrm{id}_T(\mathrm{bt}(s))$$

Objects are further partitioned as *exterior* and *interior*. Where the former may not occur in a railyard node.

$$\mathrm{XObjects} = \{\, s \in \mathrm{Objects} \mid \mathrm{ot}(s).\text{allowed-node-types}' = [] \,\}$$
$$\mathrm{IObjects} = \mathrm{Objects} \setminus \mathrm{XObjects}$$

Using these definitions, the next following requirements state further restrictions for referenced names in $.

REQUIREMENT 23 (Referenced names in paths)

1. Path identifiers shall be unique,
   uniq($.paths..id).
2. Identifiers for path user-types shall be unique,
   uniq($.paths..user-type).
3. Path user-types shall be consistent,
   $\forall x$ in $.paths' ($\forall y$ in x..id ($y \in \mathrm{Paths} \wedge \mathrm{ut}(y) = x.\text{user-type}'$))

4. Path elements shall be interior objects,
   set($.paths..path[*]) ⊆ IObjects

REQUIREMENT 24 (Referenced names in areas)

1. Area identifiers shall be unique,
   uniq($.areas..id)
2. Identifiers for area user-types shall be unique,
   uniq($.areas..user-type)
3. Area user-types shall be consistent,
   $\forall x$ in $.areas′ ($\forall y$ in x...id ($y \in$ Areas $\wedge$ ut(y) = x.user-type′))
4. Area delimiters are declared user-types
   set($.areas..delimiters[*]) ⊆ AllUserTypes$_T$
5. Objects in a delimited area are interior objects,
   set($.areas..objects[*]) ⊆ IObjects
6. Objects included in a hull area are interior objects,
   set($.areas..include[*]) ⊆ IObjects
7. Objects excluded in a hull area are interior objects,
   set($.areas..exclude[*]) ⊆ IObjects
8. Objects in a path-set area are interior objects,
   set($.areas..paths[*][*]) ⊆ IObjects
9. Identifiers in a union area type are areas,
   set($.areas..union[*]) ⊆ Areas
10. Identifiers in a union area type are previously declared areas,
    let t = $.$areas$[*]$.areas$[*]$ in
    $\forall i \in$ dom(t) ("union" $\in$ dom(t_i) $\Rightarrow$ set(t_i.union[*]) ⊆ set(t[0..i].id))

REQUIREMENT 25 (Referenced names in segments)

1. The objects in generic segment items are unique,
   $\forall x$ in $.segments..objects(uniq($x$))
2. The objects in segments are exactly the interior objects,
   set($.segments..object) ∪ set($.segments..objects[*]) = IObjects

REQUIREMENT 26 (Referenced names in attributes)

1. User-types in attribute section are unique,
   uniq($.attributes[*].user-type)
2. User-types in attribute section are declared,
   set($.attributes[*].user-type) ⊆ AllUserTypes$_T$
3. Entity identifiers in attribute section are unique,
   uniq($.attributes[*].attrs[*].entity)
4. Entity user-types in the attribute section are consistent with their declared type,

$\forall x$ in $.attributes[$*$] ($\forall y$ in x.attrs$[$*$]$.entity ($y \in$ Entities $\wedge$ x.user-type′ = ut(y)))

13.3 Nodes

The *segments* and *segment items* of $ are the following lists, respectively.

$$\begin{aligned}\mathrm{Sg} &= \$.\mathrm{segments}[*]\\ \mathrm{Si} &= \$.\mathrm{segments}[*][*]\end{aligned}$$

Next, predicates for classifying the segment items are defined.

$$\begin{aligned}\mathrm{switch}(v\colon \mathrm{json}) &= v \in \mathrm{json}^D \wedge \text{"object"} \in \mathrm{dom}(v)\\ \mathrm{generic}(v\colon \mathrm{json}) &= v \in \mathrm{json}^D \wedge \text{"objects"} \in \mathrm{dom}(v)\\ \mathrm{colocated}(v\colon \mathrm{json}) &= v \in \mathrm{json}^*\end{aligned}$$

Given the description of the normal form of the compact format, these predicates are exhaustive and mutually exclusive w.r.t. to the members of Si. We need also predicates for the items occurring in co-located segment items:

$$\begin{aligned}\mathrm{directed}(v\colon \mathrm{json}) &= v \in \mathrm{json}^D \wedge \mathrm{dom}(v) = \{\text{"dir"}, \text{"object"}\}\\ \mathrm{object}(v\colon \mathrm{json}) &= v \in \mathrm{json}^D \wedge \mathrm{dom}(v) = \{\text{"object"}\}\end{aligned}$$

