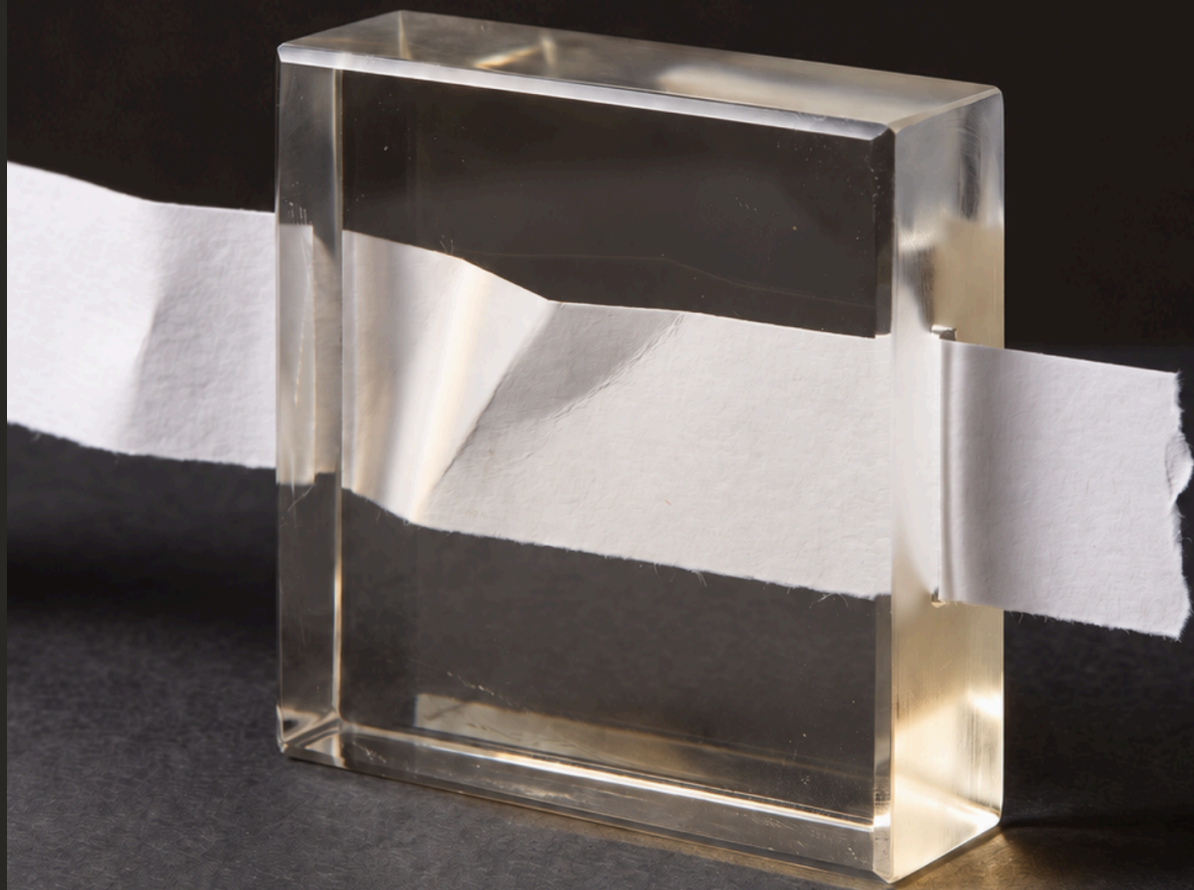


Constructions

The Craft of Formalization



ARXO

ENGLISH EDITION 2026.10

Constructions

The Craft of Formalization

Rifat Jumagulov

Prologue, five parts, eighteen chapters and epilogue

FIRST ENGLISH EDITION / 2026.10

Contents

P	The Text on the Table	6
---	-----------------------	---

PART I

Reading the First Model

01	What This Record Is About	11
02	From Two Premises to a Conclusion	20
03	Established, Refuted, Conflicting, Unknown	29
04	Why We Got This Answer	39

PART II

State the Conditions

05	First Check That the Object Exists	49
06	Derive a Consequence and Check a Requirement	60
07	Do We Have All the Information?	69

Contents

PART III

Examining the Grounds of the Answer

08	When Support Is Set Aside	80
09	The Answer That Needs a Human	90
10	What Is Permitted, What Is Required, What Happened	99
11	Who Reported It and What It Confirms	109

PART IV

Back to the Source

12	Where Does the Source Say That?	120
13	As of Which Date?	129
14	One Word, Several Definitions	138

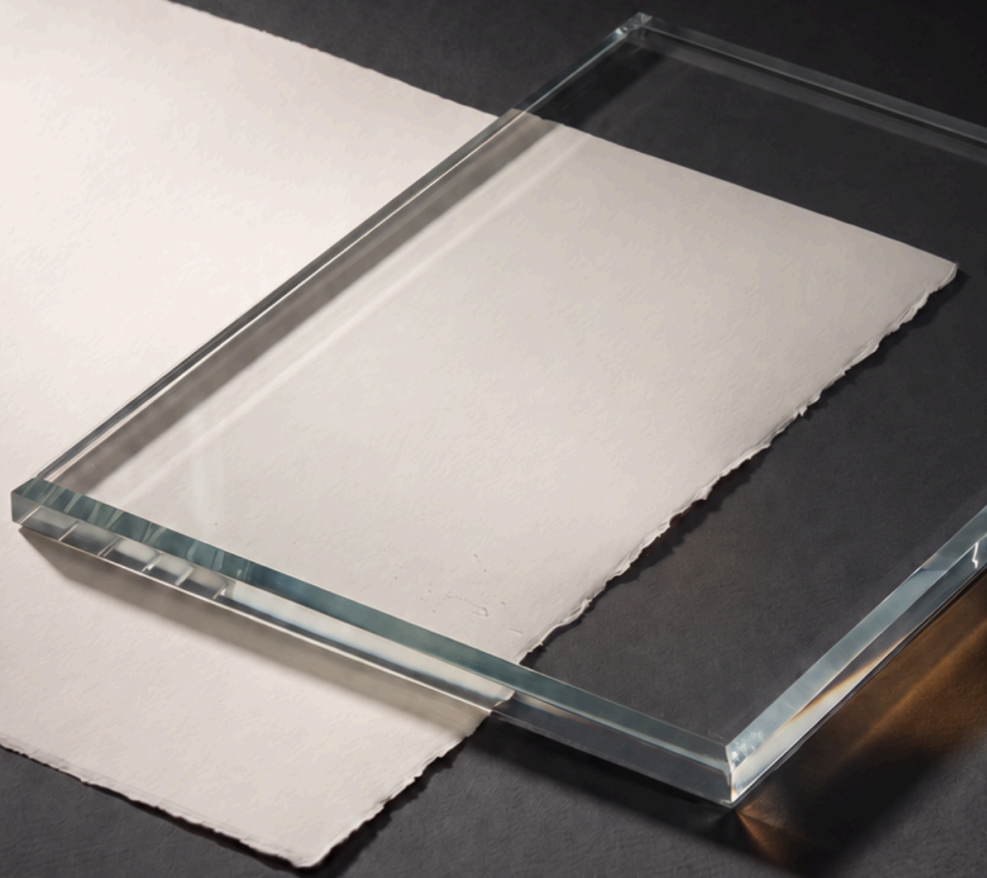
PART V

Assemble and Hand Over

15	A Number With Its Unit	149
16	Reusing a Package	156
17	A Check That Tells Models Apart	163
18	A Model You Can Hand Over	171
E	Now Show the Grounds	180

PROLOGUE

The Text on the Table



PROLOGUE

The Text on the Table

On the table lies a text. You may know it well: the rules you have long played by, the instructions you work by, the theorem you remember from school. Beside it lies a model built from it. You can ask it a question, change its initial conditions, and check what follows from them. But first we need to see exactly what it has kept from the text — and learn to read it ourselves.

The rules of a game contain both an allowed move and a piece of advice to the player. A theorem's exposition contains a conclusion and the conditions under which it is true. In a story, the hero may be sure of what the story itself does not confirm. In ordinary reading we tell such things apart without thinking. In a model these differences must be expressed explicitly: the answer depends on them.

This book teaches you to read such a model, change it, and finally write your own. You need no prior knowledge of the language, and no other books. First a statement and a question. Then the conclusion and its boundaries. Then sources, time, and interpretation. At the end — a model you can hand to another person.

One sentence

Let us start with a sentence from a general chemistry textbook, OpenStax Chemistry 2e, section 14.1. Water can give a proton to another particle, and it can accept one. The textbook calls particles able to do both amphiprotic.

That is all the chemistry we need for now. You need not remember what a proton is, only that it passes from one particle to another; there is no need to know any formulas. What matters is seeing three things in the sentence: what it is about, what it reports, and what follows from what is reported. Water. Gives a proton, accepts a proton. So it is amphiprotic.

The first four chapters will stay with this one sentence. We will write it down and ask questions about it. Then we will remove one piece of information and see what happens to the answer, add a contradictory one and see how the model keeps it, and read the answer's explanation down to the line in the textbook. After that the subject will change, but the way of reasoning will stay: a triangle, a drawing, a mushroom, a parable, a standard, a unit of measurement, acid again. Each new step will build on the previous ones, and each time a real text will stand beside it.

Six questions for the text

Pick a small text you would like to look at closely. A familiar rule, some instructions, a reference definition, a short story. Choose something you can read in full and discuss in detail. Save it and open a sheet for notes beside it.

As we read, we will return to six questions. Answer them now in ordinary words, then check your answers against the chapters where each question gets its formal expression.

1. **What is the rule here?** Which phrases report information, which link conditions to a conclusion, and which convey someone else's opinion or expectation? Chapters 2 and 5–8.
2. **What is the text silent about?** Which questions in it have no answer? How can we preserve that absence without turning it into a negation? Chapters 3 and 7.
3. **Who decides?** What follows from the recorded conditions, and what needs a separate decision by a human or an institution? Chapters 9 and 10.
4. **What counts as a unit of the source?** Does the text have articles, axioms, paragraphs? If you divide it yourself, why this way? Chapter 12.
5. **Which version of the text is before us?** What changed in other editions, and how can you tell which one your question needs? Chapters 13 and 14.
6. **How can you hand the work over?** What will another person need to repeat the check, trace the reasoning, and explain what they disagree with? Chapters 16–18.

Your answers can be provisional until you have met the language. They are landmarks. After each part, return to the notes: some answers will grow more precise, others will need revision.

How we will work

The material will be ready-made models from the Arxo corpus together with their sources and checks. A model in the book is always cited verbatim, with the file's address; when an excerpt is shown, that is stated explicitly, and the full file is available through a link. The examples refer to a pinned historical version. The reader companion supplies the selected models, recompiles them with npm CLI 0.2.4, and records the adaptations needed for this edition. All printed commands and 383 selected scenarios have passed. The companion links to Syzdyq's handbook without distributing its text; its source references are bibliographic rather than a local text snapshot. These checks do not establish equivalence to the retired runtime or replace independent reader feedback.

There are two ways to read. The reader's route: predict the answer before it is printed, read the explanation after. The practical route: repeat the checks using the reader companion. Install the CLI below, then unpack the [companion](#) and open a terminal in its `reader/` directory. The companion contains the models, dependencies, and checks; npm supplies the executable. Everything needed to understand the model is explained in the chapters.

Install the command-line tool

Install a current Node.js LTS release, including npm, from [nodejs.org](#). The CLI requires Node.js 20 or newer. The published CLI supports macOS on Apple Silicon (ARM64) and Linux on x86-64. This edition's execution checks were run on macOS ARM64. Install the version checked for this edition and confirm it is available:

```
node --version
npm --version
npm install -g @arxo/cli@0.2.4
law version --json
```

The last command should report `"tool":"0.2.4"`. Keep that version for these examples; a later CLI may require different generated models or locks. If the shell cannot find `law`, add npm's global bin directory to your `PATH` or use the alternative below. For a permissions error, follow npm's [user-owned installation guidance](#).

You can also run the same version without a global installation:

```
npx --yes --package @arxo/cli@0.2.4 law version --json
```

For a direct CLI command below, replace its initial `law` with `npx --yes --package @arxo/cli@0.2.4 law` if you choose this route. The question-preparation and receipt helpers additionally need Python 3 and `law` on `PATH`; the global installation above provides that command. No npm model package is needed: the companion supplies the book's pinned inputs. Follow its README, and start each chapter in `reader/`. Keep commands that share shell variables in the same terminal session.

One warning about what this book does not do. It does not tell how the models were made, or where their authors went wrong. Before you lies the result, with checks for the properties we examine. Where a model covers only part of the text, that is stated; where the model's author made a choice, the choice is recorded, and you can dispute it.

The grounds of knowledge

For us, the name Arxo is tied to the question of the grounds of knowledge. What holds up a statement we take to be true? What is given to us, what did we assume, and how did we reach the conclusion? In *Grounds* this question is explored through observation, evidence, and proof. Here we will walk one such path with our own hands: from a text we have read to a model whose reasoning can be checked.

Having a source does not spare us the work of interpretation. We choose word meanings, pick out conditions, and link phrases. These decisions are worth making visible, so another person can return to the same place and ask: does it really say here what we recorded? Then we can discuss a disagreement in concrete terms: find a missed qualification, propose another reading, pick a case in which the answers diverge.

Where we start

With water. The first chapter will open the textbook model's vocabulary and read it line by line: the kind of objects, two relations, one statement. Then it will ask the first question and get the first answer with the grounds that support it. More complex ideas will come later, one at a time.

Let us open the model.

PART I

Reading the First Model

CHAPTER 01

What This Record Is About

Let us start with a small example about water. The OpenStax general chemistry textbook Chemistry 2e explains in section 14.1¹ that in reactions water can donate a proton to another particle or accept one. Particles capable of playing both roles are called amphiprotic. We will return to that property in the next chapter. For now, take just one piece of information: water can donate a proton.

«In the forward reaction water acts as an acid, donating a proton to ammonia.»

OpenStax · Chemistry 2e · §14.1

The edition's English rendering; consult the source for its wording. © OpenStax, Rice University. Adapted under CC BY-NC-SA 4.0; noncommercial use; changes: English rendering.

For the first example we keep only the message about water's ability to donate a proton.

To begin, it is enough to know that a proton is a positively charged particle. We will not need the details of reactions yet. We will read a ready-made record and work out how it denotes the subject and the property reported about it.

By the end of the chapter you will be able to tell three things apart: what the model's vocabulary lets us record, what is reported about a particular object, and what question is asked. First we will examine short statements, then assemble them into a checkable example.

Vocabulary: what information can be recorded

The Arxo corpus holds a model of this textbook section. Its vocabulary contains this declaration:

```
pub entity Species {  
  label ru unofficial "химическая частица";  
  label en unofficial "chemical species";  
}
```

The word `entity` introduces a type of object. Here the type is called `Species`: these are the chemical particles the model will be able to talk about. The curly braces mark the beginning and the end of the type description.

