# Package: tidyfault (via r-universe)

October 25, 2026

**Title** Tidy Fault Tree Analysis

**Version** 0.1.0

**Description** Fault tree analysis with tidy data. Represent a fault tree as two tables, one of nodes (the top event, gates and basic events) and one of edges, so the whole workflow stays in ordinary data frames. Derive the Boolean equation of the top event, enumerate the truth table, and find minimal cut sets with the Method of Obtaining Cut Sets (MOCUS) algorithm of Fussell and Vesely (1972), implemented in 'R' and, by default, in C++ through 'Rcpp'. Evaluate thousands of binary or probabilistic scenarios in one vectorized call, put uncertainty intervals around the top-event probability, follow it over time, and rank which component fix lowers it most. Lay out and draw the tree with 'tidygraph', 'igraph' and 'ggplot2'. Methods follow the ``Fault Tree Handbook'' of Vesely, Goldberg, Roberts and Haasl (1981) <https://www.nrc.gov/docs/ML1007/ML100780465.pdf>. A 'Python' port with the same functions ships in the same repository.

**URL** https://tidyfault.netlify.app, https://github.com/timothyfraser/tidyfault

**BugReports** https://github.com/timothyfraser/tidyfault/issues

**License** GPL-3

**Encoding** UTF-8

**Language** en-US

**Roxygen** list(markdown = TRUE)

**Depends** R (>= 4.1)

**Imports** dplyr, ggplot2, graphics, igraph, methods, purrr, Rcpp, rlang, scales, stats, stringr, tibble, tidyr, tidygraph

**LinkingTo** Rcpp

**Suggests** ggraph, knitr, rmarkdown, spelling, testthat (>= 3.0.0)

**VignetteBuilder** knitr

**Config/testthat/edition** 3

**LazyData** true

**Config/roxygen2/version** 8.1.0

**RoxygenNote** 7.3.3

**Config/pak/sysreqs** libglpk-dev libicu-dev libxml2-dev

**Repository** https://timothyfraser.r-universe.dev

**Date/Publication** 2026-10-14 20:10:35 UTC

**RemoteUrl** https://github.com/timothyfraser/tidyfault

**RemoteRef** HEAD

**RemoteSha** e8348334bc9a95c1de420422e461bf0c5d7b6ba7

# Contents


ai_edges . . . 3
ai_nodes . . . 4
ai_outcomes_binary . . . 5
ai_outcomes_prob . . . 5
ai_outcomes_rates . . . 6
ai_probs . . . 7
breach_edges . . . 7
breach_nodes . . . 8
calculate . . . 9
concentrate . . . 10
curate . . . 12
db_edges . . . 13
db_nodes . . . 14
db_outcomes_binary . . . 15
db_outcomes_prob . . . 16
db_outcomes_rates . . . 17
db_probs . . . 17
equate . . . 18
fakeedges . . . 19
fakenodes . . . 20
formulate . . . 21
gate . . . 22
gate_and . . . 23
gate_or . . . 24
gate_top . . . 25
get_gate . . . 26
illustrate . . . 28
it_security_data . . . 30
it_security_edges . . . 31
it_security_nodes . . . 31
it_security_outcomes_rates . . . 33
it_security_probs . . . 33

mocus . . . . . 34
mocus_r . . . . . 36
mocus_rcpp . . . . . 37
plot.tidyfault_tree . . . . . 38
populate . . . . . 40
quantify . . . . . 41
quantify_ci . . . . . 43
quantify_if . . . . . 45
quantify_prob . . . . . 48
quantify_prob_fast . . . . . 49
quantify_when . . . . . 50
security_edges . . . . . 52
security_nodes . . . . . 53
security_outcomes_binary . . . . . 54
security_outcomes_prob . . . . . 55
security_outcomes_rates . . . . . 55
security_probs . . . . . 56
simulate . . . . . 57
tabulate . . . . . 59

**Index** **61**

---

`ai_edges` `ai_edges` *AI agent failure fault tree edge dataset*

---

## Description

An example dataset of edges (connections) in an AI agent failure fault tree.

## Usage

```
ai_edges
```

## Format

`ai_edges`**:**

A data frame with 6 rows and 2 columns (the top event `T`, id 1, links only to its one gate `G0`, id 7, which links to gate `G3` and to `CWE`):

**from** Unique `id` of the source/`from` node from which edge originates.

**to** Unique `id` of the destination/`to` node that edge connects to.

## See Also

`ai_nodes` for the corresponding nodes dataset

## Examples

```
data("ai_edges")
head(ai_edges)
dim(ai_edges)
```

---

ai_nodes ai_nodes *AI agent failure fault tree nodes dataset*

---

## Description

An example dataset of nodes in an AI agent failure fault tree. This fault tree models scenarios where an AI agent fails to complete a task due to model errors, prompt issues, API failures, and context limitations.

## Usage

```
ai_nodes
```

## Format

`ai_nodes`**:**

A data frame with 7 rows and 3 columns:

**id** Unique identifier (1 to 7) for each node.

**event** Name of event. `"T"` means top event (AI agent task failure). `"G0"` (id 7, the top event's one gate) and `"G3"` are gates. The basic events are `"AF"` = API failure, `"TO"` = Timeout, `"RL"` = Rate limit, and `"CWE"` = Context window exceeded.

**type** `factor` classification as "top", "and", "or", or "not" (meaning "not" a gate).

## Details

The fault tree structure:

- Top event: AI agent task failure. The top event is not a gate: its one child is G0.
- G0 (OR): G3 OR Context window exceeded (any branch fails the task)
- G3 (OR): API failure OR Timeout OR Rate limit

Minimal cut sets: `AF`, `CWE`, `RL`, `TO`.

## See Also

`ai_edges` for the corresponding edges dataset

## Examples

```
data("ai_nodes")
head(ai_nodes)
dim(ai_nodes)
```

## ai_outcomes_binary *AI agent failure fault tree binary scenarios*

### Description

Ten example scenarios for the basic events of the AI agent failure fault tree, coded 1 if the event occurred and 0 if not. Use it with `quantify()` to evaluate whether the top event occurs under each scenario, or with `populate()`.

### Usage

```
ai_outcomes_binary
```

### Format

`ai_outcomes_binary`**:**

A tibble with 10 rows and 5 columns:

**scenario** Scenario number (1 to 10).

**AF, TO, RL, CWE** One 0/1 column per basic event; 1 means the event occurred.

### See Also

ai_probs, ai_outcomes_prob, `quantify()`, `populate()`

### Examples

```
data("ai_outcomes_binary")
head(ai_outcomes_binary)
```

## ai_outcomes_prob *AI agent failure fault tree outcome probabilities*

### Description

Example failure probabilities for the basic events of the AI agent failure fault tree, with one row per event. It holds the same values as ai_probs.

### Usage

```
ai_outcomes_prob
```

### Format

`ai_outcomes_prob`:

A tibble with 4 rows and 2 columns:

**event** Name of the basic event: `"AF"` = API failure, `"TO"` = Timeout, `"RL"` = Rate limit, `"CWE"` = Context window exceeded.

**probability** Probability that the event occurs, between 0 and 1.

### See Also

ai_probs, ai_outcomes_binary, ai_outcomes_rates

### Examples

```
data("ai_outcomes_prob")
ai_outcomes_prob
```

---

## `ai_outcomes_rates` `ai_outcomes_rates` *AI agent failure fault tree failure rates*

---

### Description

Example exponential failure rates for the basic events of the AI agent failure fault tree, expressed per 1000 requests.

### Usage

```
ai_outcomes_rates
```

### Format

`ai_outcomes_rates`:

A tibble with 4 rows and 3 columns:

**event** Name of the basic event: `"AF"` = API failure, `"TO"` = Timeout, `"RL"` = Rate limit, `"CWE"` = Context window exceeded.

**lambda** Failure rate (failures per 1000 requests).

**time_unit** Unit of time for `lambda`: `"per_1000_requests"`.

### See Also

ai_probs, ai_outcomes_prob

### Examples

```
data("ai_outcomes_rates")
ai_outcomes_rates
```

## ai_probs — ai_probs *AI agent failure fault tree event probabilities*

### Description

Example failure probabilities for the basic events of the AI agent failure fault tree (ai_nodes, ai_edges). Pass it as event_probs to populate().

### Usage

```
ai_probs
```

### Format

ai_probs:

A tibble with 4 rows and 2 columns:

**event** Name of the basic event: "AF" = API failure, "TO" = Timeout, "RL" = Rate limit, "CWE" = Context window exceeded.

**probability** Probability that the event occurs, between 0 and 1.

### See Also

ai_outcomes_prob, ai_outcomes_binary, populate()

### Examples

```
data("ai_probs")
ai_probs
```

## breach_edges — breach_edges *data breach fault tree edge dataset*

### Description

An example dataset of edges (connections) in the data breach fault tree described in breach_nodes.

### Usage

```
breach_edges
```

### Format

breach_edges:

A data frame with 7 rows and 2 columns:

**from** Unique id of the source/from node from which edge originates.

**to** Unique id of the destination/to node that edge connects to.

### See Also

breach_nodes for the corresponding nodes dataset

### Examples

```
data("breach_edges")
head(breach_edges)
```

---

## breach_nodes `breach_nodes` *data breach fault tree nodes dataset*

---

### Description

An example dataset of nodes in a small data breach fault tree. A breach occurs if credentials are stolen through phishing, or if an insider or an unauthorized party gets access.

### Usage

```
breach_nodes
```

### Format

`breach_nodes`**:**

A data frame with 8 rows and 3 columns:

**id** Unique identifier (1 to 8) for each node.

**event** Name of event. `"Breach"` is the top event. `"G1"`, `"G2"`, `"G3"` are gates. The basic events are `"Phishing"`, `"Credential"` (credential compromise), `"Insider"` (insider threat), and `"Unauthorized"` (unauthorized access).

**type** `factor` classification as "top", "and", "or", or "not" (meaning "not" a gate).

### Details

The fault tree structure:

- Top event: Breach
- G1 (OR): G2 OR G3
- G2 (AND): Phishing AND Credential
- G3 (OR): Insider OR Unauthorized

### See Also

breach_edges for the corresponding edges dataset

### Examples

```
data("breach_nodes")
head(breach_nodes)
```

`calculate` *Enumerate the truth table of a fault tree*

## Description

This function takes a supplied function `f` and computes the system failure outcomes (1 = failure vs. 0 = not) given every possible combination of input events failing (1) or not failing (0). Outputs a `data.frame`.

## Usage

```
calculate(f)
```

## Arguments

| | |
|---|---|
| `f` | (Required) Function outputted by `formulate()`, including arguments for all input events in the fault tree (excluding gates or top events). The function should accept binary (0/1) arguments for each basic event and return a numeric value. |

## Details

This function generates a complete truth table by:

- Extracting the formal arguments from the supplied function to identify all input events
- Creating a grid of all possible binary combinations (0/1) for each event using `tidyr::expand_grid()`
- Evaluating the function for each combination to determine system failure
- Converting outcomes to binary: any value >= 1 indicates failure (1), otherwise no failure (0)

The function is designed to work with boolean functions created by `formulate()` that represent fault tree logic. The outcome threshold (>= 1) allows for functions that may return counts or probabilities rather than strict boolean values.

## Value

A `data.frame` containing a truth table with:

- One column for each input event (named according to the function's formal arguments)
- An `outcome` column indicating system failure (1) or no failure (0)

Rows are arranged in descending order by `outcome`, with failure cases listed first. The truth table contains 2^n rows where n is the number of input events, representing all possible combinations of event states.