And also predicates for segment items occurring at the ends of a segment:

$$\begin{aligned}\mathrm{first}(s \text{ in Si}) &= \exists x \text{ in Sg } (x_0 = s)\\ \mathrm{last}(s \text{ in Si}) &= \exists x \in \mathrm{Sg}\, (x_{|x|-1} = s)\\ \mathrm{end}(s \text{ in Si}) &= \mathrm{first}(s) \vee \mathrm{last}(s)\end{aligned}$$

Now, for a segment item v, $\omega(v)$ returns all objects contained in v.

$$\omega(v) = \begin{cases}\mathrm{set}(v.\mathrm{objects}[*]) & \text{if } \mathrm{generic}(v)\\ \mathrm{set}(v.\mathrm{object}) & \text{if } \mathrm{switch}(v)\\ \mathrm{set}(v[*].\mathrm{object}) & \text{if } \mathrm{colocated}(v)\end{cases}$$

Two segment items with the same contained objects represents the same *node*. For x, y in Si, let $x \simeq y$ iff $\omega(x) = \omega(y)$. The nodes in $ can then be represented by the quotient set

$$\mathrm{Nodes} = \mathrm{set}(\mathrm{Si})/\simeq$$

REQUIREMENT 27 (Node consistency)

1. An object can be contained in at most one node,
   $\forall x, y \text{ in Si } (\omega(x) \cap \omega(y) \neq \varnothing \Rightarrow x \simeq y)$
2. A segment contains no repeated node,
   $\forall s \text{ in Sg } (\forall i, j \in \mathrm{dom}(s)\ (s_i \simeq s_j \Rightarrow i = j))$
3. The keys of segment items are dependent of their position in a segment. Given a segment s with a switch or generic member s_i, the following shall apply:
   $$\begin{cases}\text{"out"} \in \mathrm{dom}(s_i) \wedge \text{"in"} \notin \mathrm{dom}(s_i) & \text{if } i = 0\\ \text{"out"} \notin \mathrm{dom}(s_i) \wedge \text{"in"} \in \mathrm{dom}(s_i) & \text{if } i = |s| - 1\\ \text{"out"} \in \mathrm{dom}(s_i) \wedge \text{"in"} \in \mathrm{dom}(s_i) & \text{otherwise}\end{cases}$$
4. Nodes have the same normal form,
   $\forall x, y \text{ in Si } (x \simeq y \Rightarrow (\mathrm{generic}(x) \wedge \mathrm{generic}(y)) \vee$
   $(\mathrm{switch}(x) \wedge \mathrm{switch}(y)) \vee (\mathrm{colocated}(x) \wedge \mathrm{colocated}(y)))$

REQUIREMENT 28 (Object type constraints)

1. A switch object may only be located in a generic node or switch node,
   $\forall s$ in Si $\forall x \in \omega(s)$ (bt(x) = "SwitchObject" $\Rightarrow$ generic$(s) \vee$ switch(s))
2. A directed object may only be located in a generic node or inside a co-located node,
   $\forall s$ in Si $\forall x \in \omega(s)$ (
   bt(x) = "DirectedObject" $\Rightarrow$ generic$(s) \vee$
   (colocated$(s) \wedge \exists y(y$ in $s \wedge$ directed$(y) \wedge y$.object$' = x)))$

REQUIREMENT 29 (Switch decorators)
A switch node has two or three occurrences in segments. This means that there will be three decorators for the node.

It is required that the decorators are "-", "~" and one of "/" or "\". If an occurrence of a switch node uses two decorators, one of them must be "~". In the formal requirement below, given a switch segment item x, let swd(x) = x.in + x.out, be the list of its decorators.

$\forall x$ in Si (switch$(x) \Rightarrow$
let t = [s | s in Si if $s \simeq x$] in
let r = conc(swd $\circ$ t) in
(set(r) = { "~", "-", "/"} $\vee$ set(r) = { "~", "-", "\"}) $\wedge$
$\forall y$ in t ($|$ swd$(y)| > 1 \Rightarrow$ "~" in swd(y))))

REQUIREMENT 30 (Signal decorators)
A directed object occurs once or twice among the segment items. In the latter case the two occurences will be included in an segment item located either at the start or end of a segment.