¹OpenStax. Chemistry 2e, §14.1 "Bronsted–Lowry Acids and Bases". Source on proton transfer between water and ammonia.

The two `label` lines give captions for the reader: `ru` means Russian, `en` means English. The captions let the model present `Species` in different languages. `unofficial` marks the unofficial status of a caption; by itself the word does not say whether it was invented by the model's author or taken from a source. The `pub` mark makes the name available to other packages. We will examine this mechanism in detail in chapter 16. The caption text is enclosed in quotation marks, and a semicolon ends the entry.

Declaring a type still reports nothing about water.

It introduces a concept we will use in records. Next, two properties that can be ascribed to a particle:

```
pub relation donates_proton(s: Species) kind empirical {  
  label ru unofficial "частица отдаёт протон другой частице";  
  label en unofficial "the species donates a proton to another compound";  
}  
  
pub relation accepts_proton(s: Species) kind empirical {  
  label ru unofficial "частица принимает протон";  
  label en unofficial "the species accepts a proton";  
}
```

The word `relation` declares a relation. It can describe a property of a single object or a link between several. Here each relation has one place for an object — a single argument.

In the entry `s: Species`, the letter `s` denotes that place, and after the colon comes the required type: a chemical species. So `donates_proton(s)` can be read as "species `s` donates a proton", and `accepts_proton(s)` as "species `s` accepts a proton". We will supply the concrete object later.

The mark `kind empirical` characterizes the relation as describing an empirical property of its subject. In our example, such information will come with the input data. Where a particular record came from is stated separately; the lone `empirical` mark does not establish that.

So far we have a vocabulary: a particle type and two relations. It lets us talk about their properties, but ascribes those properties neither to water nor to any other particle.

All the blocks above are excerpts from the complete file. At its beginning, the language, the package, and the namespace the declarations belong to are named as well. Those lines say where a record belongs. We will examine the package

arrangement in chapter 16; the complete source is available at the link².



We can already talk about water. There is no information about a particular case yet.

An assertion about a particular object

Information for a particular situation is gathered into a *case*. That is what this book calls the dataset we submit to the model. For the first case, a single entry is enough:

```
assert donates_proton(entity_ref("urn:case:chem:h2o")) { id "k12-1"; origin case_input; }
```

`assert` introduces an assertion. After it comes the familiar relation `donates_proton`, and in parentheses the object to which the property is ascribed. The entry `entity_ref("urn:case:chem:h2o")` refers to that object by an identifier — a string in quotation marks.

In our example the identifier chosen for water is `urn:case:chem:h2o`. The part `h2o` helps a human recognize what it refers to, but by itself it tells the model nothing about chemical properties. What matters for matching records is the whole string. If the same identifier appears in the question, it will denote the same object.

A name is not a property

The fragment `h2o` helps a reader recognize water. The model learns its properties from assertions and rules, not from the spelling of its name.

²Complete model vocabulary.

The assertion also has a name: `k12-1`. It stands after `id` in the curly braces and lets us refer to a particular entry. In chapter 4 such references will help us examine the grounds of an answer. The mark `origin case_input` means the assertion came in with the case input data.

So we have recorded: "In this case it is reported that the particle with the identifier `urn:case:chem:h2o` donates a proton." The vocabulary defined the form of the entry, and the assertion linked the property to a particular object.

How to read a single assertion

```
assert donates_proton(entity_ref("urn:case:chem:h2o"))
{ id "k12-1"; origin case_input; }
```

`donates_proton`
Which property is reported

`urn:case:chem:h2o`
Which object it is about

`k12-1`
The name of this assertion

`origin case_input`
The assertion came from the case input

The object's identifier links statements about the same particle. The assertion's name identifies the individual statement.

Asking about what was recorded

Now let us ask a question about the same water: is there support for asserting that it donates a proton? Before reading the query, try answering on your own. In the case, that property is named directly.

Here are the question and the expected result — an excerpt from the check scenario:

```
evaluate truth(donates_proton(entity_ref("urn:case:chem:h2o")));
expect truth_status == TRUE_ONLY;
expect evaluation_status == COMPUTED;
```

`evaluate truth(...)` asks whether the assertion in parentheses is supported. Inside, the relation `donates_proton` and the water identifier are repeated. The question matches exactly what was reported in the case.

The `expect` lines state the expectations against which the result will be compared. The sign `==` means an equality check. `TRUE_ONLY` means that there is support for the assertion and no support for its negation. In this case the supporting ground is entry `k12-1`. There is no contrary information, nor any rules deriving the negation of this property here.

Support in the model

TRUE_ONLY means there is support for the assertion and none against it. Whether the input information is trustworthy requires a separate check.

The second expectation concerns executing the query: `COMPUTED` means the result was computed. That is a separate matter. In chapter 3 we will examine the four states of support, and in chapter 9 the questions that call for a human decision.

An expectation does not serve as a ground for the answer.

It is needed to check whether the result matched the one we justified in advance.

Information about one object, a question about another

Keep the earlier information: the case records only the property of water. Now ask about ammonia, denoted in the example by the identifier `urn:case:chem:nh3`. Does the model have grounds to confirm or refute that property for ammonia?

In another scenario the question and the expectations look like this:

```
evaluate truth(donates_proton(entity_ref("urn:case:chem:nh3")));  
expect truth_status == NEITHER;  
expect evaluation_status == COMPUTED;
```

The relation `donates_proton` is declared for species, so the question can be asked. But in this case nothing is reported about ammonia's ability to donate a proton, and the package's rules do not establish it for us. The answer `NEITHER` means there are grounds neither for nor against this assertion.

Such a result proves nothing about the chemical properties of ammonia. It shows which grounds are missing from this example. The declared relation lets us ask the question, while a definite answer needs information or an applicable rule.

How to check the example

The question, the input data, and the expectations can be combined into a scenario. The first scenario, about water, looks like this in full:

```

test "urn:query:openstax-chem-acids-12-book-water-donation-recorded" {
  given {
    context {
      decision_time @2026-09-06T12:00:00Z;
      knowledge_time @2026-09-06T12:00:00Z;
      legal_time @2026-09-06;
      timezone "UTC";
    }
    assert donates_proton(entity_ref("urn:case:chem:h2o")) { id "k12-1"; ori
gin case_input; }
    evaluate truth(donates_proton(entity_ref("urn:case:chem:h2o")));
    expect truth_status == TRUE_ONLY;
    expect evaluation_status == COMPUTED;
  }
}

```

After test comes the scenario's name. The given block holds the input data. Inside it are the familiar assertion and the context — the context of the question. It sets three time fields and the time zone. In all the scenarios of the first part these values are the same and do not change the answers under discussion. We will examine their roles in detail in chapter 13; for now, leave them as they are.

After given come the already familiar question and expectations. The outer curly braces join all of this into one scenario. It lives in a file with other check cases; the excerpt shown does not replace the complete file or the model itself.

For checking, install `@arxo/cli@0.2.4` as described in the prologue and use this edition's reader companion. For a routine check of the package, run from its reader/ directory:

```
law test corpus/laws/org/openstax/chemistry-acids
```

`law test` reads the package's declared scenarios and pinned dependencies. For the chapter exercises we also use the lower-level `law engine test`: it lets us pair a copied scenario file with a particular prepared program. Run the following command from the same directory:

```

law engine test \
  corpus/laws/org/openstax/chemistry-acids/tests/openstax-chemistry-acids.lawtes
t \
  --program corpus/clir/openstax-chemistry-acids.lawir.json --imports \
  contexts/openstax-chemistry-acids.json

```

The path before `--program` points at the scenario file, and the one after it at the prepared model the scenarios are checked against. The command runs every scenario in that file. Among the results there will be a line:

```
test PASS: urn:query:openstax-chem-acids-12-book-water-donation-recorded
```

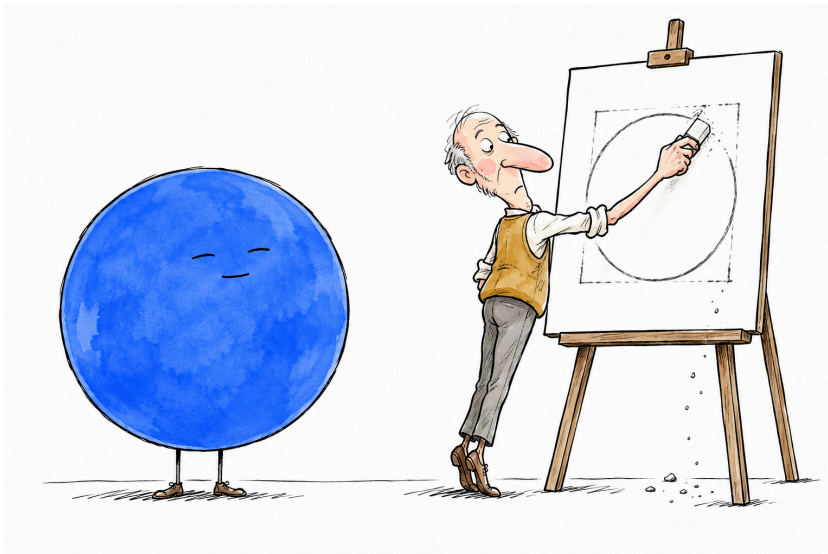
PASS means the result matched the expectations. To check the difference between a result and an expectation, make a copy of the scenario file. Only in scenario 12-book-water-donation-recorded, replace `expect truth_status == TRUE_ONLY;` with `expect truth_status == FALSE_ONLY;`. Run the command again, giving the copy's path instead of the first path. Leave the path after `--program` unchanged.

Now the modified scenario will produce:

```
test FAIL: urn:query:openstax-chem-acids-12-book-water-donation-recorded
truth_status == FALSE_ONLY: в документе TRUE_ONLY
```

Author's paraphrase of the engine output above (the engine's text is in Russian and is quoted verbatim; what follows is the author's English retelling): the check expected FALSE_ONLY, but the document holds TRUE_ONLY.

We changed the expected value, so the check found a mismatch. The answer itself stayed the same: the case still holds the positive entry about water. The check shows us that our expectation does not match the result.



Changing the expectation does not change the answer.

The answer and its test

The model, the case information, and the question



The computed answer

TRUE_ONLY



The reader's expectation

FALSE_ONLY



Comparing the answer with the expectation

TRUE_ONLY $\neq$ FALSE_ONLY

The check failed

FAIL

We changed the expectation. The information and the question stayed the same, so the answer did not change.

What such a check confirms

The relation `donates_proton` declares its argument type: `Species`. Type checking lets us catch, for example, the use of a variable of type `Solution` (a solution) in place of a particle in a rule. It checks that an entry agrees with the declared vocabulary.

The test scenario serves a different purpose: it compares the result with the expectations. Neither an admissible argument type nor matching expectations certify that the input information is trustworthy. For that, we need to find out who reported it and on what ground. We will return to that topic in chapter 11.

In our first example the answer rested directly on the case's assertion. In the coming chapters, conclusions drawn by rules will be added to the input information. Each time we will examine which grounds support the answer and what happens when they change.

Try it yourself

1. Read assertion `k12-1` in ordinary words. Name separately the object's type, its identifier, the relation, and the name of the entry itself.
2. In a fresh copy of the file, find scenario `urn:query:openstax-chem-acids-12-book-water-donation-recorded` and change only its question: instead of `donates_proton`, ask about `accepts_proton` for the same water. Leave the original assertion and the `TRUE_ONLY` expectation unchanged at first. Predict the model's answer and the check's outcome. After running, explain the mismatch, then write a justified expectation and check the scenario once more.
3. Find one more single-argument relation in the complete vocabulary. Read its caption and explain what assertion could be recorded with it. The declaration alone does not mean such an assertion is present in your case.

Check your predictions

In the second exercise there is not enough information for an answer about accepting a proton: the result is `NEITHER`, and the old expectation will not match the result. After changing the expectation to `NEITHER`, the check will pass.

Now we have a vocabulary, one assertion, and a way to ask about it. In the next chapter we will add a second piece of information about water — the ability to accept a proton — and read the rule that links the two abilities to a new property.

CHAPTER 02

From Two Premises to a Conclusion

In the first chapter we recorded that water can donate a proton and asked about that same ability. The entry itself was enough for the answer. Now let us add a second piece of information and ask about a property that follows from the two together. To derive it, the model will also need a rule linking the two premises to a conclusion.

The textbook³ describes water in two roles: in one reaction it donates a proton, in another it accepts one. Let us record both abilities with the familiar relations:

```
assert donates_proton(entity_ref("urn:case:chem:h2o")) { id "k14-1"; origin case_input; }
assert accepts_proton(entity_ref("urn:case:chem:h2o")) { id "k14-2"; origin case_input; }
```

Both assertions concern the same object, `urn:case:chem:h2o`. The assertion names differ: `k14-1` refers to donating a proton, `k14-2` to accepting one. We speak of two abilities of one particle; we do not assert that it donates and accepts a proton simultaneously in a single reaction.

A particle capable of playing both roles is called amphiprotic. The vocabulary holds a separate relation for this property:

```
pub relation amphiprotic(s: Species) kind institutional {
  label ru unofficial "частица амфипротна: способна и отдать, и принять протон";
  label en unofficial "the species is amphiprotic";
}
```

The declaration has the familiar form: the name `amphiprotic`, one argument of type `Species`, captions in two languages. With this relation we can now ask whether water is amphiprotic. But neither of our two assertions answers it directly. To get an answer, we need a link between the known abilities and the new property.

³OpenStax. Chemistry 2e, §14.1 "Brønsted–Lowry Acids and Bases". Textbook: water's two roles and amphiproticity. The fixed fragment `05_14_1_AMPHIPROTIC` is in the sources file.

«Particles capable of donating or accepting protons are called amphiprotic...»

OpenStax · Chemistry 2e · §14.1

The edition's English rendering; consult the source for its wording. © OpenStax, Rice University. Adapted under CC BY-NC-SA 4.0; noncommercial use; changes: English rendering.

This is about one particle's ability to play both roles in different reactions. That is why the formal rule requires both abilities: and.

How to record the link

First, put it in ordinary words: if a particle can donate a proton and can accept one, then it is amphiprotic. The two abilities serve as the conditions of the inference. The statements we reason from are called premises; the statement we reach is the conclusion.