## See Also

`formulate` for creating the function from a boolean equation

## Examples

```
# Load dependencies
library(dplyr)
library(tidyfault)

# Load example data into our environment
data("fakenodes")
data("fakeedges")

# Build fault tree function and generate full truth table
formula <- curate(nodes = fakenodes, edges = fakeedges) %>%
   equate() %>%
   formulate()
truth_table <- calculate(formula)
truth_table

# Optional downstream use with MOCUS minimization
curate(nodes = fakenodes, edges = fakeedges) %>%
   concentrate(method = "mocus_rcpp")
```

---

`concentrate` *Find the minimal cut sets*

---

## Description

This function *concentrates* the boolean equation of a fault tree to find the minimum cutsets in the fault tree. It takes a data.frame outputted by the `curate()` function and applies one of the available MOCUS implementations to perform boolean minimization.

## Usage

```
concentrate(data, method = c("mocus_rcpp", "mocus_r", "mocus_original"))
```

## Arguments

| | |
|---|---|
| `data` | (Required) data.frame containing output from `curate()`. Expects the gates data.frame with columns `gate`, `type`, `set`, and `items`. |
| `method` | (Optional) Character string specifying the minimization algorithm. Default is `"mocus_rcpp"`. Supported values are `"mocus_rcpp"`, `"mocus_r"`, and `"mocus_original"`. |

## Details

This function performs boolean minimization to identify the minimum cutsets (minimal failure paths) in a fault tree using the MOCUS algorithm:

- **Rcpp MOCUS** (`method = "mocus_rcpp"`):
  - Uses the Rcpp-backed MOCUS implementation (`mocus_rcpp()`)
  - Recommended default for speed on larger trees

- **Pure-R fast MOCUS** (`method = "mocus_r"`):
  - Uses the optimized pure-R implementation (`mocus_r()`)
  - Useful when compiled code is unavailable
- **Original MOCUS** (`method = "mocus_original"`):
  - Uses the original MOCUS implementation (`mocus()` with `method = "mocus_original"`) to generate all cutsets
  - Returns the cutsets as character strings

Every method then minimizes the MOCUS products by absorption: duplicate cutsets and any cutset containing another cutset are dropped. Cutsets are ordered fewest events first, then by the sorted position of their events. Minimum cutsets represent the smallest combinations of basic events that can cause the top event (system failure) to occur.

## Value

A character vector where each element is a minimum cutset represented as a boolean expression (e.g., `"A * B"` for events A AND B). Each cutset represents a minimal set of events whose simultaneous occurrence causes system failure.

## See Also

`curate` for preparing gate data for MOCUS method, `mocus_rcpp`, `mocus_r`, and `mocus` for MOCUS implementations, `tabulate` for analyzing and summarizing minimum cutsets

## Examples

```
# Load dependencies
library(dplyr)
library(tidyfault)

# Load example data into our environment
data("fakenodes")
data("fakeedges")

# Extract minimum cutset from fault tree data
formula <- curate(nodes = fakenodes, edges = fakeedges) %>%
   equate() %>%
   formulate()
curate(nodes = fakenodes, edges = fakeedges) %>%
   concentrate(method = "mocus_rcpp") %>%
   tabulate(formula = formula, method = "mocus_rcpp")
```

curate *Build the gates table from nodes and edges*

## Description

This function *curates* a `data.frame` of N `gates` and the `sets` of events immediately linked to those gates.

## Usage

```
curate(nodes, edges)
```

## Arguments

| | |
|---|---|
| `nodes` | (Required) data.frame of N `nodes` (1 to N rows), with a unique `id` for each node, an `event` name describing the node (some events occur multiple times), and a `type` vector categorizing nodes as `"and"` \| `"or"` gates, `"top"` nodes, or `"not"` a gate or top node (i.e. a normal node) |
| `edges` | (Required) data.frame of N `edges` (1 to N rows), showing ties between the `from` node's `id` to the `to` node's `id`. |

## Details

This function performs the initial curation step in the fault tree analysis workflow:

- Checks the top event first: `nodes` must hold exactly one node of type `"top"`, it must have exactly one child, and that child must be an `"and"` or `"or"` gate. The top event carries no logic of its own, so a top event with several children (or with a basic event as its child) is an error that names the children and asks you to insert an `and` or `or` gate; no default logic is applied
- Filters nodes to only gates and top events (excludes basic events with `type == "not"`)
- Joins edge information to identify which events connect to each gate
- Converts gate relationships into boolean expressions: AND gates use `*` (multiplication), OR gates use `+` (addition)
- Wraps each gate's set expression in parentheses to preserve order of operations
- Creates both string (`set`) and list (`items`) representations of gate inputs for different downstream uses

The output serves as the foundation for subsequent steps: `equate()` uses the `set` column to build the complete boolean equation, while `mocus()` uses the `items` list column for cutset generation.

## Value

A `data.frame` (tibble) with one row per gate, containing:

- `gate`: Character vector of gate/event names
- `type`: Factor or character vector indicating gate type (`"top"`, `"and"`, or `"or"`)

- `class`: Factor with levels `"top"` and `"gate"`, classifying nodes as top events or intermediate gates
- `n`: Integer count of events immediately connected to this gate
- `set`: Character string containing the boolean expression for this gate's inputs, using `*` for AND operations and `+` for OR operations, wrapped in parentheses
- `items`: List column containing character vectors of event names that are inputs to this gate

Rows are sorted with top events first, followed by gates in alphabetical order.

## See Also

`equate` for building the complete boolean equation from curated gates, `illustrate` for visualizing the fault tree structure

## Examples

```
# Load dependencies
library(dplyr)
library(tidyfault)

# Load example data into our environment
data("fakenodes")
data("fakeedges")

# Build the gate-level representation used by downstream functions
gates <- curate(nodes = fakenodes, edges = fakeedges)
gates

# Example downstream workflow from curate() output
formula <- gates %>%
   equate() %>%
   formulate()
gates %>%
   concentrate(method = "mocus_rcpp") %>%
   tabulate(formula = formula, method = "mocus_rcpp")
```

---

# db_edges — db_edges *database system failure fault tree edge dataset*

---

## Description

An example dataset of edges (connections) in a database system failure fault tree.

## Usage

```
db_edges
```

## Format

db_edges:

A data frame with 13 rows and 2 columns (the top event T, id 1, links only to its one gate G0, id 14, which links to gates G1 to G4):

**from** Unique id of the source/from node from which edge originates.

**to** Unique id of the destination/to node that edge connects to.

## See Also

db_nodes for the corresponding nodes dataset

## Examples

```
data("db_edges")
head(db_edges)
dim(db_edges)
```

---

db_nodes db_nodes *database system failure fault tree nodes dataset*

---

## Description

An example dataset of nodes in a database system failure fault tree. This fault tree models scenarios where a database system becomes unavailable due to various failure modes including data corruption, access failures, storage issues, network problems, and hardware failures.

## Usage

```
db_nodes
```

## Format

db_nodes:

A data frame with 14 rows and 3 columns:

**id** Unique identifier (1 to 14) for each node.

**event** Name of event. "T" means top event (Database system unavailable). "G0", "G1", etc. mean gates; "G0" (id 14) is the top event's one gate. "DC" = Data corruption, "AF" = Access failure, "SF" = Storage failure, "BF" = Backup failure, "NF" = Network failure, "AUF" = Authentication failure, "HF" = Hardware failure, "MF" = Monitoring failure.

**type** factor classification as "top", "and", "or", or "not" (meaning "not" a gate).

### Details

The fault tree structure:

- Top event: Database system unavailable. The top event is not a gate: its one child is G0.
- G0 (OR): G1 OR G2 OR G3 OR G4 (any failure mode makes the database unavailable)
- G1 (OR): Data corruption OR Access failure
- G2 (AND): Storage failure AND Backup failure
- G3 (OR): Network failure OR Authentication failure
- G4 (AND): Hardware failure AND Monitoring failure

Minimal cut sets: `AF`, `AUF`, `DC`, `NF`, `BF*SF`, `HF*MF`.

### See Also

`db_edges` for the corresponding edges dataset

### Examples

```
data("db_nodes")
head(db_nodes)
dim(db_nodes)
```

---

## db_outcomes_binary — `db_outcomes_binary` *database system failure fault tree binary scenarios*

---

### Description

Ten example scenarios for the basic events of the database system failure fault tree, coded 1 if the event occurred and 0 if not. Use it with `quantify()` to evaluate whether the top event occurs under each scenario, or with `populate()`.

### Usage

```
db_outcomes_binary
```

### Format

`db_outcomes_binary`**:**

A tibble with 10 rows and 9 columns:

**scenario** Scenario number (1 to 10).

**DC, AF, SF, BF, NF, AUF, HF, MF** One 0/1 column per basic event; 1 means the event occurred.

### See Also

db_probs, db_outcomes_prob, `quantify()`, `populate()`

## Examples

```
data("db_outcomes_binary")
head(db_outcomes_binary)
```

---

db_outcomes_prob — db_outcomes_prob *database system failure fault tree outcome probabilities*

---

## Description

Example failure probabilities for the basic events of the database system failure fault tree, with one row per event. It holds the same values as db_probs.

## Usage

```
db_outcomes_prob
```

## Format

db_outcomes_prob**:**

A tibble with 8 rows and 2 columns:

**event** Name of the basic event: "DC" = Data corruption, "AF" = Access failure, "SF" = Storage failure, "BF" = Backup failure, "NF" = Network failure, "AUF" = Authentication failure, "HF" = Hardware failure, "MF" = Monitoring failure.

**probability** Probability that the event occurs, between 0 and 1.

## See Also

db_probs, db_outcomes_binary, db_outcomes_rates

## Examples

```
data("db_outcomes_prob")
db_outcomes_prob
```

## db_outcomes_rates — db_outcomes_rates *database system failure fault tree failure rates*

### Description

Example exponential failure rates for the basic events of the database system failure fault tree, expressed per hour.

### Usage

```
db_outcomes_rates
```

### Format

`db_outcomes_rates`**:**

A tibble with 8 rows and 3 columns:

**event** Name of the basic event: `"DC"` = Data corruption, `"AF"` = Access failure, `"SF"` = Storage failure, `"BF"` = Backup failure, `"NF"` = Network failure, `"AUF"` = Authentication failure, `"HF"` = Hardware failure, `"MF"` = Monitoring failure.

**lambda** Failure rate (failures per hour).

**time_unit** Unit of time for `lambda`: `"hours"`.

### See Also

db_probs, db_outcomes_prob

### Examples

```
data("db_outcomes_rates")
db_outcomes_rates
```

## db_probs — db_probs *database system failure fault tree event probabilities*

### Description

Example failure probabilities for the basic events of the database system failure fault tree (`db_nodes`, `db_edges`). Pass it as `event_probs` to `populate()`.

### Usage

```
db_probs
```

### Format

db_probs:

A tibble with 8 rows and 2 columns:

**event** Name of the basic event: "DC" = Data corruption, "AF" = Access failure, "SF" = Storage failure, "BF" = Backup failure, "NF" = Network failure, "AUF" = Authentication failure, "HF" = Hardware failure, "MF" = Monitoring failure.

**probability** Probability that the event occurs, between 0 and 1.

### See Also

db_outcomes_prob, db_outcomes_binary, populate()

### Examples

```
data("db_probs")
db_probs
```

---

## equate — *Write the Boolean equation of the top event*

---

### Description

This function extracts the equation for a fault tree from a data.frame of gates and sets, outputted by curate().

### Usage

```
equate(data)
```

### Arguments

| | |
|---|---|
| data | (Required) data.frame of N gates, containing columns for gate, type, and set. Outputted by curate(). Must have at least one row with class == "top" representing the top event. |

### Details

This function builds the complete boolean equation through an iterative substitution algorithm:

- Starts with the top event's set expression, which may contain references to other gates
- Iteratively replaces each gate name with its corresponding set boolean expression
- Continues until no gate names remain in any expression (only basic events remain)
- Returns the fully expanded equation from the top event (first row of the data.frame)

The algorithm processes gates in order and handles nested gate structures by repeatedly substituting until convergence. The substitution preserves the boolean operators (* for AND, + for OR) and parentheses structure created by curate().