It is required that they have the same decorator value iff one of the items is at the start of a segment and the other is at the end.

$\forall s, t$ in Si $\forall y$ in s $\forall z$ in t (
colocated$(s) \wedge$ directed$(y) \wedge$
colocated$(t) \wedge$ directed$(z) \wedge$
$s \neq t \ \wedge y$.object$' = z$.object$' \Rightarrow$
(first$(s) \wedge$ last$(t) \vee$ first$(t) \wedge$ last$(s) \Leftrightarrow y$.dir$' = z$.dir$'))$

13.4 Edges

We first collect all consecutive pairs of segment items in $,

SiPairs = { $[x_i, x_{i+1}]$ | x in Sg $\wedge$ $i \in 0\,..\,(|x| - 2)$ }

For $s \in$ SiPairs, let $\beta(s) = [[s_0]_{\simeq}, [s_1]_{\simeq}]$, be the corresponding node pair. The *edges* of $ are all such pairs, i.e.

Edges = { $\beta(s)$ | $s \in$ SiPairs }

REQUIREMENT 31 (Edge consistency)
Edges are not repeated in segments, disregarding also the order of the edge. Formally, for $s \in$ SiPairs let $f(s) =$ set$(\beta(s))$, then it is required that f is injective.

13.5 Railyard interpretation

Given the requirements in previous sections, we can define the function $\psi(v)$, that returns the node containing an interior object v.

$$\psi(v : \text{IObjects}) = \{\, x \text{ in Si} \mid v \in \omega(x) \,\}$$

Let a *node map* for \$ be a 2-tuple (ν, ξ) where $\nu : \text{Nodes} \to \text{AllNodeTypes}_T$ and $\xi : \text{Edges} \to \text{int}^2$. Let $\xi_0(e) = a$ and $\xi_1(e) = b$ when $\xi(e) = (a, b)$. A node map is *admissible* iff the following conditions are satisfied.

1. The type of a node is compatible with its contained objects:
   $\forall x : \text{IObjects}\ (\nu(\psi(x)) \text{ in } \text{ot}(x).\text{allowed-node-types}')$
2. The interpretation of a switch node, shall be the mandatory type `SwitchNode` (c.f. Requirement 20):
   $\forall x \text{ in Si}\ (\text{switch}(x) \Rightarrow \nu([x]_{\simeq}) = \text{"SwitchNode"})$
3. The interpretation of a co-located node that contains a directed node, shall be the mandatory type `PassageNode` (c.f. Requirement 20):
   $\forall x \text{ in Si}\ (\text{colocated}(x) \wedge \exists y \text{ in } x\ (\text{directed}(y)) \Rightarrow \nu([x]_{\simeq}) = \text{ "PassageNode" })$
4. Node indices are compatible with their associated node type:
   $\forall e : \text{Edges}\ \forall i : \{\, 0, 1 \,\}\ (0 \le \xi_i(e) < \text{id}_T(\nu(e_i)).\text{degree}')$
5. Node indices can not be attached to multiple edges:
   $\forall d, e : \text{Edges}\ (d \neq e \Rightarrow$
   $\{\, [d_0, \xi_0(d)], [d_1, \xi_1(d)] \} \cap \{ [e_0, \xi_0(e)], [e_1, \xi_1(e)] \,\} = \varnothing)$
6. Node indices respects the traversal relation for consecutive edges in segments:
   $\forall s \text{ in } Sg\ \forall i \in 0..|s| - 3\ ($
   $\text{let } d = \beta([s_i, s_{i+1}]) \text{ and } e = \beta([s_{i+1}, s_{i+2}]) \text{ in}$
   $[\xi_1(d), \xi_0(e)] \text{ in } \text{id}_T(\nu(e_0)).\text{traversal}')$
7. Node indices respects explicit indices in generic items:
   $\forall s : \text{SiPairs}\ ($
   $\text{let } a = \xi_0(\beta(s)) \text{ and } b = \xi_1(\beta(s)) \text{ in}$
   $(\text{generic}(s_0) \wedge \text{out} \in \text{dom}(s_0) \Rightarrow s_0.\text{out}' = a) \wedge$
   $(\text{generic}(s_1) \wedge \text{in} \in \text{dom}(s_1) \Rightarrow s_1.\text{in}' = b))$
8. Node indices respects the intended meaning of switch decorators. Let f be the function $\{$ ("~", 0), ("/", 1), ("\", 2)$\}$ in
   $\forall s : \text{SiPairs}\ ($
   $\text{let } a = \xi_0(\beta(s)) \text{ and } b = \xi_1(\beta(s)) \text{ in}$
   $(\text{switch}(s_0) \wedge \text{out} \in \text{dom}(s_0) \wedge s_0.\text{out}' \in \text{dom}(f) \Rightarrow f(s_0.\text{out}') = a) \wedge$
   $(\text{switch}(s_1) \wedge \text{in} \in \text{dom}(s_1) \wedge s_1.\text{in}' \in \text{dom}(f) \Rightarrow f(s_1.\text{in}') = b))$
9. All indices for a node shall be connected to an edge,
   $\forall s \text{ in Si}\ ($
   $\text{let } t = [s]_{\simeq} \text{ and } a = \{\, x \mid x \in t \wedge \text{end}(x) \,\} \text{ and } b = t \setminus a \text{ in}$
   $2 * |b| + |a| = \text{id}_T(\nu(t)).\text{degree}')$