In the same vocabulary file this link is recorded by a rule⁴:

```
@source(OS_14_1_AMPHIPROTIC)
pub rule AmphiproticDonatesAndAccepts strict {
  label ru unofficial "способная и отдать, и принять протон частица амфипротна (14.1)";
  label en unofficial "a species capable of either donating or accepting protons is amphiprotic (14.1)";
  for s: Species;
  when donates_proton(s) and accepts_proton(s);
  then amphiprotic(s);
}
```

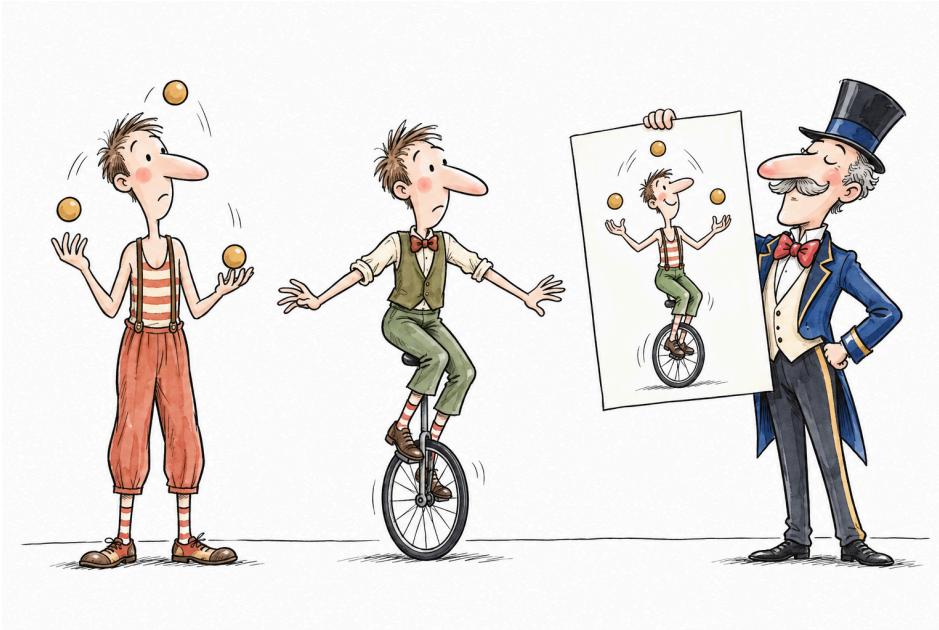
The word rule declares a rule, and AmphiproticDonatesAndAccepts sets its name. By that name we will be able to recognize the rule in the explanation and check whether it was applied. The familiar label lines help read the entry, but the path from premises to conclusion is set by the for, when, and then lines. The historical English label says "either donating or accepting", which is loose wording: the and in when requires both abilities.

for s: Species; introduces a variable s of type Species. In the relation declaration we already saw a place for an argument. The variable acts as a placeholder linking several assertions: in every place marked s, the same particle must appear. Substitute water, and the whole rule concerns water. Substitute ammonia, and it concerns ammonia.

when starts the condition. Between the two premises stands and: both are needed. If water can donate a proton and ammonia can accept one, that is not enough.

⁴Complete vocabulary and the AmphiproticDonatesAndAccepts rule.

You cannot collect two abilities from different particles and ascribe both to one of them.



Two skills exist. But whose? Both premises must concern a single object.

then starts the conclusion. Here it is *amphiprotic(s)*: the very particle for which both conditions hold is *amphiprotic*. The rule lets us derive a new assertion while keeping the link to the premises.

The mark *strict* means the rule is strict: when the condition holds, it yields the conclusion with no exception clause. That is a property of the rule in the model. By itself it does not certify that the information we recorded in the case is trustworthy. We will look at rules with exceptions later, in chapter 8.

What makes a rule strict

Strictness describes how the rule connects premises to a conclusion. It does not establish whether the premises are trustworthy.

Above the rule stands `@source(OS_14_1_AMPHIPROTIC)`. It is a reference to the textbook's fixed fragment, which the package names this way. The fragment first describes water's two roles, then introduces the term "amphiprotic". The formal entry expresses that link through two conditions. We will return to how source references are arranged in chapter 12; for now what matters is that the ground for recording the rule can be found and read.

The new relation has one more feature to explain: `kind institutional` instead of the familiar `kind empirical`. In this vocabulary the first two abilities describe empirical properties, while amphiproticity is presented as a property established by an accepted definition. Here the `institutional` mark does not mean belonging to an institution.

But a relation-kind mark alone concludes nothing. In our scenario the abilities are reported by assertions with `origin case_input`, and amphiproticity is obtained by an explicitly recorded rule. To understand where a particular piece of information came from, look at those entries and the links between them, not only at `kind`.

Checking the conclusion

Take scenario 14 from the same check set⁵ we worked with in the first chapter. Here it is shown in full; the file's other scenarios and header are not included in the excerpt.

```
test "urn:query:openstax-chem-acids-14-book-water-both-roles-amphiprotic" {
  given {
    context {
      decision_time @2026-09-06T12:00:00Z;
      knowledge_time @2026-09-06T12:00:00Z;
      legal_time @2026-09-06;
      timezone "UTC";
    }
    assert donates_proton(entity_ref("urn:case:chem:h2o")) { id "k14-1"; ori
gin case_input; }
    assert accepts_proton(entity_ref("urn:case:chem:h2o")) { id "k14-2"; ori
gin case_input; }
  }
  evaluate truth(amphiprotic(entity_ref("urn:case:chem:h2o")));
  expect truth_status == TRUE_ONLY;
  expect evaluation_status == COMPUTED;
  expect applied(AmphiproticDonatesAndAccepts);
}
```

The scenario's structure is as before. `given` sets the context and the input assertions, `evaluate` records the question, `expect` the expected results. The time fields are left as in the first chapter; at this stage we change only the information and the question.

Let us trace the path to the answer. The case holds both premises for water. The rule links them to amphiproticity. In this scenario there is a ground for such an assertion and no ground against it, so we expect `TRUE_ONLY`. As before, `COMPUTED` means the answer was computed.

⁵Complete check set, scenarios 14 and 15.

The last line adds a new check: `expect applied(AmphiproticDonatesAndAccepts);`. It checks that an application of the named rule appears in the result's supporting graph: the linked record of assertions and inferences behind the answer. That is a useful complement to checking the answer itself: now we also expect a particular line of reasoning. At the same time, `applied` does not assert that the rule was the only possible ground. The explanation will show which entries took part in its application.

Application, not uniqueness

`applied` checks that the justification includes an application of the named rule. The answer may also have other grounds.

From the reader companion directory you can repeat the first chapter's command:

```
law engine test \
  corpus/laws/org/openstax/chemistry-acids/tests/openstax-chemistry-acids.lawtest \
  --program corpus/clir/openstax-chemistry-acids.lawir.json --imports \
  contexts/openstax-chemistry-acids.json
```

It will check the whole file. Among the results there will be a line:

```
test PASS: urn:query:openstax-chem-acids-14-book-water-both-roles-amphiprotic
```

This command uses the model shipped with the reader companion. The path after `--program` points to the prepared model used to compute the answers.

Where the answer came from

For a separate analysis, make a copy of the check file and save it in the reader companion directory under the name `water.lawtest`. Keep in it the three opening language, package, namespace lines and only the `test` block of scenario 14 shown above. The explanation command needs a single scenario so it is clear which answer to unpack. The companion's question preparer supplies the pinned import context and asks the public compiler to prepare the case and question. It makes no decision about the answer; `explain` then computes and displays its grounds.

Run from the same root:

```
chapter2_explain=$(mktemp -d)
python3 reader_tools/prepare_question.py water.lawtest \
  --program corpus/clir/openstax-chemistry-acids.lawir.json --imports \
  contexts/openstax-chemistry-acids.json --out "$chapter2_explain"
law engine explain "$chapter2_explain"
```

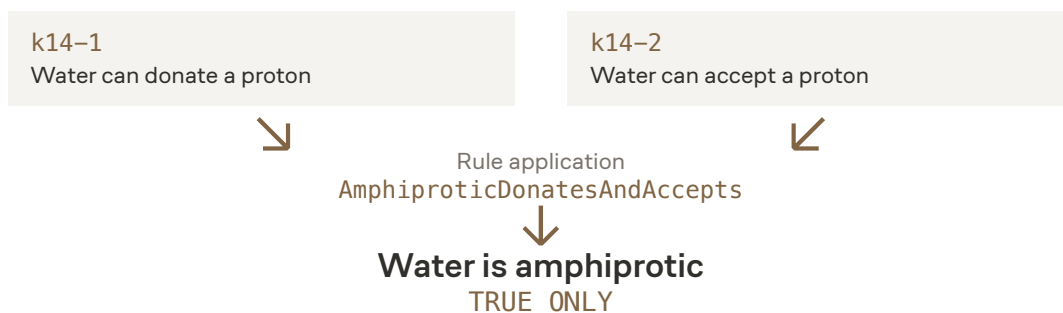
The `explain` command shows what the answer rests on. Its output is below; the long rule-application identifier is shortened with an ellipsis.

```
the species is amphiprotic(h2o): TRUE_ONLY
rule_application the species is amphiprotic(h2o)
  (urn:proof:apply:AmphiproticDonatesAndAccepts:f103e243...)
  assertion the species donates a proton to another compound(h2o)
    (urn:proof:assert:urn:openstax:clir:chemistry-acids#k14-1)
  assertion the species accepts a proton(h2o)
    (urn:proof:assert:urn:openstax:clir:chemistry-acids#k14-2)
```

This output uses the vocabulary's English captions. The first line reports the answer about water's amphiproticity. Below it sits a `rule_application`, the application of the rule named `AmphiproticDonatesAndAccepts`. Further below, indented further, are two assertions: the input assertions `k14-1` and `k14-2`. Their names let us return to the particular entries in the case.

In the first chapter the answer rested on a single assertion. Now one more step has appeared between the input information and the answer. We can show not only the resulting property, but the rule that produced it, together with both premises.

How the answer was obtained



Both premises concern `urn:case:chem:h2o`. In this scenario there is no ground against amphiproticity.

When one piece of information is missing

Remove the entry about accepting a proton. In the check set, this case corresponds to scenario 15:

```

test "urn:query:openstax-chem-acids-15-book-water-one-role-no-conclusion" {
  given {
    context {
      decision_time @2026-09-06T12:00:00Z;
      knowledge_time @2026-09-06T12:00:00Z;
      legal_time @2026-09-06;
      timezone "UTC";
    }
    assert donates_proton(entity_ref("urn:case:chem:h2o")) { id "k15-1"; ori
gin case_input; }
    evaluate truth(amphiprotic(entity_ref("urn:case:chem:h2o")));
    expect truth_status == NEITHER;
    expect evaluation_status == COMPUTED;
  }
}

```

Something is still known about water: it can donate a proton. But the question is about amphiproticity, and that conclusion needs both abilities. One premise is present, the other is not, so the rule under discussion does not apply. In this scenario there is no other ground for an answer for or against amphiproticity either. The result is NEITHER.

Compare with the first chapter. There we asked about ammonia, about which the case held no assertions at all. Now there is an entry about water, but it is not enough to answer the new question. Both cases produced NEITHER, though the input information differed.

The missing premise did not turn into a negative conclusion.

To assert that water is not amphiprotic would take a ground against that property. Mere silence about accepting a proton is not enough for such a conclusion.

The direction of the rule

The entry when ...; then ...; sets a direction: from the conditions to the conclusion. The rule under discussion lets us conclude amphiproticity from the two abilities. It does not license the reverse move.

If only amphiprotic for water is recorded in the case, this rule will not derive the abilities to donate and accept protons by reasoning backwards. Although the definition connects all three properties, the formal entry expresses one particular transition. When reading a rule, distinguish what we understand from the definition from the conclusion recorded in the model.

That is why in our scenario it matters to keep exactly the two input abilities. If the property we seek is reported in advance, the answer can rest on that entry, and we will no longer test whether the two premises suffice to derive it by rule.

What changed in the premises

Information in the case	Does the rule apply to water?	Answer about water's amphiproticity
Water donates; water accepts	Yes	TRUE_ONLY
Water donates; no information about accepting	No	NEITHER
Water donates; ammonia accepts	No	NEITHER

The scenarios under discussion are compared: there are no other grounds for or against water's amphiproticity in them.

Try it yourself

For each exercise, take a fresh copy of `water.lawtest` with scenario 14. Run it with the `test` command, substituting the copy's name for the full set's path; the model after `--program` stays the same. First predict the result, then compare it with the one obtained.

1. Swap the two assertions inside `given`. Will the answer and the rule's application change? Check this without changing the expectations.
2. Delete assertion `k14-1`, keeping only the ability to accept a proton. Keep the question and the expectations at first. Which two checks should find a mismatch? After running, explain why and correct the expectations for the new case.
3. Keep both lines, but in assertion `k14-2` replace `urn:case:chem:h2o` with `urn:case:chem:nh3`. Still ask about water's amphiproticity. Are the two entries now enough for the conclusion? First check the old expectations, then write new ones and justify them through the variable `s`.

Check your predictions

In the first exercise both premises survive, and the check passes. In the second and third the answer is `NEITHER`: either one ability is missing, or the abilities belong to different objects. The old `TRUE_ONLY` and `applied(AmphiproticDonatesAndAccepts)` expectations fail. For the revised cases, replace the expected status with `NEITHER` and drop the expectation that this rule is applied; keep `COMPUTED`. Checks with such

expectations will pass. We will learn how to express the absence of an application with a separate check later.

Now we can follow a short inference: find the two premises, make sure they concern one particle, and trace the transition to the conclusion. The next step is to work out what changes when an ability is not merely unmentioned but explicitly recorded as negated.

CHAPTER 03

Established, Refuted, Conflicting, Unknown

In the last chapter we removed one piece of information about water. The remaining data was not enough for the conclusion about amphiproticity, and the result was NEITHER. But deleting the message "water can accept a proton" and recording "water cannot accept a proton" are different actions. In the first case the information is gone; in the second, a counter-assertion appears.

Let us examine that difference using the familiar question: can water donate a proton? Set the amphiproticity conclusion aside for now and look at the input assertions themselves. That will show what changes when one entry is added or removed.

Two questions instead of one

For an answer, two sides must be checked independently: whether there is a ground for the assertion and whether there is a ground against it. That yields four combinations⁶:

Two independent questions

How to read the answer

	Ground against: present	Ground against: absent
Ground for: present	BOTH grounds conflict with each other	TRUE_ONLY assertion established
Ground for: absent	FALSE_ONLY assertion refuted	NEITHER assertion neither established nor refuted

Support for and support against the assertion are checked independently.

⁶Arxo specification, §§61–66: independent grounds, explicit negation, status checks, and rule conditions.

The language uses the statuses `TRUE_ONLY`, `FALSE_ONLY`, `BOTH`, and `NEITHER`. Two states are already familiar: in the first chapter the positive entry about water gave `TRUE_ONLY`, and the question about ammonia, about which nothing was reported, gave `NEITHER`.

The words "established" and "refuted" here refer to the grounds in the case and model under discussion. They do not certify that the input information is true. The word "unknown" does not mean that no answer can ever be found: the grounds available here support neither side.

«...contradictory data may be passed to such a knowledge base for storage, while on some topics data may be missing.»

G. Sutcliffe, F. J. Pelletier, A. P. Hazen · Making Belnap's "Useful Four-Valued Logic" Useful · 2018, p. 116
The edition's English rendering; consult the source for its wording.