### Value

A character string containing the complete boolean equation for the entire fault tree. The equation:

- Uses * (multiplication) to represent AND operations
- Uses + (addition) to represent OR operations
- Contains only basic event names (no gate references)
- Is wrapped in parentheses to preserve order of operations
- Represents the top event's boolean logic in terms of all basic events

This equation can be passed directly to `formulate()` to create an executable function.

### See Also

`curate` for creating the gates data.frame, `formulate` for converting the equation string into an executable function

### Examples

```
# Load dependencies
library(dplyr)
library(tidyfault)

# Load example data into our environment
data("fakenodes")
data("fakeedges")

# Build gate-level representation and expand to a full boolean equation
gates <- curate(nodes = fakenodes, edges = fakeedges)
equation <- equate(gates)
equation

# Example downstream use of equate() output
formula <- formulate(equation)
calculate(formula)
```

---

## fakeedges `fakeedges` *fault tree edge dataset*

---

### Description

An example dataset of edges (connections) in a demonstration fault tree.

### Usage

```
fakeedges
```

### Format

`fakeedges`:

A data frame with 11 rows and 2 columns:

**from** Unique `id` of the source/`from` node from which edge originates.

**to** unique `id` of the destination/`to` node that edge connects to. ...

### Examples

```
data("fakeedges")
head(fakeedges)
dim(fakeedges)
```

---

## fakenodes `fakenodes` *fault tree nodes dataset*

---

### Description

An example dataset of nodes in a demonstration fault tree.

### Usage

```
fakenodes
```

### Format

`fakenodes`:

A data frame with 12 rows and 3 columns:

**id** Unique identifier (1 to N) for each node. Some events occur multiple times, but each time they have a unique `id`

**event** name of event. `"T"` mean top event. `"G1"`, `"G2"`, etc. mean gates.

**type** `factor` classification as "top", "and", "or", or "not" (meaning "not" a gate). ...

### Examples

```
data("fakenodes")
head(fakenodes)
dim(fakenodes)
```

`formulate` *Turn the Boolean equation into an R function*

## Description

This function *formulates* a character string (the boolean equation of a fault tree) into an R `function` that evaluates system failure. Handles AND and OR operations.

## Usage

```
formulate(formula)
```

## Arguments

| | |
|---|---|
| `formula` | (Required) A character string containing the boolean logic equation of a fault tree. Should use `*` for AND operations, `+` for OR operations, and parentheses for grouping. The equation should contain only basic event names (no gate references). Typically output from `equate()`. |

## Details

This function converts a boolean equation string into an executable R function through the following process:

- Extracts all unique event names from the formula by splitting on operators (`+`, `*`, `(`, `)`)
- Creates formal arguments for the function, one for each unique event name
- Sets the function body to parse and evaluate the formula string
- Returns a function that can be called with named arguments for each event

The function uses R's arithmetic operators where multiplication (`*`) represents AND logic and addition (`+`) represents OR logic. When called with binary inputs (0/1), the function returns 0 for no failure and a positive value for failure. The function preserves the order of operations through parentheses in the original formula.

## Value

A function object that:

- Accepts named arguments for each basic event in the formula
- Each argument accepts numeric values (typically 0 or 1 for binary events)
- Returns a numeric value representing the system state (typically 0 for no failure, >= 1 for failure)
- Evaluates the boolean equation using R's arithmetic operators (`*` for AND, `+` for OR)

The function can be called directly with event values or passed to `calculate()` to generate a complete truth table.

### See Also

equate for generating the boolean equation string, calculate for generating truth tables from the function

### Examples

```
# Load dependencies
library(dplyr)
library(tidyfault)

# Load example data into our environment
data("fakenodes")
data("fakeedges")

# Build a boolean equation and convert it to an executable function
equation <- curate(nodes = fakenodes, edges = fakeedges) %>%
   equate()
formula <- formulate(equation)
formula

# Evaluate across all combinations of basic events
calculate(formula)
```

---

## gate — *Draw gate polygons for a nodes table*

---

### Description

This internal function gathers all gate polygons for a set of supplied gate types and their x and y coordinates.

### Usage

```
gate(data, group = "id", gate = "type", size = 1, res = 50)
```

### Arguments

| | |
|---|---|
| data | (Required) data.frame of nodes with coordinates. Must include columns for x and y coordinates, and at least one row where the gate type is "and", "or", or "top". |
| group | (Default = "id") Character string naming the column in data that contains unique identifiers for grouping nodes. Default is "id". |
| gate | (Default = "type") Character string naming the column in data that contains gate types. Must contain values "and", "or", or "top". Default is "type". |
| size | (Default = 1) Numeric value controlling the diameter of gate polygons. Larger values create bigger gate shapes. |
| res | (Default = 50) Numeric value controlling the number of line segments used to draw curved gate shapes. Higher values create smoother curves but require more computation. |

### Details

This function generates polygon coordinates for visualizing gates in fault tree diagrams:

- Filters nodes to only those with gate types (`"and"`, `"or"`, `"top"`)
- Groups nodes by their identifier and gate type
- Calls `get_gate()` for each group to generate the appropriate polygon shape
- Returns all polygon coordinates in a single data.frame suitable for use with `geom_polygon()` in ggplot2

The function is typically called internally by `illustrate()` but can be used directly to generate gate shapes for custom visualizations. Each gate type (AND, OR, top event) has a distinct shape generated by the corresponding `gate_*()` function.

### Value

A data.frame (tibble) with columns `x` and `y` containing coordinate pairs that define gate polygon shapes. The data.frame is grouped by the `group` and `gate` columns, with one polygon per unique combination. Coordinates are centered at the node's position and scaled according to `size`.

### See Also

`get_gate` for generating individual gate polygons, `gate_and` for AND gate shapes, `gate_or` for OR gate shapes, `gate_top` for top event shapes

### Examples

```
# A small nodes table: one top event, one AND gate and one OR gate
nodes <- tibble::tibble(
  id = 1:3,
  event = c("T", "G1", "G2"),
  type = factor(c("top", "and", "or"), levels = c("top", "and", "or", "not")),
  x = c(0, -1, 1),
  y = c(1, 0, 0)
)
# One polygon per gate; columns x, y, group and gate
head(gate(nodes, size = 0.5, res = 16))
```

---

## gate_and *Polygon for an AND gate*

---

### Description

This function produces the AND gate polygons for fault trees.

### Usage

```
gate_and(size = 2, res)
```

### Arguments

| | |
|---|---|
| `size` | (Default = 2) Numeric value controlling the base size/diameter of the AND gate shape. The shape is scaled proportionally. Default is 2. |
| `res` | (Required) Numeric value describing the resolution, or number of line segments used to draw the curved portions of the gate shape. Higher values create smoother curves. Typically set to 50 or higher. |

### Details

This function generates the geometric shape for AND gates in fault tree visualizations:

- Creates a rounded rectangle shape with curved sides
- Uses square root functions to generate smooth curves on the left and right sides
- Centers the shape at the origin (0, 0) for easy translation
- The shape follows standard fault tree analysis conventions where AND gates are represented as rounded rectangles

The AND gate shape is generated using mathematical curves: the left side uses `sqrt(x)` and the right side uses `sqrt(size - x)` to create symmetric rounded edges. The shape is then centered vertically and horizontally.

### Value

A data.frame (tibble) with columns `x` and `y` containing coordinate pairs that define an AND gate polygon shape. The coordinates are centered at the origin (0, 0) and represent a rounded rectangle shape characteristic of AND gates in fault tree diagrams. The shape is symmetric and can be translated and scaled as needed.

### Examples

```
# Polygon of an AND gate: one row per vertex, columns x and y
head(gate_and(size = 1, res = 12))
```

---

## gate_or *Polygon for an OR gate*

---

### Description

This function produces the OR gate polygons for fault trees.

### Usage

```
gate_or(size = 2, res)
```

### Arguments

| | |
|---|---|
| `size` | (Default = 2) Numeric value controlling the base size/diameter of the OR gate shape. The shape is scaled proportionally. Default is 2. |
| `res` | (Required) Numeric value describing the resolution, or number of line segments used to draw the curved portions of the gate shape. Higher values create smoother curves. Typically set to 50 or higher. |

### Details

This function generates the geometric shape for OR gates in fault tree visualizations:

- Creates a curved shape with a concave top edge and convex bottom edge
- Uses square root functions with different scaling factors for the top and bottom curves
- Combines two curve segments (top and bottom) to form the complete gate shape
- Centers the shape at the origin (0, 0) for easy translation
- The shape follows standard fault tree analysis conventions where OR gates are represented as curved shapes

The OR gate shape uses different mathematical curves for the top (concave, using `1/size * sqrt(x)`) and bottom (convex, using `size * sqrt(x)`) to create the distinctive curved appearance. The two curve segments are combined and centered to form the final polygon.

### Value

A data.frame (tibble) with columns `x` and `y` containing coordinate pairs that define an OR gate polygon shape. The coordinates are centered at the origin (0, 0) and represent a curved shape with a concave top and convex bottom, characteristic of OR gates in fault tree diagrams. The shape is symmetric and can be translated and scaled as needed.

### Examples

```
# Polygon of an OR gate: one row per vertex, columns x and y
head(gate_or(size = 1, res = 12))
```

---

## `gate_top` *Polygon for a top-event gate*

---

### Description

This function produces the top event gate polygons for fault trees.

### Usage

```
gate_top(size = 2, res)
```

### Arguments

| | |
|---|---|
| `size` | (Default = 2) Numeric value controlling the size of the top event rectangle. The shape is a square with sides of length `size`. Default is `2`. |
| `res` | (Required) Numeric value describing the resolution. For rectangular shapes, this parameter is accepted for consistency with other gate functions but does not affect the output (rectangles don't require curve resolution). |

### Details

This function generates the geometric shape for top events in fault tree visualizations:

- Creates a simple rectangular (square) shape
- Defines the rectangle using corner coordinates
- Centers the shape at the origin (0, 0) for easy translation
- The shape follows standard fault tree analysis conventions where top events are represented as rectangles

The top event shape is the simplest gate shape, consisting of a square defined by four corner points. The rectangle is created by specifying the four corners in order, then centering the shape by offsetting coordinates by `size/2` in both `x` and `y` directions. Unlike AND and OR gates, the top event shape does not use curved edges, so the `res` parameter is included for consistency but doesn't affect the output.

### Value

A data.frame (tibble) with columns `x` and `y` containing coordinate pairs that define a rectangular top event polygon shape. The coordinates are centered at the origin (0, 0) and represent a square shape characteristic of top events in fault tree diagrams. The shape can be translated and scaled as needed.

### Examples

```
# Polygon of a top event: the four corners of a square, columns x and y
head(gate_top(size = 2, res = 12))
```

---

## `get_gate` *Place one gate polygon at x, y*

---

### Description

This function produces the gate polygons for a set of supplied gate types and their x and y coordinates.

### Usage

```
get_gate(x, y, gate, size = 1, res = 50)
```

### Arguments

| | |
|---|---|
| `x` | (Required) Numeric value or vector representing the x coordinate(s) of the gate center in the layout. |
| `y` | (Required) Numeric value or vector representing the y coordinate(s) of the gate center in the layout. |
| `gate` | (Required) Character vector of gate type names. Must contain one or more of `"top"`, `"and"`, or `"or"`. If multiple values are provided, only the first unique value is used. |
| `size` | (Default = 1) Numeric value controlling the diameter of the gate polygon. The shape is scaled proportionally around the center point. Default is `1`. |
| `res` | (Default = 50) Numeric value controlling the resolution (number of line segments) used to draw curved gate shapes. Higher values create smoother curves. Default is `50`. |

### Details

This function is a wrapper that selects and generates the appropriate gate shape based on the gate type:

- Determines the gate type from the `gate` parameter (uses first unique value if multiple provided)
- Calls the appropriate shape-generating function: `gate_and()` for AND gates, `gate_or()` for OR gates, or `gate_top()` for top events
- Scales and translates the base shape to match the specified `(x, y)` coordinates and `size` parameter
- Returns coordinates ready for use with `geom_polygon()` in ggplot2

The function handles coordinate transformation using `scales::rescale()` to position the gate shape correctly. Each gate type has a distinct visual representation following standard fault tree analysis conventions.