REQUIREMENT 32 (Node map)
There must exist an admissible node map for \$, and an application making use of the compact format must specify such a map for the interpretation of \$. Note that

this specification may also state preconditions for $ in order to ensure that the map is admissible. The full set of requirements for $ is therefore all requirements in this section and all extra requirements stated in the specification of the node map.

The remaining definitions and requirements are dependent on a given admissible node map (ν, ξ).

13.6 Paths

A list p of nodes is a *node path* iff

- p is unique,
- $|p| \geq 2$,
- consecutive pairs of nodes in p are in the symmetric closure of the edges, and
- the passage of nodes in p respects the traversal relation.

The last two conditions can be expressed formally by:

$$\begin{aligned}&\forall i : 1\,..\,(|p| - 2)\ \exists d, e : \text{Edges}\ \exists j, k \in \{0, 1\}\ (\\ &\quad [p_{i-1}, p_i] = [d_{1-j}, d_j] \wedge [p_i, p_{i+1}] = [e_k, e_{1-k}] \wedge \\ &\quad [\xi_j(d), \xi_k(e)]\ \text{in}\ \mathrm{id}_T(\nu(p_i)).\text{traversal}')\end{aligned}$$

Let the predicate $\text{npath}(p)$ be true iff p is a node path. Note that it follows that any segment in $.segments[*] corresponds to a node path from the restrictions of the node map. A list p of interior objects *denotes a unique path* in $ iff the following condition is satisfied:

$$\exists! q\,(\text{npath}(q) \wedge q_0 = \psi(p_0) \wedge q_{|q|-1} = \psi(p_{|p|-1}) \wedge (\psi \circ p) \sqsubseteq q)$$

REQUIREMENT 33 (Path consistency)

1. Each list in $.paths..path shall denote a unique path.
2. Each non-singleton list in $.areas..paths[*] shall denote a unique path.

14 Requirement Identifiers

The tables below provides stable identifiers to all the requirements in this document. The requirement numbers may be reordered in new versions of this document in case new requirements are added or existing requirements are removed. The identifiers will however be kept for the same requirement. Hence, all references to the requirements in this document shall use the identifiers only.

Table 1 Requirement identifiers

Nr	Id	Nr	Id	Nr	Id	Nr	Id
1	types-1	9	project-1	16	table-1	20	compact-1
2	types-2	10	project-2	17	table-2	21	compact-2
3	types-3	11	project-3	18	table-3	22	compact-3
4	types-4	12	project-4	19	table-4	23	compact-4
5	types-5	13	project-5			24	compact-5
6	types-6	14	project-6			25	compact-6
7	types-7	15	project-7			26	compact-7
8	types-8					27	compact-8
						28	compact-9
						29	compact-10
						30	compact-11
						31	compact-12
						32	compact-13
						33	compact-14

REFERENCES

[1] "The JavaScript Object Notation (JSON) Data Interchange Format," ISSN: 2070-1721, Dec. 2017.

[2] P. Larsson, "RJSON Format, Language Definition Document ," Prover Technology, RJSON-LDD version 1.1, May 2021.