The paper discusses FDE logic. The distinction between contradiction and a gap in the data is compared here; the conditions of Arxo rules are set by its specification.

How to record a negation

A positive assertion began with `assert donates_proton(...)`. For a negative one, add `not` before the relation name. In scenario 16⁷ it looks like this:

```
test "urn:query:openstax-chem-acids-16-book-water-donation-refuted" {
  given {
    context {
      decision_time @2026-09-06T12:00:00Z;
      knowledge_time @2026-09-06T12:00:00Z;
      legal_time @2026-09-06;
      timezone "UTC";
    }
    assert not donates_proton(entity_ref("urn:case:chem:h2o")) { id "k16-1";
  origin case_input; }
  }
  evaluate truth(donates_proton(entity_ref("urn:case:chem:h2o")));
  expect truth_status == FALSE_ONLY;
  expect evaluation_status == COMPUTED;
}
```

The new line reads: "Recorded: water cannot donate a proton." The question stayed positive: `evaluate truth(donates_proton(...))`. We ask about the ability to donate a proton, while the only assertion is directed against it. Hence the answer `FALSE_ONLY`.

⁷Complete check set: scenarios 16–19 and 22 discussed in this chapter.

This is a deliberately altered teaching example. Earlier we recorded water's ability to donate a proton following the textbook's description; now we check how the language handles the opposite message. The resulting answer shows what follows from the entered information, not a new discovery about water's properties.

The test block is shown in full. As before, it belongs to the shared file with the language, package, namespace header. The context, the question, and the `COMPUTED` expectation are arranged as in the previous chapters. New here are only the explicit negation and its matching answer status.

What conflict produces

Now keep the positive entry and add the negative one. In scenario 17 both concern one ability of the same water:

```
test "urn:query:openstax-chem-acids-17-book-water-donation-both" {
  given {
    context {
      decision_time @2026-09-06T12:00:00Z;
      knowledge_time @2026-09-06T12:00:00Z;
      legal_time @2026-09-06;
      timezone "UTC";
    }
    assert donates_proton(entity_ref("urn:case:chem:h2o")) { id "k17-yes"; o
  rigin case_input; }
    assert not donates_proton(entity_ref("urn:case:chem:h2o")) { id "k17-no"
; origin case_input; }
  }
  evaluate truth(donates_proton(entity_ref("urn:case:chem:h2o")));
  expect truth_status == BOTH;
  expect evaluation_status == COMPUTED;
}
```

Assertion k17-yes gives a ground for water donating a proton, and k17-no gives a ground against it. Neither entry has vanished. Hence the answer `BOTH`: the case holds grounds on both sides.

The scenarios can be checked with the familiar command from the reader companion directory:

```
law engine test \
  corpus/laws/org/openstax/chemistry-acids/tests/openstax-chemistry-acids.lawtes
t \
  --program corpus/clir/openstax-chemistry-acids.lawir.json --imports \
  contexts/openstax-chemistry-acids.json
```

For scenario 17 it will print:

```
test PASS: urn:query:openstax-chem-acids-17-book-water-donation-both
```

Author's paraphrase of the engine output above: the check of scenario 17 passed — the obtained conflict matched the expected BOTH status.

The `evaluation_status == COMPUTED` expectation held as well: the BOTH answer could be computed.

Conflicting information and a failed query run are different things.

Let us look at the explanation. Make a copy of the check file, `water-both.lawtest`, in the reader companion directory. Keep in it the three header lines and only the scenario 17 block, as we did with scenario 14 in the last chapter. Run:

```
chapter3_explain=$(mktemp -d)
python3 reader_tools/prepare_question.py water-both.lawtest \
  --program corpus/clir/openstax-chemistry-acids.lawir.json --imports \
  contexts/openstax-chemistry-acids.json --out "$chapter3_explain"
law engine explain "$chapter3_explain"
```

Both grounds are visible in the explanation:

```
the species donates a proton to another compound(h2o): BOTH
  assertion ~ the species donates a proton to another compound(h2o)
    (urn:proof:assert:urn:openstax:clir:chemistry-acids#k17-no)
  assertion the species donates a proton to another compound(h2o)
    (urn:proof:assert:urn:openstax:clir:chemistry-acids#k17-yes)
```

Author's paraphrase of the engine output above: the question of whether water donates a proton is answered BOTH — the negating entry k17-no and the affirming entry k17-yes each support their own side, with no rules applied.

The sign `~` before a caption denotes negation. Next to each of the two entries appears its identifier: we can go back to the assertion for and the assertion against. In this case the answer rests directly on them, with no rules applied.

Swapping those two lines inside given leaves the answer BOTH. A line's later position does not give it greater weight and does not cancel the other entry. To resolve the conflict, the grounds of the messages themselves must be investigated. We will see in chapter 11 how to present sources of information and their authors.

BOTH does not mean "fifty percent probability". The four states report what grounds exist, not a degree of confidence. In this scenario neither vote-counting nor picking the more convincing entry is specified.

Conflicting support is not a probability

BOTH preserves support both for and against the assertion. It does not assign a probability to it.

When nothing is said about the ability

In scenario 18 only water's other ability is reported:

```
assert accepts_proton(entity_ref("urn:case:chem:h2o")) { id "k18-1"; origin case_input; }
```

While the question and the expectation concern donating a proton:

```
evaluate truth(donates_proton(entity_ref("urn:case:chem:h2o")));  
expect truth_status == NEITHER;
```

These are two excerpts from the scenario: the input assertion and the lines with the question and the expected answer. The complete block lives in the same check set. It holds neither a positive nor a negative entry about donating a proton. In this model, donating does not follow from accepting a proton either. Hence the answer NEITHER.



Two opposite messages about the same gate at the same moment — BOTH. No message at all — NEITHER.

Now compare all four cases for the same question:

Entries about water donating a proton	Scenario	Answer
only positive	12, chapter 1	TRUE_ONLY
only negative	16	FALSE_ONLY
both	17	BOTH
neither	18	NEITHER

The last two lines are especially easy to mix up.

When entries disagree, they cannot be treated as absent.

When there are no entries, the assertion cannot be treated as refuted. An explicit not reports a negation; the mere absence of information creates no such entry. Separate conditions on the completeness of information will appear in chapter 7.

A negative conclusion by rule

Return to amphiproticity. In the last chapter we obtained that property from two abilities. Now take a different case: the ability to donate a proton is established, while the ability to accept one is refuted. For a conclusion about amphiproticity the vocabulary has a separate rule⁸:

```
@source(OS_14_1_AMPHIPROTIC)
pub rule NotAmphiproticWithoutAcceptance strict {
  label ru unofficial "отдающая протон частица с опровергнутой акцепторностью
не амфипротна (14.1)";
  label en unofficial "a proton donor whose acceptance is refuted is not amphi
protic (14.1)";
  for s: Species;
  when donates_proton(s) and refuted(accepts_proton(s));
  then not amphiprotic(s);
}
```

It can be read as: "If a particle can donate a proton, while its ability to accept a proton is refuted, then the particle is not amphiprotic." The variable *s* still binds both premises and the conclusion to one particle.

The condition introduces the expression `refuted(accepts_proton(s))`. It checks that accepting a proton has exactly the `FALSE_ONLY` status: a ground against it is present and no ground for it is present. It fails for `NEITHER`, where no ground against it is present. It also fails for `BOTH`, where a ground for it is present as well as a ground against it.

Two actions matter here. `assert not` in a case adds a negative assertion. `refuted(...)` in a rule checks the state of an assertion given the available grounds. Those grounds may be input or derived; the check does not reduce to finding a line with `assert not`.

A negative assertion and its test

`assert not` adds a negative assertion. `refuted` requires `FALSE_ONLY`: support against the assertion and none for it. The condition is not met when the state is `BOTH`.

The conclusion reads `then not amphiprotic(s)`. This is a negative conclusion: applying the rule yields a ground against amphiproticity. Let us check it on scenario 19:

⁸The `NotAmphiproticWithoutAcceptance` rule in the complete vocabulary. The `refuted` check is defined in §65 of the specification; the ordinary positive premise in §66.

```

test "urn:query:openstax-chem-acids-19-book-water-acceptance-refuted-not-amphiprotic" {
  given {
    context {
      decision_time @2026-09-06T12:00:00Z;
      knowledge_time @2026-09-06T12:00:00Z;
      legal_time @2026-09-06;
      timezone "UTC";
    }
    assert donates_proton(entity_ref("urn:case:chem:h2o")) { id "k19-1"; origin case_input; }
    assert not accepts_proton(entity_ref("urn:case:chem:h2o")) { id "k19-2"; origin case_input; }
  }
  evaluate truth(amphiprotic(entity_ref("urn:case:chem:h2o")));
  expect truth_status == FALSE_ONLY;
  expect evaluation_status == COMPUTED;
  expect applied(NotAmphiproticWithoutAcceptance);
}

```

Here donating a proton has the status `TRUE_ONLY`, accepting — `FALSE_ONLY`. Both premises of the negative rule hold. The positive rule from the last chapter does not apply: the ability to accept a proton is not established. As a result, there is a ground against amphiproticity and none for it. The answer is `FALSE_ONLY`, and the applied expectation confirms the application of `NotAmphiproticWithoutAcceptance`.

This differs from scenario 15 in the last chapter. There nothing was said about accepting a proton, and the answer about amphiproticity came out `NEITHER`. Here a negative piece of information appeared, which the rule builds on. A missing premise and a refuted premise give different results.

When a disputed premise stops the conclusion

One case remains to test our understanding of `refuted`. Suppose the ability to donate a proton is established, while accepting a proton has entries both for and against. Accepting a proton gets `BOTH`. How will that affect amphiproticity?

In the positive rule, `accepts_proton(s)` is an ordinary positive premise, rather than an explicit status check. It requires `TRUE_ONLY`. A ground for accepting a proton is not enough when a ground against it is also present. So the positive rule does not apply. The negative rule does not apply either: its `refuted` condition requires `FALSE_ONLY`, not `BOTH`.

In this case there are no other grounds either for or against amphiproticity, so the result is `NEITHER`. Scenario 22 records this case. In every row below, donating a proton remains established; only the status of accepting a proton changes:

Which rules apply

State of accepting a proton	Positive rule	Negative rule	Answer about amphiproticity
TRUE_ONLY	Applies	Does not apply	TRUE_ONLY
FALSE_ONLY	Does not apply	Applies	FALSE_ONLY
NEITHER	Does not apply	Does not apply	NEITHER
BOTH	Does not apply	Does not apply	NEITHER

In all rows proton donation is established. The positive premise requires TRUE_ONLY, refuted requires FALSE_ONLY. There are no other grounds for amphiproticity in these scenarios.

The table concerns these rules and this input information. A conflict in one ability need not become a conflict in the derived property. To predict the result, read the conditions of the rules being applied.

Try it yourself

For single-scenario work use a copy with the three header lines, as when running the explanation. In the test command, replace the shared set's path with your copy's name; the path after `--program` stays the same. Before each change, predict the answer.

1. In a copy of scenario 17, swap the two entries. Check the result and the explanation. Then make three fresh copies: in one keep only the positive entry, in another keep only the negative one, and in the third delete both. Do not change the question or the context. For each fresh copy, first run the check with the old BOTH expectation and note the mismatch; then replace it with the expectation you can justify from the entries that remain.
2. In a copy of scenario 19, delete the negative entry k19-2. Why do the old status and rule-application expectations no longer hold? Explain the result through both premises, then fix the expectations.
3. In a fresh copy of scenario 19, keep both input entries and add a positive entry about water accepting a proton. Give it a new name, k19-3. Still ask about amphiproticity. First check the old expectations, then correct them, explaining separately why each of the two rules under discussion does not apply.

Check your predictions

In the first exercise the swap keeps BOTH. Deleting the entries one by one gives TRUE_ONLY, FALSE_ONLY, and NEITHER. In the second and third exercises the answer about amphiproticity is NEITHER, but the reasons differ: information about accepting

a proton is either missing or conflicting. In both cases replace the `FALSE_ONLY` expectation with `NEITHER` and drop the expectation that the negative rule is applied. Keep `COMPUTED`. Checks with the new expectations will pass.

We have learned to tell apart a missing ground, a refutation, and a conflict. But the same answer state does not yet say how exactly it came about. In the next chapter we will examine the explanation in more detail: walk from the answer back to the input information and the rule's source, then find out why an expected conclusion was not reached.

CHAPTER 04

Why We Got This Answer

We have already obtained several answers about the same water. With both abilities established, the model concluded that water was amphoteric. When one piece of information was missing, the rule could not yield that conclusion. An explicit refutation of an ability gave a ground for a negative conclusion, while a conflict could stop both rules.

Now let us try explaining these differences to another person. Naming the state alone is not enough: we must show which information we used, how the answer followed from it, and where the rule itself came from. Start with the familiar positive case, then examine why the conclusion is missing in another case.

What the explanation shows

In scenario 14 both abilities of water are recorded. The `explain` command from the second chapter shows the following answer. The long rule-application identifier is shortened here with an ellipsis:

```
the species is amphoteric(h2o): TRUE_ONLY
rule_application the species is amphoteric(h2o)
  (urn:proof:apply:AmphotericDonatesAndAccepts:f103e243...)
  assertion the species donates a proton to another compound(h2o)
    (urn:proof:assert:urn:openstax:clir:chemistry-acids#k14-1)
  assertion the species accepts a proton(h2o)
    (urn:proof:assert:urn:openstax:clir:chemistry-acids#k14-2)
```

Author's paraphrase of the engine output above: water's amphotericity is established (TRUE_ONLY) by applying the rule AmphotericDonatesAndAccepts to the two input entries — k14-1 about donating a proton and k14-2 about accepting one.

Read it bottom-up. Assertion k14-1 reports donating a proton, k14-2 accepting one. They concern one object. Above is the application of AmphotericDonatesAndAccepts: the rule links these abilities to amphotericity. The first line shows the result, TRUE_ONLY.

Such an analysis answers the question "what did this conclusion come from?" We can find the particular input entries and see the step from them to the conclusion. But another question remains: why does the model contain this rule?

Above its declaration we already saw this line:

@source(OS_14_1_AMPHIPROTIC)

This is a separate excerpt from the declaration, not a rule of its own. It points at the fragment named `OS_14_1_AMPHIPROTIC` in the same package's `sources.law` file⁹. Find it there: the `locator` field records `14.1/amphiprotic`, and the `text` field holds the OpenStax Chemistry 2e paragraph about water's two roles and amphiprotic particles. We will examine the structure of such entries in detail in chapter 12. For now it is enough to find, via the reference, the text the rule rests on.

There are two linked checks. The explanation lets us retrace the conclusion from the information in the case. The rule's source reference lets us compare the formal rule with the textbook text. The paragraph defining amphiproticity does not replace checking the trustworthiness of each input assertion. In our example the information about water is taken from the textbook description; in another case we would need to investigate the grounds for each input assertion.