### Value

A data.frame (tibble) with columns `x` and `y` containing coordinate pairs that define the gate polygon shape. The coordinates are centered at the provided `(x, y)` position and scaled according to `size`. The shape depends on the gate type: AND gates have a rounded rectangle shape, OR gates have a curved shape, and top events have a rectangular shape.

### See Also

`gate_and` for AND gate shape generation, `gate_or` for OR gate shape generation, `gate_top` for top event shape generation

### Examples

```
# An AND gate polygon centered at (0, 0); columns x and y
head(get_gate(0, 0, gate = "and", size = 1, res = 10))
```

## illustrate — *Lay out a fault tree for plotting*

### Description

This function takes nodes and edges from a fault tree, computes a **tree** layout using `tidygraph` and `igraph`, and returns ggplot-friendly data.frames of nodes, edges, and gate polygons.

### Usage

```
illustrate(
  nodes,
  edges,
  type = c("nodes", "edges", "both", "all"),
  node_key = "id",
  layout = "tree",
  size = 0.25,
  scale_size = FALSE,
  res = 50
)
```

### Arguments

| | |
|---|---|
| `nodes` | (Required) dataset of nodes, including an `id` field (or whatever is written in `node_key`) |
| `edges` | (Required) dataset of edges, including `from` and `to` fields corresponding to the `id` of each node |
| `type` | (Optional) Character string specifying output format. Default is `"both"`, which returns a list with `"nodes"`, `"edges"`, and `"gates"`. Other options: `"nodes"` returns only node coordinates, `"edges"` returns only edge coordinates, `"all"` returns nodes, edges, gates, and pairwise edge data. |
| `node_key` | (Optional) Character string naming the column in `nodes` that contains unique node identifiers. Default is `"id"`. |
| `layout` | (Optional) Character string specifying the graph layout algorithm. Default is `"tree"`, the Reingold-Tilford tree layout of `igraph`. Other igraph layouts work by short name (e.g., `"fr"`, `"kk"`, `"nicely"`, `"sugiyama"`). Layouts that only `ggraph` provides, such as `"dendrogram"`, need `ggraph` installed. |
| `size` | (Optional) Numeric value passed to `gate()` function for constructing polygons. Controls the diameter of gate shapes. Defaults to `0.25`. When `scale_size = TRUE`, this is scaled by the layout extent. |
| `scale_size` | (Optional) Logical. If `TRUE`, scale gate size to the layout extent. Default is `FALSE`; set to `TRUE` to try consistent proportions across very different trees. |
| `res` | (Optional) Numeric value passed to `gate()` function for constructing polygons. Controls the number of line segments used to draw curved gate shapes. Defaults to `50`. |

## Details

This function prepares fault tree data for visualization with ggplot2 by:

- Creating a `tidygraph` object from nodes and edges
- Computing graph layout coordinates with the specified `igraph` layout algorithm, as `ggraph::create_layout()` does
- Extracting node coordinates and preserving original node attributes
- Computing edge coordinates by joining node positions to edge endpoints
- Generating gate polygon shapes for AND, OR, and top event gates using the `gate()` function

The output is designed to work seamlessly with ggplot2. Nodes can be plotted with `geom_point()`, edges with `geom_line()`, and gates with `geom_polygon()`. The tree layout algorithm positions nodes hierarchically, making fault tree structure easy to visualize.

## Value

The return type depends on the `type` parameter:

- For `type = "nodes"`: A data.frame with node coordinates (`x`, `y`) and all original columns from `nodes`
- For `type = "edges"`: A data.frame with edge coordinates formatted for `geom_line()`, containing columns `direction`, `id`, `x`, `y`, `edge_id`
- For `type = "both"` (default): A named list of class `c("tidyfault_tree", "list")` (so `plot()` dispatches to `plot.tidyfault_tree()`) with three elements:
  - `nodes`: Node coordinates data.frame
  - `edges`: Edge coordinates data.frame
  - `gates`: Gate polygon coordinates data.frame with `x`, `y` columns for drawing gate shapes
- For `type = "all"`: A named list of class `c("tidyfault_tree", "list")` with four elements: `nodes`, `edges`, `gates`, and `pairwise` (edge data with from/to coordinates)

## See Also

`gate` for generating gate polygons, `tbl_graph` for creating graph objects, `layout_as_tree` for the tree layout

## Examples

```
data(db_nodes, db_edges, package = "tidyfault")

# A tibble of laid-out nodes, with x and y coordinates added
illustrate(db_nodes, db_edges, type = "nodes")

# A list of nodes, edges and gate polygons; plot() draws it
ill <- illustrate(db_nodes, db_edges, type = "both")
plot(ill)
```

## it_security_data `it_security_data` *Binary scenario data for IT security fault tree*

### Description

A small tibble of binary (0/1) scenarios for the 10 basic events in the IT security fault tree (sensitive data leaves the organization). Used with `quantify()` to evaluate whether the top event occurs under each scenario.

### Usage

```
it_security_data
```

### Format

`it_security_data`**:**

A data frame with 3 rows and 10 columns. Each column is a basic event (two-letter code); 1 indicates the event occurred (condition true), 0 otherwise.

**DA** No controls to stop data from being copied or sent out.

**EP** User had more access than needed for their job.

**IM** Employee did it on purpose or account was taken over.

**LR** Passwords exposed in a breach or reused elsewhere.

**MN** No second login step (e.g. code on phone) required.

**PO** Security updates not applied on time.

**PC** Passwords obtained via fake email or link (phishing).

**PM** Admin-level access set up incorrectly.

**VS** System has a known security hole.

**WB** Web traffic filter missing or attacker got around it.

### See Also

`it_security_nodes`, `quantify`

### Examples

```
data("it_security_data")
head(it_security_data)
dim(it_security_data)
```

## it_security_edges `it_security_edges` *IT security (data leak) fault tree edge dataset*

### Description

An example dataset of edges (connections) in an IT security fault tree for sensitive data leaving the organization.

### Usage

```
it_security_edges
```

### Format

`it_security_edges`**:**

A data frame with 15 rows and 2 columns:

**from** Source node `id`.

**to** Destination node `id`.

### See Also

`it_security_nodes` for the corresponding nodes dataset

### Examples

```
data("it_security_edges")
head(it_security_edges)
dim(it_security_edges)
```

## it_security_nodes `it_security_nodes` *IT security (data leak) fault tree nodes dataset*

### Description

An example dataset of nodes in a fault tree for sensitive data leaving the organization. The tree models three ways a breach can happen: an outsider gets high-level access, an attacker uses a known flaw that was not fixed, or someone inside sends data out.

### Usage

```
it_security_nodes
```

## Format

`it_security_nodes`:

A data frame with 16 rows and 3 columns:

**id** Unique identifier (1 to 16) for each node.

**event** Two-letter event code. "DE" = Sensitive data leaves the organization (top). Gates: "BP" = Ways a breach can happen (OR), "EA" = Outsider gets high-level access (AND), "UE" = Attacker uses a known flaw that was not fixed (AND), "IE" = Someone inside sends data out (AND), "CC" = Log-in details stolen or misused (OR). Basic events: "PC" = Passwords obtained via fake email or link, "LR" = Passwords exposed in a breach or reused elsewhere, "MN" = No second login step required, "PM" = Admin-level access set up incorrectly, "VS" = System has a known security hole, "PO" = Security updates not applied on time, "WB" = Web traffic filter missing or bypassed, "IM" = Employee did it on purpose or account was taken over, "DA" = No controls to stop data from being copied out, "EP" = User had more access than needed for their job.

**type** Factor classification as "top", "and", "or", or "not".

## Details

The fault tree structure (event codes):

- Top event: DE (Sensitive data leaves the organization)
- BP (OR): EA OR UE OR IE
- EA (AND): CC AND MN AND PM
- UE (AND): VS AND PO AND WB
- IE (AND): IM AND DA AND EP
- CC (OR): PC OR LR

## See Also

`it_security_edges` for the corresponding edges dataset

## Examples

```
data("it_security_nodes")
head(it_security_nodes)
dim(it_security_nodes)
```

## it_security_outcomes_rates

`it_security_outcomes_rates` *IT security (data leak) fault tree failure rates*

### Description

Example exponential failure rates for the 10 basic events of the IT security fault tree, expressed per hour (the reliability-engineering convention): a rate of 0.25 failures per year is stored as 0.25 / 8760 failures per hour, so `pexp(8760, lambda)` is the chance of failing within one year.

### Usage

```
it_security_outcomes_rates
```

### Format

`it_security_outcomes_rates`**:**

A tibble with 10 rows and 3 columns:

**event** Two-letter code of the basic event (`"DA"`, `"EP"`, `"IM"`, `"LR"`, `"MN"`, `"PO"`, `"PC"`, `"PM"`, `"VS"`, `"WB"`); see it_security_data for what each code means.

**lambda** Failure rate (failures per hour).

**time_unit** Unit of time for `lambda`: `"hours"`.

### See Also

it_security_probs, it_security_data

### Examples

```
data("it_security_outcomes_rates")
it_security_outcomes_rates
```

## it_security_probs

`it_security_probs` *IT security (data leak) fault tree event probabilities*

### Description

Example probabilities for the 10 basic events of the IT security fault tree (`it_security_nodes`, `it_security_edges`), in wide format: one row and one column per basic event. Pass it to `quantify()` with `prob = TRUE`.

### Usage

```
it_security_probs
```

### Format

`it_security_probs`:

A tibble with 1 row and 10 columns. Each column is a basic event (two-letter code, as described in it_security_data) and holds the probability, between 0 and 1, that the event occurs: `DA`, `EP`, `IM`, `LR`, `MN`, `PO`, `PC`, `PM`, `VS`, `WB`.

### See Also

it_security_data, it_security_outcomes_rates, `quantify()`

### Examples

```
data("it_security_probs")
it_security_probs
```

---

## mocus — *Cut sets by the MOCUS algorithm*

---

### Description

This function *simplifies* a `data.frame` of gates and their sets identified by `curate()`, generating each cutset in the fault tree using the MOCUS (Method of Obtaining Cutsets) algorithm.

### Usage

```
mocus(data, method = c("mocus_rcpp", "mocus_r", "mocus_original"))
```

### Arguments

| | |
|---|---|
| `data` | (Required) data.frame containing gates and their sets, outputted by `curate()`. Must contain columns `gate`, `type`, `class`, and `items` (list column of event vectors). The data.frame must have exactly one row with `class == "top"`, whose one item is an AND/OR gate (see `curate()`). |
| `method` | (Optional) Character string specifying which implementation to run. Default is `"mocus_rcpp"`. Supported values are `"mocus_rcpp"`, `"mocus_r"`, and `"mocus_original"` (same labels as `concentrate()` and `tabulate()`). |

## Details

This function implements the MOCUS (Method of Obtaining Cutsets) algorithm, which systematically expands gates in a fault tree to identify all cutsets. The algorithm works as follows:

- **Initialization**: Starts with the top event's one gate as the first cutset (the top event itself carries no logic and is never expanded)
- **Iterative Expansion**: For each cutset containing gate references:
  - Identifies gates that appear in the current cutset
  - For AND gates: Replaces the gate with all its input events (events are combined)
  - For OR gates: Creates separate cutsets for each input path (events are alternatives)
  - Removes the expanded gate from consideration
- **Convergence**: Continues until no gates remain in any cutset (only basic events)
- **Deduplication**: Removes duplicate events within each cutset

The algorithm handles nested gate structures by repeatedly expanding gates until all references are resolved. AND gates represent events that must all occur together, while OR gates represent alternative failure paths. The result includes all possible cutsets, which can then be minimized using boolean algebra (e.g., via `concentrate()` with `method = "mocus_original"`).

The default method (`"mocus_rcpp"`) uses the compiled implementation for speed. Use `"mocus_r"` for a pure-R queue-based variant, or `"mocus_original"` for the historical R loop used in early versions of the package.