«Data provenance is information about entities, activities, and people involved in producing data or things...»

W3C · PROV-Overview · 30 April 2013 · §1

The edition's English rendering; consult the source for its wording. © 2013 W3C. Commentary for this software tutorial; the document's own licence applies.

Provenance information helps check the result. The principle of traceability is compared here; Arxo's conformance to PROV standards is not claimed.

Two paths to check

What the answer was obtained from

Entries k14-1 and k14-2 about water



Rule application



Water's amphiproticity: `TRUE_ONLY`

Where the rule came from

OpenStax Chemistry 2e, §14.1



Fragment in `sources.law`



@source reference at the declaration

The left path traces the derivation; the right traces the rule back to its source. Neither replaces checking whether the input information is trustworthy.

How to ask about a conclusion that failed

Return to the case where only the ability to donate a proton is known. The answer about amphiproticity will be `NEITHER`, as in the second chapter. Now we care about the reason: what kept the positive conclusion from coming about?

⁹The model's source entries, fragment `OS_14_1_AMPHIPROTIC`; its use — in the rule's declaration.

For that, there is the `why_not` question¹⁰. It stands in place of the familiar `truth`. Here is scenario 20 in full from the check set:

```
test "urn:query:openstax-chem-acids-20-book-water-why-not-without-acceptance" {
  given {
    context {
      decision_time @2026-09-06T12:00:00Z;
      knowledge_time @2026-09-06T12:00:00Z;
      legal_time @2026-09-06;
      timezone "UTC";
    }
    assert donates_proton(entity_ref("urn:case:chem:h2o")) { id "k20-1"; origin case_input; }
  }
  evaluate why_not(amphiprotic(entity_ref("urn:case:chem:h2o")));
  expect truth_status == NEITHER;
  expect blocked_by("AmphiproticDonatesAndAccepts");
}
```

In given one entry about water remains. The line `evaluate why_not(amphiprotic(...))` asks why amphiproticity is not established. The name inside the file uses an underscore: `why_not`.

The new expectation `blocked_by("AmphiproticDonatesAndAccepts")` checks that an entry about the named rule is present in the `why_not` result. Here the name is passed as a quoted string. That check alone neither establishes that the rule failed to fire nor checks the particular reason. It must be read together with the expected answer status and the condition's explanation¹¹.

Naming the rule does not explain the obstacle

`blocked_by` checks that the `why_not` answer includes an entry about the named rule. Read the result status and the state of the relevant premise separately.

First run the check over the whole set from the reader companion directory:

```
law engine test \
  corpus/laws/org/openstax/chemistry-acids/tests/openstax-chemistry-acids.lawtest \
  --program corpus/clir/openstax-chemistry-acids.lawir.json --imports \
  contexts/openstax-chemistry-acids.json
```

Among the results there will be:

¹⁰Arxo specification, §§183–185: explanation and why-not. A missing condition must not be declared false.

¹¹Arxo specification, §267.4: expectation-observations. Example — scenario 20 in the complete check set.

```
test PASS: urn:query:openstax-chem-acids-20-book-water-why-not-without-acceptanc
e
```

Author's paraphrase of the engine output above: the check of scenario 20 passed — the NEITHER status and the mention of the rule AmphiproticDonatesAndAccepts in the why_not result matched the expectations.

To read the reason, make a copy of the check file, water-why.lawtest, in the reader companion directory. Keep in it the three opening language, package, namespace lines and only scenario 20. This is the same way of preparing a single scenario we used for explain.

```
chapter4_why_not=$(mktemp -d)
python3 reader_tools/prepare_question.py water-why.lawtest \
  --program corpus/clir/openstax-chemistry-acids.lawir.json --imports \
  contexts/openstax-chemistry-acids.json --out "$chapter4_why_not"
law engine why-not "$chapter4_why_not"
```

The command's name uses a hyphen: why-not. The command asks why a positive conclusion is missing; it can also be applied to a single scenario with a truth question. Below is the substantive part of the answer. The closing section on kinds of diagnostics is omitted here.

```
статус результата: NEITHER

блокираторов, подтверждённых документом, нет

не определено (1) — условие НЕ опровергнуто, вывода просто нет:
  urn:openstax:clir:chemistry-acids#AmphiproticDonatesAndAccepts
    правило a species capable of either donating or accepting protons is amhipr
    otic (14.1) не сработало, но условие НЕ опровергнуто: the species accepts a prot
    on(h2o) — пара опор §62: NEITHER
```

Author's paraphrase of the engine output above (the engine's text is in Russian and is quoted verbatim; what follows is the author's English retelling): the result status is NEITHER, with no document-confirmed blockers. The rule AmphiproticDonatesAndAccepts did not fire, but its condition was not refuted either — the premise about water accepting a proton has NEITHER support (no ground for or against, cf. §62), so the conclusion is simply missing for lack of information.

The answer names the rule and the particular premise: accepts_proton(h2o). Its state is NEITHER: no ground either for or against accepting a proton. So the rule's condition is not established, but not refuted either.

Scenario 20: where the derivation stopped

Why is water's amphiproticity not established?

Proton donation

TRUE_ONLY

Entry k20-1 is present

Proton acceptance

NEITHER

No ground either for or against

AmphiproticDonatesAndAccepts

The positive rule does not apply

Answer about amphiproticity: NEITHER

Both premises must be TRUE_ONLY for the positive rule. Unknown proton acceptance also fails to meet the refuted condition of the negative rule.

Together, "no blockers" and "undetermined" mean that the condition is not refuted; the conclusion is missing for lack of information. The `blocked_by` expectation still passes because the analysis contains an entry about the named rule. The name guarantees less than it may suggest: finding the rule in the analysis does not establish that it was blocked, or explain why it did or did not apply.

The §62 reference in the last line points at the specification's description of the four states. The familiar NEITHER suffices for reading the example.

The missing ground did not become a negation, and the explanation pinned down which piece of information is lacking.

What an extra entry changes

Now check the reverse situation: both needed abilities are already recorded, and we add one more piece of information. The vocabulary holds a relation:

```
pub relation conjugate_acid_of(acid: Species, base: Species) kind empirical {
  label ru unofficial "первая частица – сопряжённая кислота второй";
  label en unofficial "the first species is the conjugate acid of the second"
;
}
```

It takes two arguments of type `Species`: first the particle that acts as the conjugate acid, then its base. The argument order sets the meaning of the link. This is a small extension of the familiar form: where a relation used to describe one particle, it now links two.

In scenario 21 an assertion is added to water's two abilities:

```
assert conjugate_acid_of(entity_ref("urn:case:chem:h3o"), entity_ref("urn:case:chem:h2o")) { id "k21-3"; origin case_input; }
```

It reports that hydronium h3o is the conjugate acid of water h2o. This is a separate excerpt; the complete scenario lives in the same check file. The question is still about water's amphiproticity.

The rule `AmphiproticDonatesAndAccepts` has two premises: donating and accepting a proton. In this package there is no chain of rules that would use the added link for an answer about amphiproticity. So the result stays `TRUE_ONLY`.

Check scenario 21 and view its explanation the same way as scenario 14: keep only the needed block in the copy and call `explain`. The explanation will hold assertions k21-1 and k21-2, while k21-3 will not appear. Deleting the third entry while keeping the rest changes neither this scenario's answer nor its explanation.

That observation concerns the particular question and model. In another model the link could turn out to be a premise of a separate rule. To call information extraneous, check whether it takes part — directly or through further conclusions — in obtaining the answer.

Where to look for an error in the entry

There is a change that looks smaller still. In a copy of scenario 14, replace only in the assertion about accepting a proton `urn:case:chem:h2o` with `urn:case:chem:h20`: the last character becomes a zero instead of the letter "o".

To a reader, that is a typo. To the engine, it is a valid identifier of a different object. Now donating belongs to h2o and accepting to h20. The two premises no longer come together for water, and the answer about its amphiproticity becomes `NEITHER`.

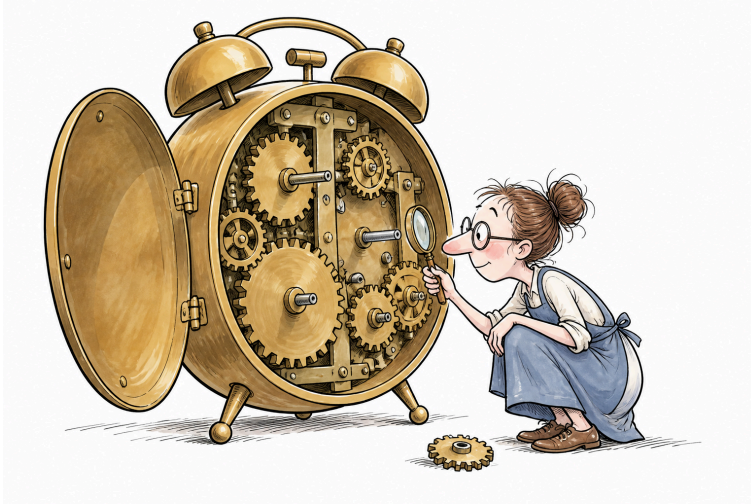
Similar spelling, different objects

The last character in h2o is the letter o; in h20 it is the digit 0. The explanation identifies the missing condition for h2o. It cannot infer which identifier the author intended.

The old `TRUE_ONLY` and rule-application expectations will fail: the check prints `FAIL`. Then the `why-not` command will point at the unknown status of proton acceptance by h2o in particular. It will not say that we mistyped the last character, nor offer to equate the two names automatically. We discover the typo by matching the named premise against the input entries.

So expectation and explanation help each other. The expectation catches a mismatch with the intended result; the explanation narrows the search to a particular premise. In this exercise the identifier needs fixing. Simply replacing the expecta-

tion with `NEITHER` would make the check pass but leave the wrongly recorded case in place.



The explanation narrows the search to a particular link. A person must check the input and correct the error.

What a table or a program needs

Two abilities can be kept in a table, with the third property computed by a formula. In Python a short program suffices for that too. The question is which distinctions the chosen data representation keeps.

Consider a small self-contained Python example. Lines with `>>>` are commands being entered; the following lines, the result. Here one entry links a particle's name to the value `True`:

```
>>> recorded = {"h2o": True}
>>> recorded.get("nh3", False)
False
>>> records = [("h2o", True), ("h2o", False)]
>>> dict(records)["h2o"]
False
```

In the first case ammonia is absent from the dictionary. The `get` call returns the preselected default, `False`. Missing information thus becomes indistinguishable from a recorded negation. In the second case two entries about one name collapse into one: the dictionary keeps the last value. A conflict is no longer visible in such a representation.

Both effects follow from how the information is stored. In Python, support for an assertion and support against it can be kept separately to recover all four states.

In a table, separate columns can be added for them. To reproduce this chapter's explanation, the input entries' names, information about applied rules, and source references are needed as well.

Arxo sets a shared way to link these elements: assertion states, rule-application conditions, and result explanations.

The model's author remains responsible for the choice of rules and input information.

When a model is handed to another person, the shared conventions let them read the entry and check how the conclusion was reached without knowing how a particular program is built.

Try it yourself

In each exercise work with a separate copy holding the file header and one scenario. Substitute the copy's name in the commands; the model path after `--program` stays the same. Try each exercise before reading the discussion that follows.

1. Take scenario 18 from the last chapter: accepting a proton is recorded, while the question asks about donating. Call `why-not`. How does the reason for this `NEITHER` result differ from the one in scenario 20? Is there a rule in the vocabulary that could derive donation of a proton?
2. Reproduce the typo in scenario 14. First check the old expectations, then read `why-not` and find the mismatching identifiers in the input entries. Fix the typo and make sure the old expectations hold again.
3. Delete only entry `k21-3` from a copy of scenario 21. Compare the answer and the explanation before and after deletion. Name both premises that keep supporting the conclusion.

Check your predictions

In the first exercise the explanation will point out that this program has no rule with `donates_proton` as its conclusion: the information must come from the case. In scenario 20 there is a rule for the sought property, but it lacks a premise. In the second exercise fixing the name returns `TRUE_ONLY` and the rule's application. In the third, deleting the entry keeps both grounds and the old answer.

We have walked the first small example from vocabulary to explanation. Now try telling someone who has not read the chapter about it: what information is set, what follows by rule, why a missing premise differs from a refuted one, and where to find the text the rule refers to.

In the next chapter the subject changes: instead of water's properties, triangle side lengths appear. But start with the familiar question — whether the input conditions suffice for the conclusion.

PART II

State the Conditions

CHAPTER 05

First Check That the Object Exists

In the first part, the conclusion about amphiproticity required two abilities of a single particle. If one of the premises was not established, the rule did not apply. Now consider a similar problem with numbers: we are given the lengths of three sides and must find out whether they form a right-angled triangle.

For sides 3, 4, and 5 the answer is familiar: the square on five equals the sum of the squares on three and four. Here “the square on” a side means the square of that side's length. But before using the right-angle test, we must make sure that the given sides can form a triangle at all. Let us record both steps so that one conclusion becomes a premise for the other.

We will work with the package on Euclid's *Elements*. Proposition I.20 states the triangle inequality: the sum of any two sides of a triangle is greater than the third. Proposition I.48 gives the test for a right angle: if the square on one side of a triangle equals the sum of the squares on the other two, the angle between those two sides is right. In the package, separate rules correspond to these propositions¹².

«In any triangle, the sum of any two sides is greater than the remaining side.»

Euclid · *Elements* · Book I, Proposition 20
The edition's own paraphrase of Proposition I.20.

The word “greater” sets a strict bound. Equality of the sum with the third side describes a degenerate case.

How to record three lengths

Objects of the new vocabulary have the type `Trigonon`. Its name means “triangle”, but assigning an object to this type does not yet confirm that the given lengths form a triangle. The object lets us gather information about one figure under consideration; we will establish separately whether its side lengths can form a triangle.

The relation `pleurai` records lengths:

¹²Euclid, *Elements*, I.20 and I.48: the right-angle test, the electronic edition of David Joyce. The package's Greek fragments are pinned in `sources.law`. The English renderings of the propositions in this chapter are the translator's own.

```

relation pleurai(t: Trigonon, a: Integer, b: Integer, c: Integer)
  kind empirical {
    key(t);
    label grc unofficial "αἱ τρεῖς πλευραὶ τοῦ τριγώνου";
    label ru unofficial "три стороны треугольника в порядке записи дела; единица
длины у всех трёх одна и та же по построению записи – величины безразмерны (Integer), и вопрос пакета от неё не зависит";
  }

```

There are four arguments here. The first, *t*, denotes an object of type `Trigonon`. The other three denote the side lengths in the order of recording. Each carries the type `Integer`, a whole number. Thus the familiar declaration of a relation lets us join an object with numerical information about it.

Numbers are written without quotes. Here is the assertion from the scenario with three unit sides:

```

assert pleurai(entity_ref("urn:case:eucl:trigonon"), 1, 1, 1) {
  id "eucl-t10-1"; origin case_input;
}

```

First comes the reference to the object `urn:case:eucl:trigonon`, then the values 1, 1, and 1. The assertion's name `eucl-t10-1` and its origin `case_input` are already familiar.

This model assumes that all three lengths are expressed in one unit. The `Integer` values themselves store no units: from the record alone one cannot tell whether they were centimetres or metres, or check that the units were consistent. So convert all lengths to one unit before entering the numbers. For our questions, what matters are the ratios of the lengths, which stay the same when all lengths are scaled together. We will consider quantities with explicitly stated units in chapter 15.

Choose a common unit before entering lengths

`Integer` stores a whole number without a unit of length. The values 3, 4, 5 in this package must use the same unit to be comparable.

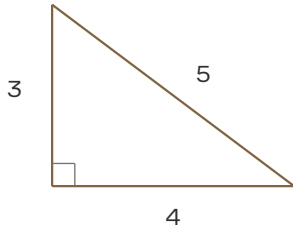
The line `key(t)` binds one object to one triple of side values. If two different triples are accepted for the same object, a key conflict arises; the later entry does not replace the first. Repeating the same triple does not by itself mean a difference of values. We will examine consistency checks in detail later; here each case will contain one assertion about the sides¹³.

¹³The Arxo specification, §§45–45.1: kinds of relations and key constraints. The declaration of `pleurai` is in the package vocabulary.

The code `grc` in the labels stands for Ancient Greek, and `ru` for Russian. In the explanations below, the tool uses the Greek labels. You do not need to read Greek to follow the explanation: the rule and entry names let us match each step against the code.

Equality marks the boundary

Sides 3, 4, 5

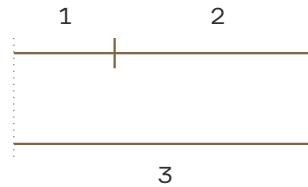


$$3 + 4 > 5; 4 + 5 > 3; 3 + 5 > 4$$

All three inequalities hold

On the right, overlapping segments are drawn at different heights so their lengths remain visible. Each figure uses its own scale.

Sides 1, 2, 3



$$1 + 2 = 3$$

The vertices lie on one straight line

First rule: do the sides form a triangle

Let us check each pair of sides. The sum of the first and second must be greater than the third, of the second and third greater than the first, of the first and third greater than the second. For the triple 1, 1, 1, all three checks come down to one true inequality: two is greater than one.

In the model this is recorded by the rule `TrigononEstin`:

```
@source(ST0_I20)
rule TrigononEstin strict {
  label grc official "Παντὸς τριγώνου αἱ δύο πλευραὶ τῆς λοιπῆς μείζονές εἰσι
  πάντῃ μεταλαμβανόμεναι";
  label ru unofficial "I.20: тройка составляет треугольник – каждые две сторон
  ы, взятые вместе, строго больше третьей";
  for t: Trigonon;
  for a: Integer;
  for b: Integer;
  for c: Integer;
  when pleurai(t, a, b, c) and (a + b > c) and (b + c > a) and (a + c > b);
  then trigonon_estin(t);
}
```

The rule has four variables. The line for `t`: `Trigonon` introduces the object, and the three lines with `Integer` introduce the numerical variables `a`, `b`, `c`. When the premise `pleurai(t, a, b, c)` is matched against our entry, all three receive the value 1.

The remaining conditions are comparisons. The sign `+` denotes addition, `>` means "strictly greater". Parentheses group each comparison, and `and` requires all conditions to hold, as in the rule about amphiproticity. Equality of the sum with the third side is not enough.

The conclusion `trigonon_estin(t)` reads: "the given sides form a triangle". In the vocabulary this relation is declared with the mark `kind derived`. That is how a property obtained by derivation is marked here. This mark and the familiar `institutional` are different kinds of relations; by themselves they neither change the meaning of the four states nor perform inference. In our case the ground is the application of a particular rule to the assertion about the sides.

The scenario with unit sides states a question and expectations:

```
evaluate truth(trigonon_estin(entity_ref("urn:case:eucl:trigonon")));
expect result_kind == PROPOSITION;
expect truth_status == TRUE_ONLY;
expect evaluation_status == COMPUTED;
```

This is an excerpt from the complete file. The new line `result_kind == PROPOSITION` checks the kind of result: the query returns a proposition whose support can be assessed. The line with `truth_status` checks its state, and `COMPUTED` checks that the calculation ran. These three checks concern different properties of the result.

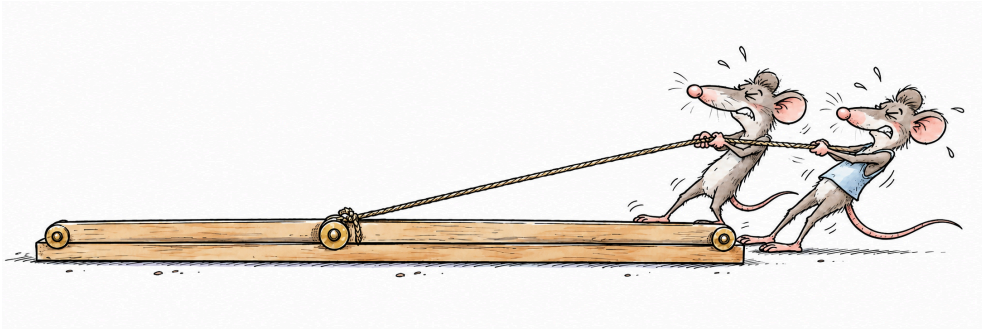
Run the supplied file from the reader companion directory:

```
law engine test \
  corpus/laws/doctrine/math/euclid-pythagoras/tests/10-trigonon-estin-111.lawtest \
  --program corpus/clir/grc-euclid-pythagoras.lawir.json --imports \
  contexts/grc-euclid-pythagoras.json
```

The check passes: `TRUE_ONLY`. Only the subject and the file paths changed; the procedure is the same. The file already contains the needed header and scenario context; for this trial they need no changes.

When the inequality fails

Take sides 1, 2, and 3. The first two together equal the third: the strict inequality fails. Geometrically this is a degenerate case: the segments can be laid along one straight line, but they do not form an ordinary triangle.



If the sum of two sides equals the third, the hinged frame stays flat. Effort does not change the ratio of the lengths.

The rule `TrigononEstin` does not apply. But the package contains a separate rule for concluding that the sides cannot form a triangle:

```
@source(STO_I20)
rule OudeTrigononPleuraiAB strict {
  label grc official "αἱ μὲν ΒΑ, ΑΓ τῆς ΒΓ";
  label ru unofficial "I.20 нарушено первой парой: первая и вторая стороны, взятые вместе, не больше третьей – тройка не составляет треугольника";
  for t: Trigonon;
  for a: Integer;
  for b: Integer;
  for c: Integer;
  when pleurai(t, a, b, c) and (a + b <= c);
  then oude_trigonon(t);
}
```

The sign `<=` means "less than or equal". It covers both a third side that is too long and the boundary case of equality. For our triple, $1 + 2 \leq 3$ holds, so the conclusion `oude_trigonon(t)` follows — "the sides do not form a triangle".

Note the form: `oude_trigonon` is the name of a separate relation. There is no construction then not `trigonon_estin(t)` in the rule. The negative sense of the label does not turn one relation into an explicit negation of the other. So the question about `oude_trigonon` receives `TRUE_ONLY`, while `trigonon_estin` on this case stays `NEITHER`.

A separate property and an explicit negation

`oude_trigonon(t)` and not `trigonon_estin(t)` are different statements. In this package, deriving `oude_trigonon(t)` does not create negative support for `trigonon_estin(t)`.

In the third chapter we already saw an explicit `not`. What matters here is recognizing the distinction in a real vocabulary: a separately named property and the negation of another property are linked only by the rules actually recorded. A similarity in meaning is not enough for such an inference.

For the other two pairs of sides the package contains analogous rules: `OudeTrigononPleuraiBC` and `OudeTrigononPleuraiAC`. They let us see which sum exactly failed to exceed the remaining side. For the triple 1, 2, 3, only the shown rule for the first and second sides fires.

Second rule: is there a right angle

Now return to sides 3, 4, and 5. They pass all three inequalities: 7 is greater than 5, 9 greater than 3, 8 greater than 4. For these sides, `trigonon_estin` is established. It can be used as a premise of the next rule:

```
@source(STO_I48)
rule OrthogonionProsTritei strict {
  label grc official "Ἐὰν τριγώνου τὸ ἀπὸ μιᾶς τῶν πλευρῶν τετράγωνον ἴσον ᾖ τῷ ἀπὸ τῶν λοιπῶν τοῦ τριγώνου δύο πλευρῶν τετραγώνοις, ἡ περιεχομένη γωνία ὑπὸ τῶν λοιπῶν τοῦ τριγώνου δύο πλευρῶν ὀρθή ἐστίν";
  label ru unofficial "I.48, угол против третьей стороны: квадрат третьей равен сумме квадратов первых двух – треугольник прямоугольный";
  for t: Trigonon;
  for a: Integer;
  for b: Integer;
  for c: Integer;
  when trigonon_estin(t) and pleurai(t, a, b, c)
    and (a * a + b * b == c * c);
  then orthogonion(t);
}
```

Start with the familiar parts. The variable `t` denotes the same object, and `a`, `b`, `c` receive the lengths from `pleurai`. The premise `trigonon_estin(t)` requires the established existence of a triangle with these sides.

The arithmetical condition uses multiplication `*` and an equality check `==`. The expression `a * a` is the square of the first length. For the triple 3, 4, 5 we get `9 + 16 == 25`. All conditions hold, and the rule derives `orthogonion(t)`: the triangle is right-angled.

This rule checks the case in which the square of the third side equals the sum of the other two squares. Two more rules check the corresponding equalities for the first and second sides. So the package does not require the longest side always to be recorded last¹⁴.

Read the explanation of the supplied case with sides 3, 4, 5:

¹⁴The triangle-existence rules and the three right-angle rules. Each right-angle rule requires `trigonon_estin(t)`.

```
chapter5_explain=$(mktemp -d)
python3 reader_tools/prepare_question.py corpus/laws/doctrine/math/euclid-pythagoras/tests/01-aigyptiako-345.lawtest \
  --program corpus/clir/grc-euclid-pythagoras.lawir.json --imports \
  contexts/grc-euclid-pythagoras.json --out "$chapter5_explain"
law engine explain "$chapter5_explain"
```

Below is the output with shortened rule-application identifiers:

```
ὀρθογώνιον ἔστι τὸ τρίγωνον(trigonon): TRUE_ONLY
rule_application ὀρθογώνιον ἔστι τὸ τρίγωνον(trigonon)
(urn:proof:apply:OrthogonionProsTritei:5afa6b14...)
rule_application τρίγωνόν ἐστιν(trigonon)
(urn:proof:apply:TrigononEstin:6c0e0a77...)
assertion αἱ τρεῖς πλευραὶ τοῦ τριγώνου(trigonon, 3, 4, 5)
(urn:proof:assert:urn:grc:eukleides:clir:pythagoras#eucl-t01-1)
assertion αἱ τρεῖς πλευραὶ τοῦ τριγώνου(trigonon, 3, 4, 5) ↑ раскрыт выше
(urn:proof:assert:urn:grc:eukleides:clir:pythagoras#eucl-t01-1)
```

Author's paraphrase of the engine output above: the triangle is right-angled (TRUE_ONLY) by applying the rule OrthogonionProsTritei, which itself rests on an application of TrigononEstin — both grounded in the entry eucl-t01-1 about sides 3, 4, and 5.

Under OrthogonionProsTritei we see the application of TrigononEstin, and under it the source assertion eucl-t01-1 about the sides.

Now one rule's conclusion serves as another's premise.

The same assertion about the sides is also needed for checking the squares; the mark "раскрыт выше" ("expanded above") shows a repeated reference to it.

The arithmetical comparisons themselves are not written out as separate lines in this textual explanation. They can be checked in the rule's condition by substituting the numbers from the source entry. So the explanation and the code complement each other.

The equality holds, but there is no triangle

Take sides 0, 3, and 3. The arithmetic checks out: zero squared plus nine equals nine. But the sum of the zero side with either of the other two is not greater than the remaining one. The triple fails the triangle-existence condition.

The check set contains a scenario with two questions about this case:

```

test "EUCL-16-GATE-033" {
  given {
    context {
      decision_time @2026-08-26T12:00:00+05:00;
      knowledge_time @2026-08-26T12:00:00+05:00;
      legal_time @2026-08-31;
      timezone "Asia/Qyzylorda";
    }

    assert pleurai(entity_ref("urn:case:eucl:trigonon"), 0, 3, 3) {
      id "assert-sides";
      origin case_input;
    }
  }

  evaluate truth(orthogonion(entity_ref("urn:case:eucl:trigonon")));
  expect truth_status == NEITHER;
  expect issues_within(info);

  evaluate truth(oude_trigonon(entity_ref("urn:case:eucl:trigonon")));
  expect truth_status == TRUE_ONLY;
  expect issues_within(info);
}

```

Each evaluate starts a new question, and the expect lines following it state expectations for that question only. The two questions share the same source information. The dates belong to the context of the test case and do not mark the time the *Elements* were composed.

The expectation `issues_within(info)` bounds the severity of diagnostic messages: there must be no messages above the informational level. It checks how the evaluation ran, and does not replace stating the expected answer status. We will examine violations of requirements in the next chapter.

Run the complete set containing this scenario as follows:

```

law engine test corpus/laws/doctrine/math/euclid-pythagoras/tests/pythagoras.law
test \
  --program corpus/clir/grc-euclid-pythagoras.lawir.json --imports \
  contexts/grc-euclid-pythagoras.json

```

The two answers differ:

Question about sides 0, 3, 3	Answer
Is right-angledness established (orthogonion)?	NEITHER
Is it established that the sides do not form a triangle (oude_trigonon)?	TRUE_ONLY

The first answer does not mean "the triangle is not right-angled". The right-angle rule's applicability condition is not met: `trigonon_estin` is not established. The second answer has grounds of its own: the inequalities for two pairs of sides are violated.

The equality of the squares does not make up for the missing premise.

To see the reason for the first answer, use the separate supplied file with one question about the same triple:

```
chapter5_why_not=$(mktemp -d)
python3 reader_tools/prepare_question.py corpus/laws/doctrine/math/euclid-pythagoras/tests/05-pyle-033.lawtest \
  --program corpus/clir/grc-euclid-pythagoras.lawir.json --imports \
  contexts/grc-euclid-pythagoras.json --out "$chapter5_why_not"
law engine why-not "$chapter5_why_not"
```

In the analysis, three rules about the right angle are named. In each, find the premise `τρίγωνόν ἐστιν(trigonon)` with the status `NEITHER`: this is the Greek label of `trigonon_estin`. The text also contains messages about side variables not linked to the query's goal. They belong to the rule's analysis and do not mean that the equality of the squares was computed and found false. Here, the absence of the needed premise can be checked directly against the inequalities of the sides.