## Value

A list where each element is a character vector representing a cutset (a set of basic events that can cause system failure). Each cutset vector contains the names of basic events. The list includes all cutsets found in the fault tree, not just minimum cutsets. Duplicate events within each cutset are removed (each event appears only once per cutset).

## See Also

`curate()` for preparing the gates data.frame, `concentrate()` for finding minimum cutsets from the generated cutsets, `mocus_rcpp()`, `mocus_r()`

## Examples

```
data(db_nodes, db_edges, package = "tidyfault")
gates <- curate(db_nodes, db_edges)

# mocus(), mocus_r() and mocus_rcpp() return the same cut sets;
# concentrate() then minimizes them.
mocus(gates)
```

`mocus_r` *MOCUS in pure R*

## Description

Drop-in replacement for mocus() using a queue-based expansion, pre-indexed gate lookup, and integer-encoded events. Same inputs/outputs as mocus().

## Usage

```
mocus_r(data)
```

## Arguments

| | |
|---|---|
| `data` | data.frame produced by curate(). Must have columns: gate (chr), type (chr/fct), class (chr/fct), items (list of chr vectors). |

## Details

Key improvements over original:

1. Pre-indexes gate table into a named list — zero dplyr::filter() calls inside the loop.
2. Integer-encodes every event/gate name once up front; all inner-loop operations run on integer vectors (faster %in%, faster unique()).
3. Queue-based (append-only) expansion — never deletes/reallocates the working list mid-loop. A tail pointer advances; new cutsets are appended to the end.
4. Expands every gate in a cutset in a single pass rather than one gate per outer-loop iteration.
5. Convergence is tracked with a simple counter, not by re-flattening the entire list every iteration.

## Value

Named list where each element is a character vector of basic events forming one cutset. Identical structure to mocus() output.

## Examples

```
data(db_nodes, db_edges, package = "tidyfault")
gates <- curate(db_nodes, db_edges)

# mocus(), mocus_r() and mocus_rcpp() return the same cut sets;
# concentrate() then minimizes them.
mocus_r(gates)
```

`mocus_rcpp` *MOCUS with the C++ backend*

## Description

Drop-in replacement for mocus() that delegates the inner expansion loop to compiled C++ via Rcpp. The R layer handles encoding/decoding; C++ handles the queue-based gate expansion with zero R-level allocation in the hot path.

## Usage

```
mocus_rcpp(data)

mocus_cpp(data)
```

## Arguments

| | |
|---|---|
| `data` | data.frame produced by curate(). Must have columns: gate (chr), type (chr/fct), class (chr/fct), items (list of chr vectors). |

## Details

Encoding strategy:

- Every unique token (gate name or basic event name) is mapped to a positive integer ID once. All C++ operations work on ints.
- Gate children are stored in a flat integer vector with per-gate start/end offsets, giving cache-friendly sequential access.
- After C++ returns a list of integer-vector cutsets, R decodes them back to character vectors in one vectorized pass.

## Value

Named list where each element is a character vector of basic events forming one cutset. Identical structure to mocus() / mocus_r() output.

## Examples

```
data(db_nodes, db_edges, package = "tidyfault")
gates <- curate(db_nodes, db_edges)

# mocus(), mocus_r() and mocus_rcpp() return the same cut sets;
# concentrate() then minimizes them.
mocus_rcpp(gates)
```

## plot.tidyfault_tree *Plot a fault tree from illustrate() output*

### Description

An S3 method for plot on the tidyfault_tree objects returned by illustrate(type = "both") and illustrate(type = "all"). Calling plot() on any other object dispatches to base R as usual.

### Usage

```
## S3 method for class 'tidyfault_tree'
plot(
  x,
  type_col = "type",
  gate_types = c("and", "or", "top"),
  edge_linewidth = 1,
  gate_linewidth = 0.8,
  gate_alpha = 1,
  point_radius = NULL,
  point_n = 100,
  point_linewidth = 0.8,
  outline_colour = "#0b0405",
  gate_fill = c(top = "#382a54", and = "#395d9c", or = "#3eb4ad", `basic event` =
    "#def5e5"),
  coord_fixed = TRUE,
  theme_void = TRUE,
  expand = 0.2,
  normalize = TRUE,
  basic_radius_ratio = 0.18,
  edge_colour = "#b7c1cb",
  ...
)
```

### Arguments

| | |
|---|---|
| x | A tidyfault_tree list returned by illustrate(..., type = "both") or illustrate(..., type = "all"), with elements nodes, edges, and gates. |
| type_col | Character. Name of the column in x$nodes that holds node type (e.g. "top", "and", "or", "not"). Default is "type". |
| gate_types | Character vector. Node types that are drawn as gate polygons. Default is c("and", "or", "top"). |
| edge_linewidth | Numeric. Line width for edges. Default 1. |
| gate_linewidth | Numeric. Outline width for gate polygons. Default 0.8. |
| gate_alpha | Numeric. Fill opacity for gate polygons (0–1). Default 1. |

point_radius
: Numeric or NULL. Radius of basic-event circles in data units. If NULL (default), it is estimated from gate sizes (see basic_radius_ratio). Set explicitly (e.g. 0.1) for full control when auto sizing is too small or too big.

point_n
: Integer. Number of vertices used to draw each basic-event circle polygon. Default 100.

point_linewidth
: Numeric. Outline width for basic-event circles. Default 0.8.

outline_colour
: Character. Color for gate polygon and basic-event outlines. Default "black".

gate_fill
: Named character vector of fill colors for gate types (and, or, top) and optionally "basic event" for leaf nodes. Default is c(top = "#382a54", and = "#395d9c",or = "#3eb4ad", "basic event" = "#def5e5"). If NULL, the viridis discrete palette is used instead.

coord_fixed
: Logical. If TRUE (default), use equal aspect ratio so shapes are not stretched.

theme_void
: Logical. If TRUE (default), remove axes and panel.

expand
: Numeric. Fixed padding (in data units) added to the view limits on each side. Default 0.2. For trees with few nodes, the function automatically tightens this to 0.1.

normalize
: Logical. If TRUE (default), scale and center node, edge, and gate coordinates so the plot extent is standardized across different trees.

basic_radius_ratio
: Numeric. Used only when point_radius = NULL. Basic-event radius is set to this fraction of the median gate extent (each gate's extent is half the sum of its x and y span). Default 0.18. Larger values give bigger circles; smaller values give smaller circles.

edge_colour
: Character. Color for edges. Default "#b7c1cb".

...
: Ignored (for compatibility with the generic plot).

## Details

Builds a ggplot of the fault tree using the nodes, edges, and gate polygons returned by illustrate. Gate nodes (and, or, top) are drawn as polygons with outlines that follow their shape; other nodes (e.g. basic events) are drawn as circle polygons in the same data coordinate system, so their size stays visually consistent relative to the gates. Nodes and gates are labeled using the event column from x$nodes.

To avoid pixelated output when saving, use ggsave() with an explicit dpi (e.g. ggsave("plot.png", p, dpi = 300)). Saving from the RStudio Plots pane or at a small size often produces low-resolution images.

Basic-event circle size when point_radius = NULL: the function takes each gate polygon's extent (half of x-span + y-span), then uses the *median* of those extents times basic_radius_ratio. So circle size follows a typical gate size. It can still look too small or too big when gate sizes vary a lot (e.g. one large top gate and many small gates), or when the layout is very compact or very spread out. In those cases, set point_radius explicitly (e.g. 0.08 or 0.15) or adjust basic_radius_ratio.

## Value

A ggplot object.

## Examples

```
data(it_security_nodes, it_security_edges, package = "tidyfault")
ill <- illustrate(nodes = it_security_nodes, edges = it_security_edges, type = "both")
p <- plot(ill)
ggplot2::ggsave(tempfile(fileext = ".png"), p, dpi = 300)
```

---

`populate` *Fill a binary scenario table with event probabilities*

---

## Description

Converts binary outcomes (0/1) to probability outcomes by replacing 1s with event failure probabilities. The output has the same structure as the input binary outcomes, but with probabilities instead of binary values.

## Usage

```
populate(binary_outcomes, event_probs)
```

## Arguments

`binary_outcomes`
: (Required) Data frame with binary (0/1) event states. Must have one column per basic event (with event names as column names) and optionally a `scenario` column. Each row represents one scenario.

`event_probs`
: (Required) Data frame with two columns: `event` (event names) and `probability` (failure probabilities in `[0, 1]`). Must contain probabilities for all events present in `binary_outcomes`.

## Details

This function transforms binary event states by replacing occurrences (1s) with their corresponding failure probabilities. For each event column, if the binary value is 1 (event occurred), it is replaced with the failure probability P(event) from `event_probs`. If the binary value is 0 (event did not occur), it remains 0. This creates a data frame suitable for probabilistic evaluation where only occurring events have probability values.

## Value

A tibble with the same structure as `binary_outcomes`, where:

- Values of 1 are replaced with P(event) from `event_probs`
- Values of 0 remain as 0

The `scenario` column (if present) is preserved unchanged.

### See Also

quantify for evaluating fault trees with binary or probabilistic inputs, quantify_prob for computing top event failure probability

### Examples

```
library(dplyr)
library(tidyfault)

data("db_outcomes_binary")
data("db_probs")

# Convert binary outcomes to probability outcomes
populate(db_outcomes_binary, db_probs)

# Compare: binary has 0/1, probability version has P(event) where binary was 1, 0 otherwise
db_outcomes_binary
populate(db_outcomes_binary, db_probs)
```

---

## quantify — *Evaluate the top event for many scenarios*

---

### Description

Unified wrapper function for evaluating fault trees. Evaluates whether the top event occurs (binary evaluation) or computes top event failure probability (probabilistic evaluation) based on the prob parameter.

### Usage

```
quantify(f, newdata, prob = FALSE, fast = TRUE)
```

### Arguments

| | |
|---|---|
| f | (Required) Function from formulate(), with one argument per basic event. Used to identify event names and to evaluate the fault tree. |
| newdata | (Required) Data for evaluation. When prob = FALSE, this represents binary scenarios (0/1 or TRUE/FALSE) for each basic event. When prob = TRUE, this represents failure probabilities (0 to 1) for each basic event. Can be a tibble/data frame with one column per basic event, a single vector/list, or (for prob = TRUE) a matrix. See details below. |
| prob | (Optional) Logical. If FALSE (default), performs binary evaluation. If TRUE, computes failure probability. |
| fast | (Optional) Logical. If TRUE (default), dispatches to optimized implementations (quantify_binary_fast() or quantify_prob_fast()). If FALSE, uses the legacy R implementations (quantify_binary() or quantify_prob()). |

## Details

This function provides a unified interface for both binary and probabilistic fault tree evaluation:

- **Binary evaluation** (`prob = FALSE`): evaluates whether the top event occurs given binary event states.
- **Probabilistic evaluation** (`prob = TRUE`): computes the top event failure probability from basic event failure probabilities.
- **Fast mode** (`fast = TRUE`): uses optimized paths for larger scenario batches while preserving output semantics.
- **Legacy mode** (`fast = FALSE`): uses the original pure-R implementations.

When `prob = FALSE`, `newdata` can be:

- A data frame with one column per basic event (names from `formalArgs(f)`)
- A single vector/list (unnamed = positional, or named)
- Values are logical (TRUE/FALSE) or numeric (0/1)

When `prob = TRUE`, `newdata` can be:

- A single vector or list (one scenario) with probabilities in `[0, 1]`
- A data frame or matrix with one row per scenario and columns for each basic event (names must match `formalArgs(f)`)
- Unnamed vectors are interpreted in `formalArgs(f)` order

## Value

A vector of outcomes. The type depends on `prob`:

- If `prob = FALSE`: Returns a logical vector (TRUE = system failure, FALSE = no failure). If `newdata` is a data frame with multiple rows, returns a vector with one outcome per row. If `newdata` is a single scenario, returns a single logical value.
- If `prob = TRUE`: Returns a numeric vector of top event failure probabilities. If `newdata` is a single scenario, returns a single numeric value. If `newdata` has multiple rows, returns a numeric vector with one probability per scenario.

Both modes return vectors in `predict()`-style format.

## See Also

`formulate` for creating the function, `calculate` for the full truth table, `quantify_binary`, `quantify_binary_fast`, `quantify_prob`, and `quantify_prob_fast`

## Examples

```
library(dplyr)
library(tidyfault)
data("fakenodes")
data("fakeedges")
```

```
f <- curate(nodes = fakenodes, edges = fakeedges) %>%
  equate() %>%
  formulate()

# Binary evaluation (default): single scenario
f %>% quantify(c(TRUE, TRUE, TRUE, FALSE))
f %>% quantify(c(TRUE, FALSE, TRUE, FALSE))

# Binary evaluation: multiple scenarios
scenarios_tbl <- tibble(
  A = c(1L, 0L, 1L),
  B = c(0L, 1L, 1L),
  C = c(1L, 0L, 0L),
  D = c(0L, 1L, 1L)
)
f %>% quantify(scenarios_tbl)

# Probabilistic evaluation: single scenario
f %>% quantify(c(0.1, 0.2, 0.05, 0.15), prob = TRUE)

# Legacy mode
f %>% quantify(c(0.1, 0.2, 0.05, 0.15), prob = TRUE, fast = FALSE)

# Probabilistic evaluation: multiple scenarios
probs_df <- as.data.frame(replicate(4, runif(100), simplify = FALSE))
names(probs_df) <- formalArgs(f)
f %>% quantify(probs_df, prob = TRUE)
```

---

`quantify_ci` *Uncertainty interval for the top-event probability*

---

## Description

Puts an uncertainty interval around the probability of the top event. We rarely know a failure rate exactly, so `quantify_ci()` draws `n` plausible values for each basic event from a Normal distribution centered on its estimate (standard deviation `cv` times the estimate, floored at zero), evaluates the top event for every draw, and reports the median and the central `level` interval. It is the uncertainty engine behind `quantify_if(ci = TRUE)` and `quantify_when(ci = TRUE)`.

## Usage

```
quantify_ci(
  f,
  data,
  n = 1000,
  cv = 0.2,
  time = NULL,
  level = 0.9,
  seed = NULL,
```

```
  draws = FALSE
)
```

## Arguments

| | |
|---|---|
| `f` | (Required) Function from `formulate()`, with one argument per basic event. |
| `data` | (Required) One or more rows of values, one column per basic event of `f`: a data frame or tibble, or a named numeric vector for a single row. With `time` (or a `time` column), the values are failure rates (events per hour, by default); without it, they are failure probabilities. An optional `scenario` column and an optional `time` column pass through to the result; other extra columns are ignored. |
| `n` | (Optional) The number of draws (simulated worlds) per row. Default `1000`. |
| `cv` | (Optional) The coefficient of variation: the standard deviation of each draw as a share of its value. Default `0.2` (+/- 20\ `0` for no uncertainty. |
| `time` | (Optional) One or more non-negative times, in the same unit as the rates (hours, the reliability-engineering convention; 8,760 hours is one year). If given, `data` holds failure rates, every row is evaluated at every time, and each drawn rate becomes the probability of failing by that time, `pexp(time, rate)`. Leave it `NULL` when `data` already has a `time` column (per-row times), or when `data` holds probabilities. |
| `level` | (Optional) The coverage of the interval, in `(0, 1)`. Default `0.90`: `lower` and `upper` are the 5th and 95th percentiles of the draws. |
| `seed` | (Optional) A single number passed to `set.seed()` before drawing, as in `simulate`, so the draws are reproducible. Default `NULL` uses the current random stream. |
| `draws` | (Optional) `FALSE` (default) returns one summary row per input row (and time). `TRUE` returns every draw instead. |

## Details

For each basic event and each draw, `quantify_ci()` draws one standard Normal value `z` and forms the noise factor `max(1 + cv * z, 0)`. Each value is multiplied by its noise factor, which is the same as drawing from `Normal(mean = value, sd = cv * value)` and flooring at zero. Drawn probabilities are capped at 1; drawn rates need no cap, because `pexp()` turns them into probabilities.

**Common random numbers.** The noise factors are drawn *once*, an `n` by events matrix, and every row of `data` (every scenario, every time) reuses them: draw 5 of "Fix A" and draw 5 of "Neither" share one noise factor per event. Each row still gets the right spread, but the differences between rows come only from the rows themselves, not from different random draws, which makes comparisons much sharper for the same `n`.

The noise is drawn event by event (all `n` draws for the first event column of `data`, then the next), in the order the events' columns appear in `data`, so the same seed and the same `data` give the same draws. All `n * nrow(data)` rows go through `quantify(f, prob = TRUE)` in one call.

## Value

A tibble. With `draws = FALSE`, one row per row of `data` (times each value of `time`, if given), in that order, with columns:

- scenario: if data has it (a factor stays a factor).
- time: when the values are rates.
- p_top: the median probability of the top event across the draws.
- lower, upper: the central level interval of the draws (quantile(), type 7).

With draws = TRUE, n rows per input row: scenario and time as above, sim (the draw, 1 to n) and p_top for that draw.

## See Also

quantify_when for scenarios over time, quantify_if for one-at-a-time cuts, quantify for the top event probability.

## Examples

```
library(dplyr)
library(tidyr)
library(tidyfault)

f <- curate(nodes = it_security_nodes, edges = it_security_edges) %>%
  equate() %>%
  formulate()

# One row of failure rates (events per hour), one column per basic event
rates <- it_security_outcomes_rates %>%
  select(event, lambda) %>%
  pivot_wider(names_from = event, values_from = lambda)

# The chance that data leaks within one year (8,760 hours), with a 90%
# interval from 1,000 plausible sets of failure rates, each +/- 20%
quantify_ci(f, rates, time = 8760, seed = 1)

# The same draws at three horizons: one, two and five years
quantify_ci(f, rates, time = c(1, 2, 5) * 8760, seed = 1)

# Every draw, for a histogram of the risk
quantify_ci(f, rates, time = 8760, n = 200, seed = 1, draws = TRUE)

# Probabilities instead of rates: leave out `time`
quantify_ci(f, it_security_probs, n = 200, seed = 1)
```

---

# quantify_if *Which fix buys the most*

---

## Description

Asks "which fix buys the most?". Cuts the failure rate (or probability) of each basic event in turn by the same share, keeps every other event as it is, and measures how much the probability of the top event drops. It is a one-at-a-time sensitivity analysis of the top event.

## Usage

```
quantify_if(
  f,
  data,
  cut = 0.05,
  time = NULL,
  events = NULL,
  ci = FALSE,
  n = 1000,
  cv = 0.2,
  level = 0.9,
  seed = NULL
)
```

## Arguments

| | |
|---|---|
| `f` | (Required) Function from `formulate()`, with one argument per basic event. |
| `data` | (Required) One row of values, one column per basic event of `f` (extra columns are ignored): a one-row data frame or tibble, or a named numeric vector. With `time`, the values are failure rates (events per hour, by default); without it, they are failure probabilities. |
| `cut` | (Optional) The share to cut each value by, in `(0, 1]`: each event's value is multiplied by `1 - cut`. Default `0.05` (a 5\ cut). `cut = 1` removes the event entirely. |
| `time` | (Optional) A single positive number, in the same unit as the rates (hours, the reliability-engineering convention; 8,760 hours is one year). If given, `data` holds failure rates, and each (cut) rate becomes the probability of failing by `time`, `pexp(time, rate)`. If `NULL` (default), `data` holds probabilities and is used as is. |
| `events` | (Optional) Character vector naming the basic events to cut. Default `NULL` cuts every basic event of `f`, in `formalArgs(f)` order. |
| `ci` | (Optional) `FALSE` (default) returns the exact values only. `TRUE` adds an uncertainty interval per row from `quantify_ci`. |
| `n`, `cv`, `level`, `seed` | (Optional) Passed to `quantify_ci` when `ci = TRUE`: the number of draws (default `1000`), the coefficient of variation of each value (default `0.2`), the coverage of the interval (default `0.90`) and a seed for `set.seed()` (default `NULL`, the current random stream). Ignored when `ci = FALSE`. |

## Details

The baseline and every what-if go through `quantify(f, prob = TRUE)` in one call, so the exact (truth-table) top event probability is used throughout. The cut is applied to the value in `data` before any conversion: with `time`, the rate is cut and then turned into a probability, which is what a fix to a component does.

With `ci = TRUE`, the baseline and every what-if share the same random draws (common random numbers), so the intervals can be compared row to row.

## Value

A tibble with one row per cut event, sorted by `pct_change` from the biggest drop to the smallest (ties keep the order of `events`), with columns:

- `event`: the basic event that was cut.
- `p_top`: the probability of the top event after the cut.
- `baseline`: the probability of the top event with no cut (the same in every row).
- `change`: `p_top - baseline`.
- `pct_change`: the percent change from the baseline, `100 * (p_top / baseline - 1)`.
- `lower`, `upper`: only with `ci = TRUE`, the central `level` interval of the top event's probability after the cut, from `quantify_ci`. `p_top` stays the exact value.

## See Also

`quantify` for the top event probability, `quantify_when` for scenarios over time, `quantify_ci` for uncertainty intervals.

## Examples

```
library(dplyr)
library(tidyr)
library(tidyfault)

f <- curate(nodes = it_security_nodes, edges = it_security_edges) %>%
  equate() %>%
  formulate()

# One row of failure rates (events per hour), one column per basic event
rates <- it_security_outcomes_rates %>%
  select(event, lambda) %>%
  pivot_wider(names_from = event, values_from = lambda)

# Cut each failure rate by 5%, one at a time: which fix lowers the chance
# that data leaks within one year (8,760 hours) the most?
quantify_if(f, rates, cut = 0.05, time = 8760)

# Only two candidate fixes, each removing 90% of the failures, with 90%
# uncertainty intervals (each rate +/- 20%)
quantify_if(f, rates, cut = 0.9, time = 8760, events = c("MN", "PO"),
            ci = TRUE, n = 200, seed = 1)

# With probabilities instead of rates, leave out `time`
quantify_if(f, it_security_probs)
```

`quantify_prob` *Top-event probability from event probabilities*

## Description

Computes the top event failure probability from a fault tree when given basic event failure probabilities. Assumes independent basic events; uses the full truth table to sum probabilities over all failure combinations.

## Usage

```
quantify_prob(f, newdata, truth_table = NULL)
```

## Arguments

| | |
|---|---|
| `f` | (Required) Function outputted by `formulate()`, with one argument per basic event. Used to identify event names and, if `truth_table` is not supplied, to generate the truth table via `calculate(f)`. |
| `newdata` | (Required) Failure probabilities (0 to 1) for each basic event. Can be a single vector or list (one scenario), or a data frame or matrix with one row per scenario and columns for each basic event (names must match `formalArgs(f)`). Unnamed vectors are interpreted in `formalArgs(f)` order. Values should be in `[0, 1]`, where each value represents the probability that the corresponding basic event occurs. |
| `truth_table` | (Optional) Data frame with one column per basic event (0/1) and an `outcome` column (1 = system failure, 0 = no failure). If provided, the truth table is not recomputed (useful when `calculate()` was already run). |

## Details

The top event probability is computed exactly via the complete truth table: for each of the 2^n combinations of basic event states, the probability of that combination is the product of `newdata[e]` for events that occur (1) and `1 - newdata[e]` for events that do not (0). The function sums these probabilities over all combinations where `outcome == 1`. This is exact for independent basic events but has complexity O(2^n); for trees with many basic events, consider using minimal cut set approximations elsewhere. When `newdata` has multiple rows, the truth table is computed once and all scenarios are evaluated in one pass.