Two steps: existence, then the right angle

Sides 3, 4, 5

All three inequalities hold



trigonon_estin: TRUE_ONLY



$3^2 + 4^2 = 5^2$ and the premise is established



orthogonion: TRUE_ONLY

Sides 0, 3, 3

$0 + 3 = 3$: strictness violated



trigonon_estin: NEITHER



$0^2 + 3^2 = 3^2$, but the premise is not established



orthogonion: NEITHER

Shown here is the course of checking the conditions against the model's code. For 0, 3, 3, oude_trigonon: TRUE_ONLY is derived separately; this is not an explicit negation of trigonon_estin.

Try it yourself

Work on copies of the supplied files. Replace the scenario file's path with your copy's path. For test, it follows the command name; in the explain and why-not sequences, it follows the preparation script name. The prepared model after --program stays the same. Try all three exercises before reading the discussion below.

1. Copy pythagoras.lawtest, keeping the three header lines and only the scenario EUCL-16-GATE-033. Replace sides 0, 3, 3 with 5, 12, 13. First keep the old expectations and predict both answers. After running, explain the mismatches and correct the expectations for the new case.
2. In a copy of 02-ekfylismenon-123.lawtest, replace sides 1, 2, 3 with 2, 3, 5. This file asks about oude_trigonon. Name the rule that must apply, and check the old TRUE_ONLY expectation. Confirm the ground via explain.
3. In a copy of 01-aigyptiako-345.lawtest, reorder the sides: instead of 3, 4, 5 record 5, 3, 4. Will the answer about right-angledness change? Which rule must now appear in the explanation instead of OrthogonionProsTritei?

Check your predictions

In the first exercise, right-angledness will receive TRUE_ONLY, and the "not a triangle" assertion NEITHER. The status expectations must be swapped; the issues_within(info) checks stay. In the second, OudeTrigononPleuraiAB will apply: the sum of 2 and 3 equals 5. In the third, the answer will stay TRUE_ONLY, and the right-angle test will apply OrthogonionProsProtei, for the first side. The changed numbers explain the new results; the rules are unchanged.

We added numerical arguments, comparisons, and a chain of two rules to the familiar way of recording. The main distinction stayed the same: to obtain a conclusion, all its premises must be established. Naming the object a triangle in the vocabulary, or checking the equality of the squares, is still not enough.

Next we will consider another problem: how to record a requirement on the consistency of information, and how to tell its violation apart from deriving a new property.

CHAPTER 06

Derive a Consequence and Check a Requirement

In the last chapter the rules let us first establish a triangle's existence, then tell whether it has a right angle. Now ask another question: are the derived properties consistent with each other? If one and the same figure is recognized as both right-angled and obtuse-angled, the fault lies in the model or in the source information.

Arxo provides a constraint for such a check. It tests information against a requirement and reports the result without deriving a new assertion. We will examine it in the same Euclid package, first with the original rules, then on a copy with a deliberate error.

Which properties we will compare

Besides the familiar orthogonion, the vocabulary contains two more relations: amblygonion means "the triangle is obtuse-angled", and oxygonion means "the triangle is acute-angled". Each relation, like orthogonion, takes a single argument of type `Trigonon`.

The rules recognize these properties by comparing the squares of the sides. For 3, 4, 5 we get a right-angled triangle. For 2, 3, 4 the sum of the squares of the smaller sides is 13, and the square of the largest is 16: the package derives obtuseness. For the triple 1, 1, 1 each square is less than the sum of the other two; acuteness follows.

The rules about the acute and obtuse angles refer to propositions II.12, II.13, and definition I.21. The package separately explains how the numerical tests were obtained from these fragments. They must not be mistaken for Euclid's verbatim converse propositions¹⁵. What matters for us is matching the results: an ordinary triangle belongs to one of three kinds, and the kinds exclude each other.

A requirement that derives nothing

Start with a pair of properties. If a triangle is recognized as right-angled, it must not at the same time be obtuse-angled. This requirement is recorded in the package as follows:

¹⁵The classification and constraints module: the opening comment accounts for deriving the numerical tests and the completeness role of the three kinds. Euclid's fragments are pinned in [sources.law](#). The English renderings of the propositions in this chapter are the translator's own.

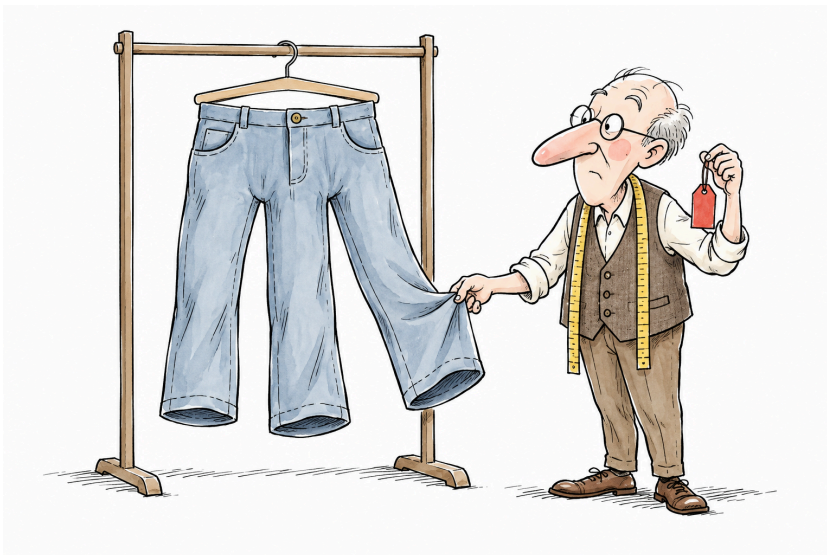

```

@source(STO_DEF21)
constraint OrthogonionKaiAmblygonionAsymbata {
  label grc unofficial "ἀδύνατον τὸ αὐτὸ τρίγωνον ὀρθογώνιον τε εἶναι καὶ ἀμβλ
υγώνιον";
  label ru unofficial "инвариант §93.1: прямоугольный и тупоугольный – разные
роды (определение I.21), и один треугольник не может быть обоими";
  for t: Trigonon;
  when orthogonion(t);
  require not amblygonion(t);
  severity error;
  message "треугольник признан и прямоугольным (I.48), и тупоугольным (II.12):
при какой-то позиции квадрат стороны оказался и равен сумме квадратов двух други
х, и больше её";
}

```

After the constraint keyword stands the constraint's name. The variable *t* and the when condition are familiar from rules: we check the properties of one object for which right-angledness is established. Instead of a then conclusion, a require clause is recorded here.

The line `require not amblygonion(t)` requires the negation of obtuseness. It does not itself create such a negative assertion. If information for the check is lacking, the constraint may remain undetermined. But if obtuseness is established, and there are no grounds against it, the requirement is violated.



The check marks the mismatch. Correction requires a separate action.

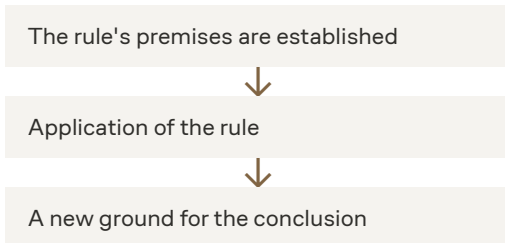
`severity error` sets the diagnostic message's level for a violation: an error. The message field holds an explanation that shows which properties proved incompat-

ible. The words inside the message neither correct the source information nor change the rules.

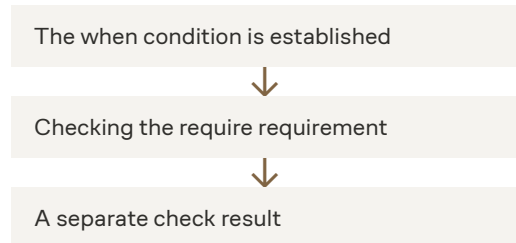
Here the difference between the two constructs becomes visible. A rule adds a ground for its conclusion. A constraint checks a requirement against information already obtained, and stores the result of that check separately.

Deriving a conclusion and checking a requirement

rule



constraint



This diagram shows a constraint whose when condition has TRUE_ONLY support. A constraint neither adds support to its requirement nor corrects the input.

At least one of the three kinds

Another constraint checks the completeness of the classification: if the sides form a triangle, at least one of the three kinds must be established.

```

@source(STO_DEF21)
constraint TrichotomiaPliris {
  label grc unofficial "πᾶν τρίγωνον ἢ ὀρθογώνιον ἢ ἀμβλυγώνιον ἢ ὀξυγώνιον ἐστίν";
  label ru unofficial "инвариант §93.1, ПОЛНОТА: всякий треугольник принадлежит одному из трёх родов – прямоугольному (I.48), тупоугольному (II.12) или остроугольному (II.13); четвертого рода определение I.21 не знает";
  for t: Trigonon;
  when trigonon_estin(t);
  require orthogonion(t) or amblygonion(t) or oxygonion(t);
  severity error;
  message "тройка составляет треугольник, но не отнесена ни к одному из трёх родов: перечень I.21 перестал быть исчерпывающим, и обратные к II.12/II.13 лишились основания";
}
  
```

Here the requirement uses `or` for the first time. In the simple case without contradictions, establishing at least one of the listed properties suffices. The `or` itself does not require exactly one variant to hold. Other constraints cover the kinds' incompatibility, including the one considered above.

At least one and no more than one
Completeness requires at least one kind. Pairwise incompatibility forbids combinations of kinds. Only together do these requirements express “exactly one”.

The condition when `trigonon_estin(t)` is also essential. The triple 0, 3, 3 from the last chapter is not recognized as a triangle, so this requirement does not apply to it. The absence of a classification for such a triple does not violate the completeness of the triangle-kinds list.

No error does not mean every requirement is met

In the earlier scenarios we have already met the check:

```
expect issues_within(info);
```

It admits messages no higher than the informational level. This does not require the message list to be empty, nor does it confirm that every constraint is met. To see the latter, one must look at the result of the constraint's check itself.

When the `when` condition is established without contradiction, four outcomes are possible:

Requirement state	Check result
TRUE_ONLY	SATISFIED — met
FALSE_ONLY	VIOLATED — violated
BOTH	CONFLICTED — information contradictory
NEITHER	UNDETERMINED — grounds insufficient

These are the familiar four states applied to a requirement. The condition under which it is checked must be considered separately: the table concerns an established `when`¹⁶.

Look at triangle 3, 4, 5 in the source package. Right-angledness is established, so the completeness requirement `TrichotomiaPliris` is met: one of the three kinds is found. But `amblygonion` on this case has grounds neither for nor against. Its negation also has the state `NEITHER`. The constraint `OrthogonionKaiAmblygonionAsymbata` receives `UNDETERMINED`.

¹⁶The Arxo specification, §§93.1–93.3: constraints, outcomes, and diagnostics. With a `TRUE_ONLY` condition, an unknown requirement yields `UNDETERMINED`; it produces no error message.

An unknown negation stays unknown

If amblygonion has NEITHER, then not amblygonion also has NEITHER. The applicable constraint therefore receives UNDETERMINED; no negative ground arises on its own.

No contradiction is found here, but the negation of obtuseness is not established from this record. The constraint itself does not derive it. In the current execution UNDETERMINED produces no error message, so `issues_within(info)` passes.

The absence of a reported error does not mean that the requirement is satisfied.

To see these results, one can print the explanation graph of the supplied scenario:

```
chapter6_graph=$(mktemp -d)
python3 reader_tools/prepare_question.py corpus/laws/doctrine/math/euclid-pythagoras/
tests/01-aigyptiako-345.lawtest \
  --program corpus/clir/grc-euclid-pythagoras.lawir.json --imports \
  contexts/grc-euclid-pythagoras.json --out "$chapter6_graph"
law engine graph "$chapter6_graph" --json
```

The graph command shows nodes and links, and `--json` selects structured output. Search it for the constraint names, then find their `constraint_check` nodes. Each label contains the constraint's name, the check result status, and the requirement's `requirementStatus`. For `OrthogonionKaiAmblygonionAsymbata`, these are UNDETERMINED and NEITHER; for `TrichotomiaPliris`, they are SATISFIED and TRUE_ONLY. The rest of the graph can wait.

Prepare one check

To see a violation, change the rule on a separate copy of the package. We will introduce a deliberate error for the exercise: let the right-angledness test fire not only on the equality of the squares, but also when one square exceeds the sum of the other two.

Run all commands below from the reader companion directory in one terminal window: later commands use variables set by earlier ones. Create a temporary directory and copy the package and the supplied scenario about sides 2, 3, 4 into it:

```
chapter6_work=$(mktemp -d)
cp -R corpus/laws/doctrine/math/euclid-pythagoras "$chapter6_work/euclid-pythago
ras"
cp corpus/laws/doctrine/math/euclid-pythagoras/tests/12-amblygonion-234.lawtest \
  "$chapter6_work/case234.lawtest"
```

The variable `chapter6_work` stores the working copy's path. Open `case234.lawtest` in the created directory. Keep the header, context, and assertion about the sides as they are. Replace the question about obtuseness with a question about right-angledness and replace its expectations with the following. This is a change made for the exercise, not a verbatim excerpt from the corpus:

```
evaluate truth(grc.euclid.stoicheia_i::orthogonion(entity_ref("urn:case:eucl:tri
gonon")));
expect result_kind == PROPOSITION;
expect truth_status == NEITHER;
expect evaluation_status == COMPUTED;
expect issues_within(info);
```

For sides 2, 3, 4 the equality of the squares fails, so we expect exactly **NEITHER** for orthogonion. A separate relation about non-right-angledness in this package does not automatically create an explicit negation of orthogonion — the same manner of recording we examined in the last chapter.

First check the case against the original model:

```
law engine test "$chapter6_work/case234.lawtest" --program \
  corpus/clir/grc-euclid-pythagoras.lawir.json --imports \
  contexts/grc-euclid-pythagoras.json
```

All expectations hold. Now we have a baseline result against which to compare the effect of changing the rule.

Change the condition on the copy

In the working copy of the package, open the file `03-protasis-48.law`. In the rule `OrthogonionProsTritei` find the line:

```
and (a * a + b * b == c * c);
```

Replace only that line with the following:

```
and (c * c >= a * a + b * b);
```

The sign `>=` means "greater than or equal". The side `c` is moved to the left, so now the rule also covers the case in which its square exceeds the sum of the other two squares. That is exactly how things stand for the triple 2, 3, 4: 16 is greater than 13.