## Value

A numeric vector of top-event failure probabilities. If `newdata` is a single scenario (vector/list or single-row data frame), returns a single numeric value. If `newdata` has multiple rows (one per scenario), returns a numeric vector with one probability per scenario (same order as rows of `newdata`). Similar to R's `predict()` function style.

## See Also

formulate for creating the function from a boolean equation, calculate for generating the truth table, quantify for evaluating whether the top event occurs given binary (0/1) basic event states. The result matches the FaultTree package when the same tree is built and `ftree.calc(DF, use.bdd = TRUE)` is used (BDD accounts for repeated basic events; gate-by-gate with `use.bdd = FALSE` does not).

## Examples

```
library(dplyr)
library(tidyfault)
data("fakenodes")
data("fakeedges")

f <- curate(nodes = fakenodes, edges = fakeedges) %>%
  equate() %>%
  formulate()
# Pipe-friendly: probability vector in event order (e.g. A, B, C, D)
curate(nodes = fakenodes, edges = fakeedges) %>%
  equate() %>%
  formulate() %>%
  quantify_prob(c(0.1, 0.2, 0.05, 0.15))

# Named probs (names must match event names in the tree)
probs <- setNames(c(0.1, 0.2, 0.05, 0.15), formalArgs(f))
f %>% quantify_prob(probs)

# Reuse an existing truth table to avoid recomputing
tt <- calculate(f)
f %>% quantify_prob(probs, truth_table = tt)

# Many scenarios at once (vectorized): pass a data frame, one row per scenario
probs_df <- as.data.frame(replicate(4, runif(1000), simplify = FALSE))
names(probs_df) <- formalArgs(f)
f %>% quantify_prob(probs_df, truth_table = tt)  # length-1000 vector
```

---

# quantify_prob_fast *Top-event probability in one vectorized pass*

---

## Description

Computes top-event failure probability using the same exact truth-table approach as `quantify_prob()`, but with a compiled inner loop for faster evaluation across many scenarios.

## Usage

```
quantify_prob_fast(f, newdata, truth_table = NULL)
```

### Arguments

| | |
|---|---|
| `f` | (Required) Function outputted by `formulate()`, with one argument per basic event. |
| `newdata` | (Required) Failure probabilities in `[0, 1]`. Can be a single vector/list, or a data frame/matrix with one row per scenario. |
| `truth_table` | (Optional) Data frame with one column per basic event and an `outcome` column. If `NULL`, computed via `calculate(f)`. |

### Value

Numeric vector of top-event failure probabilities (length 1 for a single scenario).

### Examples

```
data(db_nodes, db_edges, db_outcomes_prob, package = "tidyfault")
f <- formulate(equate(curate(db_nodes, db_edges)))

# One named probability per basic event, in the order f expects
probs <- stats::setNames(db_outcomes_prob$probability, db_outcomes_prob$event)
quantify_prob_fast(f, probs[methods::formalArgs(f)])
```

---

## `quantify_when` *Top-event probability over time*

---

### Description

Asks "when does it fail, and what if we fix something?". Builds what-if scenarios from one row of failure rates, each scenario multiplying the rates of some basic events by a factor (for example `0.1` for a fix that removes 90\ top event over time for every scenario, with an uncertainty interval.

### Usage

```
quantify_when(
  f,
  data,
  ...,
  time = seq(0, 87600, by = 4380),
  baseline = "Neither",
  ci = TRUE,
  n = 1000,
  cv = 0.2,
  level = 0.9,
  seed = NULL
)
```

### Arguments

- `f` — (Required) Function from `formulate()`, with one argument per basic event.
- `data` — (Required) One row of failure rates (events per hour, by default), one column per basic event of `f`: a one-row data frame or tibble, or a named numeric vector. Extra columns are ignored.
- `...` — (Required) One or more named scenarios. Each is a named numeric vector of multipliers, `c(event = factor)`, such as `"Fix A only" = c(MN = 0.1)`. Events a scenario does not name keep their rate. Names with spaces need quotes or backticks.
- `time` — (Optional) The times to evaluate, in the same unit as the rates: one or more finite, non-negative numbers. The default, `seq(0, 87600, by = 4380)`, is 0 to 10 years in half-year steps, in hours (the reliability-engineering convention; 8,760 hours is one year).
- `baseline` — (Optional) The name of the unchanged scenario placed first. Default `"Neither"`. Use `NULL` to leave it out.
- `ci` — (Optional) `TRUE` (default) adds an uncertainty interval from `quantify_ci`; `FALSE` returns the exact point estimate only, and ignores `n`, `cv`, `level` and `seed`.
- `n, cv, level, seed` — (Optional) Passed to `quantify_ci` when `ci = TRUE`: the number of draws (default `1000`), the coefficient of variation of each rate (default `0.2`), the coverage of the interval (default `0.90`) and a seed for `set.seed()` (default `NULL`, the current random stream).

### Details

Multipliers are applied to the rates; each scenario's rates then become probabilities of failing by each time, `pexp(time, rate)`, and go through `quantify(f, prob = TRUE)` in one call. With `ci = TRUE`, every scenario and every time shares the *same* random draws (common random numbers; see `quantify_ci`), so the gaps between the scenarios come from the fixes, not from noise.

### Value

A tibble with one row per scenario and time (the baseline first, then the scenarios in the order given; times in the order given), with columns:

- `scenario`: a factor whose levels follow that order.
- `time`: the time, in the unit of the rates.
- `p_top`: the probability that the top event has happened by `time`. With `ci = TRUE`, the median across the draws; with `ci = FALSE`, the exact value at the given rates.
- `lower`, `upper`: the central `level` interval of the draws. Only with `ci = TRUE`; with `ci = FALSE` these columns are left out.
- `reliability`: `1 - p_top`, the chance the system has not failed by `time`.

### See Also

`quantify_ci` for the uncertainty draws, `quantify_if` for one-at-a-time cuts, `quantify` for the top event probability.

## Examples

```
library(dplyr)
library(tidyr)
library(tidyfault)

f <- curate(nodes = it_security_nodes, edges = it_security_edges) %>%
  equate() %>%
  formulate()

# One row of failure rates (events per hour), one column per basic event
rates <- it_security_outcomes_rates %>%
  select(event, lambda) %>%
  pivot_wider(names_from = event, values_from = lambda)

# Four scenarios: no fix, add multi-factor authentication (MN), patch on
# time (PO), or both. Each fix removes 90% of that component's failures.
# The chance that data leaks within 0 to 10 years, with 90% intervals.
over_time <- quantify_when(
  f, rates,
  "Fix A only" = c(MN = 0.1),
  "Fix B only" = c(PO = 0.1),
  "Fix A and B" = c(MN = 0.1, PO = 0.1),
  n = 200, seed = 1
)
over_time %>% filter(time == 8760)   # one year

# Point estimates only, at one, two and five years
quantify_when(f, rates, "Fix A only" = c(MN = 0.1),
              time = c(1, 2, 5) * 8760, ci = FALSE)
```

---

security_edges security_edges *security breach fault tree edge dataset*

---

## Description

An example dataset of edges (connections) in a security breach fault tree.

## Usage

```
security_edges
```

## Format

security_edges**:**

A data frame with 11 rows and 2 columns (the top event T, id 1, links only to its one gate G0, id 12, which links to gates G2, G3 and G4):

**from** Unique id of the source/from node from which edge originates.

**to** Unique id of the destination/to node that edge connects to.

### See Also

security_nodes for the corresponding nodes dataset

### Examples

```
data("security_edges")
head(security_edges)
dim(security_edges)
```

---

security_nodes security_nodes *security breach fault tree nodes dataset*

---

### Description

An example dataset of nodes in a security breach fault tree. This fault tree models scenarios where a security breach occurs due to external attacks, internal threats, vulnerabilities, and authentication failures.

### Usage

```
security_nodes
```

### Format

security_nodes**:**

A data frame with 12 rows and 3 columns:

**id** Unique identifier (1 to 12) for each node.

**event** Name of event. "T" means top event (Security breach detected). "G0", "G2", "G3", "G4" mean gates; "G0" (id 12) is the top event's one gate. "VE" = Vulnerability exists, "ES" = Exploit successful, "PH" = Phishing, "MW" = Malware, "UA" = Unauthorized access, "WP" = Weak password, "N2F" = No 2FA.

**type** factor classification as "top", "and", "or", or "not" (meaning "not" a gate).

### Details

The fault tree structure:

- Top event: Security breach detected. The top event is not a gate: its one child is G0.
- G0 (OR): G2 OR G3 OR G4 (any attack path is a breach)
- G2 (AND): Vulnerability exists AND Exploit successful
- G3 (OR): Phishing OR Malware OR Unauthorized access
- G4 (AND): Weak password AND No 2FA

Minimal cut sets: MW, PH, UA, ES*VE, N2F*WP.

### See Also

security_edges for the corresponding edges dataset

### Examples

```
data("security_nodes")
head(security_nodes)
dim(security_nodes)
```

---

## security_outcomes_binary

security_outcomes_binary *security breach fault tree binary scenarios*

---

### Description

Ten example scenarios for the basic events of the security breach fault tree, coded 1 if the event occurred and 0 if not. Use it with quantify() to evaluate whether the top event occurs under each scenario, or with populate().

### Usage

```
security_outcomes_binary
```

### Format

security_outcomes_binary**:**

A tibble with 10 rows and 8 columns:

**scenario** Scenario number (1 to 10).

**VE, ES, PH, MW, UA, WP, N2F** One 0/1 column per basic event; 1 means the event occurred.

### See Also

security_probs, security_outcomes_prob, quantify(), populate()

### Examples

```
data("security_outcomes_binary")
head(security_outcomes_binary)
```

security_outcomes_prob

security_outcomes_prob *security breach fault tree outcome probabilities*

## Description

Example failure probabilities for the basic events of the security breach fault tree, with one row per event. It holds the same values as security_probs.

## Usage

```
security_outcomes_prob
```

## Format

security_outcomes_prob**:**

A tibble with 7 rows and 2 columns:

**event** Name of the basic event: "VE" = Vulnerability exists, "ES" = Exploit successful, "PH" = Phishing, "MW" = Malware, "UA" = Unauthorized access, "WP" = Weak password, "N2F" = No 2FA.

**probability** Probability that the event occurs, between 0 and 1.

## See Also

security_probs, security_outcomes_binary, security_outcomes_rates

## Examples

```
data("security_outcomes_prob")
security_outcomes_prob
```

security_outcomes_rates

security_outcomes_rates *security breach fault tree failure rates*

## Description

Example exponential failure rates for the basic events of the security breach fault tree, expressed per day.

## Usage

```
security_outcomes_rates
```

### Format

`security_outcomes_rates`:

A tibble with 7 rows and 3 columns:

**event** Name of the basic event: `"VE"` = Vulnerability exists, `"ES"` = Exploit successful, `"PH"` = Phishing, `"MW"` = Malware, `"UA"` = Unauthorized access, `"WP"` = Weak password, `"N2F"` = No 2FA.

**lambda** Failure rate (failures per day).

**time_unit** Unit of time for `lambda`: `"days"`.

### See Also

security_probs, security_outcomes_prob

### Examples

```
data("security_outcomes_rates")
security_outcomes_rates
```

---

## security_probs `security_probs` *security breach fault tree event probabilities*

---

### Description

Example failure probabilities for the basic events of the security breach fault tree (`security_nodes`, `security_edges`). Pass it as `event_probs` to `populate()`.

### Usage

```
security_probs
```

### Format

`security_probs`:

A tibble with 7 rows and 2 columns:

**event** Name of the basic event: `"VE"` = Vulnerability exists, `"ES"` = Exploit successful, `"PH"` = Phishing, `"MW"` = Malware, `"UA"` = Unauthorized access, `"WP"` = Weak password, `"N2F"` = No 2FA.

**probability** Probability that the event occurs, between 0 and 1.

### See Also

security_outcomes_prob, security_outcomes_binary, `populate()`

## Examples

```
data("security_probs")
security_probs
```

---

`simulate` *Simulate a random fault tree*

---

## Description

This function *simulates* a fault tree structure with nodes, edges, and basic event probabilities. Generates a valid fault tree with one top event, one gate under it, multiple gates (AND/OR), and basic events, ensuring each gate has at least two input events.

## Usage

```
simulate(n_gates = 3L, n_basic = 8L, p_range = c(0.01, 0.2), seed = NULL)
```

## Arguments

| | |
|---|---|
| `n_gates` | (Optional) Integer specifying the desired number of internal gates. Default is `3L`. The actual number of gates may be reduced if `n_basic` is insufficient to provide at least two basic events per gate. |
| `n_basic` | (Optional) Integer specifying the number of basic events (leaf nodes) in the fault tree. Default is `8L`. Must be at least 1. |
| `p_range` | (Optional) Numeric vector of length 2 specifying the minimum and maximum probability values for basic events. Default is `c(0.01, 0.2)`. Must satisfy `0 < p_range[1] < p_range[2] < 1`. |
| `seed` | (Optional) Integer seed for random number generation to ensure reproducibility. Default is `NULL` (no seed set). |

## Details

This function generates a valid fault tree structure suitable for use with other `tidyfault` functions:

- **Gate constraint**: Each gate (AND/OR) is guaranteed to have at least two input events. If `n_basic < 2 * n_gates`, the number of gates is automatically reduced to `floor(n_basic / 2)`.
- **Degenerate case**: If `n_basic < 2`, no internal gates are created; the top event connects to `G0`, which connects directly to all basic events.
- **Gate types**: Gate types (AND/OR) are randomly assigned using `sample()`.
- **Event allocation**: Basic events are allocated to gates such that each gate receives at least two events, with any remaining events randomly distributed.
- **Top event**: The top event is not a gate, so it has exactly one child. With two or more gates (or none) that child is an added OR gate `"G0"` (any branch fails the system); with one gate it is that gate. `G0` adds no random draw, so a seed reproduces the same gates, events and probabilities.

- **Event naming**: Top event is named "T", the added OR gate "G0", other gates are named "G1", "G2", etc., and basic events are named "E1", "E2", etc.

The generated fault tree can be used directly with curate(), equate(), formulate(), and other tidyfault workflow functions.

## Value

A list with three components:

- nodes: Data frame with columns id, event, and type. Contains one top event (type "top"), n_gates_eff internal gates (type "and" or "or"), and n_basic basic events (type "not"), plus, unless n_gates_eff == 1, the OR gate G0 with the last id. The type column is a factor with levels c("top", "and", "or", "not").
- edges: Data frame with columns from and to, representing connections between nodes. The top event connects to its one gate (G0, which connects to all other gates, or the only gate G1), and gates connect to their assigned basic events.
- prob: Data frame with columns event and probability, containing probability values for each basic event, sampled uniformly from p_range.

## See Also

curate for processing the simulated nodes and edges, illustrate for visualizing the generated fault tree structure

## Examples

```
# Load dependencies
library(tidyfault)

# Simulate a fault tree with default parameters
sim <- simulate(n_gates = 4L, n_basic = 10L, seed = 12345)
sim$nodes
sim$edges
sim$prob

# Use the simulated tree in the tidyfault workflow
gates <- curate(nodes = sim$nodes, edges = sim$edges)
equation <- equate(gates)
formula <- formulate(equation)
cutsets <- concentrate(gates)

# Simulate a larger tree
sim_large <- simulate(n_gates = 10L, n_basic = 25L,
                      p_range = c(0.05, 0.3), seed = 42)
```

tabulate *Coverage of each minimal cut set*

## Description

This function *tabulates* a `data.frame` of `N` minimum cutsets in your fault tree, describing each `mincut` set, a `query` used to filter a table of all sets, the number of `cutsets` that include that `mincut`, the total number of cutsets leading to `failure`, and the `coverage` (percentage of `cutsets` covered out of total failures) for that `mincut`.

## Usage

```
tabulate(
  data,
  formula,
  method = c("mocus_rcpp", "mocus_r", "mocus_original"),
  query = FALSE
)
```

## Arguments

| | |
|---|---|
| `data` | (Required) Output from `concentrate()` function. Expects a character vector of minimum cutset expressions (e.g., `"A * B"`). |
| `formula` | (Required) Function output from `formulate()`. Used to generate the truth table for calculating coverage statistics. |
| `method` | (Optional) Character string specifying the method used in `concentrate()`. Default is `"mocus_rcpp"`. Supported values are `"mocus_rcpp"`, `"mocus_r"`, and `"mocus_original"`. |
| `query` | (Optional) Logical - Whether to include the query column in the output. Default is `FALSE`. |

## Details

This function analyzes minimum cutsets to quantify their importance in system failure:

- **Truth Table Generation**: Uses `formula` to generate a complete truth table via `calculate()`.
- **Cutset Parsing**: Splits each minimum cutset expression into individual events and determines their required states (1 for occurrence, 0 for non-occurrence)
- **Query Construction**: Creates filter expressions that identify truth table rows matching each cutset's event combination
- **Coverage Calculation**: Counts how many failure cases (rows with `outcome == 1`) are covered by each minimum cutset, then calculates coverage as a proportion

## Value

A `data.frame` (tibble) with one row per minimum cutset, containing:

- `mincut`: Character string representing the minimum cutset as a boolean expression (e.g., `"A * B"` for events A AND B)
- `query`: Character string containing a filter expression that can be used to identify truth table rows matching this cutset (e.g., `"filter(A == 1, B == 1, outcome == 1)"`) (only included if `query = TRUE`)
- `cutsets`: Integer count of truth table rows (cutsets) that match this minimum cutset and result in system failure
- `failures`: Integer count of total truth table rows that result in system failure (same for all rows)
- `coverage`: Numeric value (0 to 1) representing the proportion of failure cases covered by this minimum cutset. Calculated as `cutsets / failures`.

## See Also

`concentrate` for generating minimum cutsets, `formulate` for creating the function used in truth table generation, `calculate` for generating truth tables

## Examples

```
# Load dependencies
library(dplyr)
library(tidyfault)

# Load example data into our environment
data("fakenodes")
data("fakeedges")

formula <- curate(nodes = fakenodes, edges = fakeedges) %>%
   equate() %>%
   formulate()
curate(nodes = fakenodes, edges = fakeedges) %>%
   concentrate(method = "mocus_rcpp") %>%
   tabulate(formula = formula, method = "mocus_rcpp")
```

# Index


∗ **boolean**
calculate, 9
formulate, 21
mocus, 34
∗ **cutset**
concentrate, 10
mocus, 34
tabulate, 59
∗ **datasets**
ai_edges, 3
ai_nodes, 4
ai_outcomes_binary, 5
ai_outcomes_prob, 5
ai_outcomes_rates, 6
ai_probs, 7
breach_edges, 7
breach_nodes, 8
db_edges, 13
db_nodes, 14
db_outcomes_binary, 15
db_outcomes_prob, 16
db_outcomes_rates, 17
db_probs, 17
fakeedges, 19
fakenodes, 20
it_security_data, 30
it_security_edges, 31
it_security_nodes, 31
it_security_outcomes_rates, 33
it_security_probs, 33
security_edges, 52
security_nodes, 53
security_outcomes_binary, 54
security_outcomes_prob, 55
security_outcomes_rates, 55
security_probs, 56
∗ **equation**
calculate, 9
equate, 18
formulate, 21
∗ **evaluation**
quantify, 41
∗ **fault**
calculate, 9
concentrate, 10
curate, 12
equate, 18
formulate, 21
gate, 22
gate_and, 23
gate_or, 24
gate_top, 25
get_gate, 26
mocus, 34
plot.tidyfault_tree, 38
populate, 40
quantify, 41
quantify_ci, 43
quantify_if, 45
quantify_prob, 48
quantify_prob_fast, 49
quantify_when, 50
simulate, 57
∗ **formula**
formulate, 21
∗ **gate**
gate, 22
gate_and, 23
gate_or, 24
gate_top, 25
get_gate, 26
∗ **ggplot**
illustrate, 28
plot.tidyfault_tree, 38
∗ **logic**
calculate, 9
∗ **maker**
gate, 22

gate_and, 23
gate_or, 24
gate_top, 25
get_gate, 26
∗ **minimization**
concentrate, 10
∗ **minimum**
concentrate, 10
tabulate, 59
∗ **network**
illustrate, 28
∗ **polygon**
gate, 22
gate_and, 23
gate_or, 24
gate_top, 25
get_gate, 26
∗ **probability**
populate, 40
quantify_prob, 48
quantify_prob_fast, 49
∗ **quantification**
quantify, 41
quantify_prob, 48
quantify_prob_fast, 49
∗ **reliability**
quantify_when, 50
∗ **scenarios**
quantify_when, 50
∗ **scenario**
populate, 40
∗ **sensitivity**
quantify_if, 45
∗ **simulation**
quantify_ci, 43
simulate, 57
∗ **tree**
calculate, 9
concentrate, 10
curate, 12
equate, 18
formulate, 21
gate, 22
gate_and, 23
gate_or, 24
gate_top, 25
get_gate, 26
mocus, 34
plot.tidyfault_tree, 38
populate, 40
quantify, 41
quantify_ci, 43
quantify_if, 45
quantify_prob, 48
quantify_prob_fast, 49
quantify_when, 50
simulate, 57
∗ **uncertainty**
quantify_ci, 43
∗ **visualize**
illustrate, 28
plot.tidyfault_tree, 38

ai_edges, 3, *4*
ai_nodes, *3*, 4
ai_outcomes_binary, 5, *6, 7*
ai_outcomes_prob, 5, *5–7*
ai_outcomes_rates, 6, *6*
ai_probs, *5, 6*, 7

breach_edges, 7, *8*
breach_nodes, *7*, 8, *8*

calculate, 9, *22*, *42*, *49*, *60*
concentrate, 10, *60*
concentrate(), *35*
curate, *11*, 12, *19*, *58*
curate(), *35*

db_edges, 13, *15*
db_nodes, 14, *14*
db_outcomes_binary, 15, *16*, *18*
db_outcomes_prob, *15*, 16, *17, 18*
db_outcomes_rates, *16*, 17
db_probs, *15, 16*, 17, *17*

equate, *13*, 18, *22*

fakeedges, 19
fakenodes, 20
formulate, *9*, *19*, 21, *42*, *49*, *60*

gate, 22, *29*
gate_and, 23, *23*, *27*
gate_or, *23*, 24, *27*
gate_top, *23*, 25, *27*
get_gate, *23*, 26

illustrate, *13*, 28, *39*, *58*
it_security_data, 30, *33, 34*
it_security_edges, 31, *32*
it_security_nodes, *30*, 31, *31*
it_security_outcomes_rates, 33, *34*
it_security_probs, 33, *33*

layout_as_tree, *29*

mocus, *11*, 34
mocus_cpp *(*mocus_rcpp*)*, 37
mocus_r, *11*, 36
mocus_r(), *35*
mocus_rcpp, *11*, 37
mocus_rcpp(), *35*

plot, *38*
plot *(*plot.tidyfault_tree*)*, 38
plot.tidyfault_tree, 38
populate, 40
populate(), *5*, *7*, *15*, *17, 18*, *54*, *56*

quantify, *30*, 41, *41*, *45*, *47*, *49*, *51*
quantify(), *5*, *15*, *33, 34*, *54*
quantify_binary, *42*
quantify_binary_fast, *42*
quantify_ci, 43, *46, 47*, *51*
quantify_if, 45, *45*, *51*
quantify_prob, *41, 42*, 48
quantify_prob_fast, *42*, 49
quantify_when, *45*, *47*, 50

security_edges, 52, *54*
security_nodes, 53, *53*
security_outcomes_binary, 54, *55, 56*
security_outcomes_prob, *54*, 55, *56*
security_outcomes_rates, 55, *55*
security_probs, *54, 55*, 56, *56*
simulate, *44*, 57

tabulate, *11*, 59
tbl_graph, *29*