Until now we used the supplied model from `corpus/clir`. Editing a source rule does not change that prepared model: we must rebuild the copy. Refresh the copy's compiled program and lock, then use `pack` to link its pinned dependencies. The final

copy command selects the complete compiled model, including its dependencies, from that release:

```
mkdir -p "$chapter6_work/clir"
cp clir/grc-euclid-stoicheia-i.lawir.json "$chapter6_work/clir/"
law engine lower "$chapter6_work/euclid-pythagoras" \
  --imports contexts/grc-euclid-pythagoras.json \
  > "$chapter6_work/clir/grc-euclid-pythagoras.lawir.json"
law engine lock "$chapter6_work/euclid-pythagoras" --clir "$chapter6_work/clir" \
  --write
law pack --project "$chapter6_work/euclid-pythagoras" \
  --out "$chapter6_work/modified-release" --json
cp "$chapter6_work/modified-release/world.lawir.json" \
  "$chapter6_work/modified.lawir.json"
law engine test "$chapter6_work/case234.lawtest" --program \
  "$chapter6_work/modified.lawir.json" --imports contexts/grc-euclid-pythagoras.
json
```

Three mismatches will appear in the check's output:

```
truth_status == NEITHER: в документе TRUE_ONLY
evaluation_status == COMPUTED: в документе NON_EXECUTABLE
issues_within(info): в документе CONSTRAINT_VIOLATED/error
```

Author's paraphrase of the engine output above: instead of the expected NEITHER, the document gives TRUE_ONLY; instead of COMPUTED, NON_EXECUTABLE; and instead of messages within info, a CONSTRAINT_VIOLATED message of level error.

The first concerns the assertion: the changed rule gave orthogonion positive support, so TRUE_ONLY resulted instead of NEITHER. The second concerns execution: instead of COMPUTED, NON_EXECUTABLE was received. The third reports a constraint violation of level error.

We left the rules about obtuseness and non-right-angledness unchanged. They still apply to sides 2, 3, 4. So the mistaken right-angledness leads to two violations:

Constraint	Which properties turned out established together
OrthogonionKaiAmblygonionAsymbata	right-angledness and obtuseness
OrthogonionKaiOukOrthogonionAsymbata	right-angledness and non-right-angledness

To find them in the graph for this same case, run:

```
chapter6_graph=$(mktemp -d)
python3 reader_tools/prepare_question.py "$chapter6_work/case234.lawtest" \
  --program "$chapter6_work/modified.lawir.json" --imports \
  contexts/grc-euclid-pythagoras.json --out "$chapter6_graph"
law engine graph "$chapter6_graph" --json
```

For the two named `constraint_check` nodes, `status` will carry the value `VIOLATED`. All assertions concern one object, `urn:case:eucl:trigonon`, for which the case states sides 2, 3, 4. The constraints found the results for this object incompatible. They do not say in which line we made the error, nor do they correct it.

The same sides, different results

What we read	The source rule	The changed rule
orthogonion	NEITHER	TRUE_ONLY
The two incompatibility constraints	Do not apply	VIOLATED
evaluation_status	COMPUTED	NON_EXECUTABLE
Constraint error	None	CONSTRAINT_VIOLATED

In both cases the sides are 2, 3, 4. Two constraints named above are compared. The changed rule added the support for right-angledness; the constraints did not remove it.

Why one derivation's explanation is not enough

The changed model can still show the path to right-angledness: the assertion about the sides, the existence rule's application, the changed test's application. That path helps reconstruct the derivation's course, but it does not certify the recorded condition's correctness.

An error-level violation is also reflected in the execution status¹⁷. `NON_EXECUTABLE` in our trial does not mean that no step was computed: the grounds and the check results are present. It means that, because of a blocking error, the result did not receive the successful-execution status `COMPUTED`. Reading one `TRUE_ONLY` while ignoring this status would be an error.

A constraint can find a problem not only in a rule. If, under the source rules, right-angledness for sides 2, 3, 4 is recorded directly, the incompatibility will also arise.

¹⁷The Arxo specification, §181.2: result under a blocking diagnostic. An error-level error blocks `COMPUTED`; for a constraint violation `NON_EXECUTABLE` is used here.

Both the rules and the source information must be checked. In this trial the cause is known in advance, because we changed one line ourselves.

Even satisfied constraints do not certify the whole model: they check only the stated requirements. In our problem this is a useful independent check of the consistency of several conclusions about one subject.

«Program testing can be used to show the presence of bugs, but never to show their absence!»

Edsger W. Dijkstra · Notes on Structured Programming · EWD249, p. 7
The edition's English rendering; consult the source for its wording.

The quotation concerns program testing. In our trial, constraints check the stated requirements; a successful result does not certify the whole model.

Try it yourself

1. Read `TrichotomiaPliris` in ordinary words. Explain why this constraint can be met even when the triangle is assigned two kinds. Which constraints must find such an incompatibility?
2. In the working copy, restore the original line with `==`. Repeat the engine `lock`, `pack`, and `copy` commands above to rebuild `modified.lawir.json`, then run the same `case234.lawtest` against it. Predict which three mismatches will vanish, and check the result.
3. Compare the case with sides 0, 3, 3 from the last chapter with our changed model for 2, 3, 4. In which case does the rule lack an established premise, and in which did the rule apply and the resulting properties violate a requirement? Name the assertion's state and the execution status separately.

Check your predictions

In the first exercise, at least one established kind suffices; the completeness of the list alone does not check mutual exclusion. In the second, after the rule is restored, `NEITHER`, `COMPUTED`, and no messages above level `info` will result again: the source check will pass. In the third, for sides 0, 3, 3 right-angledness has `NEITHER` with `COMPUTED`, while for the deliberately changed model for 2, 3, 4 it has `TRUE_ONLY` with `NON_EXECUTABLE`.

Now we can tell apart deriving a property, checking a requirement, and the success of execution. An undetermined requirement returns us to the question of completeness of information: when may the absence of an entry be taken to say something about the subject? That is the subject of the next chapter.

CHAPTER 07

Do We Have All the Information?

In the last chapter a requirement could remain undetermined: no negative ground was available to confirm it. In the first examples the absence of an entry also left the question open. But sometimes we want to report not only separate pieces of information, but also that the supplied list is complete. Then a gap in it gains meaning.

Picture a list of points on a straight line. If it names A and B, nothing is yet known about point C. Now make the conditions more precise: C belongs to the drawing under consideration, and the list of points in that drawing that lie on this line is declared complete. Then C's absence from the list becomes a ground for a negative answer. It matters to state exactly to which relation and which objects the word "all" refers.

«Such information is especially important in OWL, where by default everything is considered possible until you state the contrary.»

W3C · OWL 2 Web Ontology Language Primer · Second Edition, §4.4

The edition's English rendering; consult the source for its wording. © 2012 W3C. Commentary for this software tutorial; the document's own licence applies.

Here the sense of missing information is compared. The semantics of OWL and Arxo differ; Arxo's local closure is set by a separate completeness declaration.

What the drawing contains

Consider the package on David Hilbert's *Foundations of Geometry*. Its source is the pinned English translation¹⁸. The objects of the first examples are points and lines, and the source information reports which of them belong to the presented configuration and which points lie on which lines.

One feature of this package needs explaining before we begin. In our scenarios, separate answers about whether a point lies on a line are computed, but the drawing's description violates some of the axioms' requirements. In the reviewed public runtime, these incidence queries receive `COMPUTED` together with diagnostics of violated constraints. The violations concern requirements outside the chain of facts and rules on which these queries depend, so they do not block these answers.

¹⁸Hilbert's source and pinned translation. The relations, the domain rule, and the closure are in the package vocabulary. The English label `A lies upon a` is the package's pinned wording, quoted verbatim in code; the surrounding prose renderings are the translator's own.

This differs from Chapter 6, where the violated constraints directly concern the queried property. We will examine the reasons for the violations after introducing closure: a declaration that specified information is complete within a defined domain.¹⁹

The vocabulary calls the point type `Punkt`, the line type `Gerade`. Belonging to the configuration is recorded by two relations:

```
relation punkt_der_figur(p: Punkt) kind empirical {
  label ru unofficial "точка входит в предъявленную конфигурацию";
}
```

```
relation gerade_der_figur(l: Gerade) kind empirical {
  label ru unofficial "прямая входит в предъявленную конфигурацию";
}
```

Each has one argument of the corresponding type. The link between a point and a line is recorded by a separate relation:

```
@source(GDG_I_1)
relation liegt_auf(p: Punkt, l: Gerade) kind empirical {
  label en official "A lies upon a";
  label ru unofficial "точка лежит на прямой";
}
```

In `liegt_auf(p, l)` the point stands first, then the line. This is the familiar two-argument form, but now the arguments have different types. The German names here are the vocabulary's identifiers. The Russian labels describe the same three relations explained above: a point belongs to the configuration, a line belongs to it, and a point lies on a line.

Scenario 01 records three points A, B, C and three lines ab, bc, ac. For each point and line there is an entry about belonging to the configuration. Positive assertions that a point lies on a line look like this:

Point	Line ab	Line bc	Line ac
A	lies	—	lies
B	lies	lies	—
C	—	lies	lies

¹⁹The runtime migration note records the reviewed CLI, the changed execution-status expectations and their checks. A diagnostic about another requirement does not make every answer non-executable; an answer depending on a violated blocking constraint still receives `NON_EXECUTABLE`.

The dash so far means only the absence of a positive entry. The source case contains no `liegt_auf` assertion for these pairs. The table presents six positive `liegt_auf` entries. The complete scenario's other entries list the objects and give them names via `punktname` and `geradenname`.

Where the negative answer comes from

Ask whether C lies on ab. The query and the first expectations in scenario 01 read:

```
evaluate truth(liegt_auf(entity_ref("urn:de:hilbert:case:point:C"), entity_ref("urn:de:hilbert:case:line:ab")));
expect result_kind == PROPOSITION;
expect truth_status == FALSE_ONLY;
```

`FALSE_ONLY` is expected, though the case contains no negative entry. To see the ground, run from the reader companion directory:

```
chapter7_explain=$(mktemp -d)
python3 reader_tools/prepare_question.py corpus/laws/doctrine/math/hilbert-grundlagen/tests/01-gerade-meidet-punkt.lawtest \
  --program corpus/clir/de-hilbert-grundlagen.lawir.json --imports \
  contexts/de-hilbert-grundlagen.json --out "$chapter7_explain"
law engine explain "$chapter7_explain"
```

The result:

```
A lies upon a(C, ab): FALSE_ONLY
assertion ~ A lies upon a(C, ab)
(urn:proof:closure:urn:de:hilbert:clir:grundlagen#InzidenzAbschluss:2)
```

Author's paraphrase of the engine output above: C's lying on ab is refuted (`FALSE_ONLY`), on the ground of an explicit negation derived by the closure `InzidenzAbschluss` — not on a case entry of its own.

The output uses the English label `A lies upon a` with the particular arguments `C` and `ab`. The sign `~` denotes negation. The ground's identifier contains `proof:closure`, then the name `InzidenzAbschluss`. This is a negative ground obtained by closure, not an assertion with a name of its own from the source case.

A closure declaration states that the information at hand counts as complete for a specified relation in a specified domain. For our relation it is declared as follows:

```

closure InzidenzAbschluss {
  label ru unofficial "замыкание §70: точка, не записанная лежащей на прямой,
на ней не лежит";
  predicate liegt_auf;
  domain punkt_gerade_paar;
  snapshot "urn:de:hilbert:snapshot:inzidenz:1899";
  complete_as_of @1899-06-17T00:00:00Z;
  derive_explicit_negative true;
}

```

The word `closure` introduces the declaration, and `InzidenzAbschluss` gives its name. The field `predicate` `liegt_auf` names the relation the completeness concerns. The declaration does not automatically extend to all information in the package²⁰.

`domain` `punkt_gerade_paar` defines the domain of pairs inside which the closure acts. If such a pair belongs to the domain, and there is no positive ground for `liegt_auf`, the closure may yield a negative ground. In this package, positive grounds for `liegt_auf` come from source entries; in general, positive grounds derived by rules must also be taken into account.

The field `derive_explicit_negative` `true` allows deriving an explicit negation. The value `true` enables it. The obtained ground is visible in the explanation, so the negation can be distinguished from an entry written directly into the case.

The `snapshot` field holds the name of the snapshot to which the completeness declaration refers, and `complete_as_of` sets its time. These are different fields: an identifier and a time. The year 1899 in this entry is part of the model's declaration. It does not by itself prove that our study drawing was composed then, or that the list truly contains no gaps. We will examine time in detail in chapter 13.

Which information is complete, and as of when?

`snapshot` names the set of information; `complete_as_of` sets its moment of completeness. These fields record the model author's declaration but do not prove the completeness of the list itself.

²⁰The Arxo specification, §70: *local declarations of completeness*. The domain must be finite and exactly stated; the absence of positive support is checked for the tuples belonging to it.

Why closure yields a negation

- 1 **Completeness declared**
InzidenzAbschluss is set for `liegt_auf`; obtaining a negation is enabled.
- 2 **The pair belongs to the domain**
C and ab belong to the configuration: `punkt_gerade_paar(C, ab)` is established.
- 3 **No positive support**
Neither an entry nor a rule has established `liegt_auf(C, ab)`.

`not liegt_auf(C, ab)`

FALSE_ONLY

In scenario 01 the negative ground has the form `proof:closure`. Without the needed declaration, or outside its domain, such a negation does not arise.

For which pairs the list counts as complete

Look at what `punkt_gerade_paar` means. In the vocabulary it is a relation with two arguments, a point and a line. A small rule derives it:

```
rule PunktGeradePaare strict {
  label ru unofficial "область инцидентности: точка и прямая взяты из конфигурации";
  for p: Punkt; for l: Gerade;
  when punkt_der_figur(p) and gerade_der_figur(l);
  then punkt_gerade_paar(p, l);
}
```

Read it in ordinary words: if a point belongs to the configuration and a line belongs to the configuration, their pair belongs to the closure's domain. For our three points and three lines this yields nine pairs — every cell of the table.

In six cells there is a positive `liegt_auf` entry. For the remaining three pairs the closure yields a negation. In particular, C and ab belong to the domain, while there is no positive ground for that incidence. That is how **FALSE_ONLY** results, with the closure's name in the explanation.

Closure applies to this set of pairs

	ab	bc	ac
A	+	−	+
B	+	+	−
C	−	+	+

D: outside the configuration Pairs with D do not belong to this domain

+ : TRUE_ONLY — a positive entry
− : FALSE_ONLY — a ground by closure

Scenario 04: 3 points × 3 lines = 9 pairs. For the question about D and ab the answer is NEITHER: there are no other grounds for or against in the case.

The bare listing of three points does not mean that no other points exist at all. What is declared complete is the `liegt_auf` relation for the pairs of this configuration. Such a declaration does not settle the broader question about all points of geometry.



A complete list of fish in an aquarium does not become a complete list of fish in the sea. The domain of completeness must be named.

A point outside the list

In scenario 04 the source drawing is the same, but the question is asked about point D and line ab:

```
evaluate truth(liegt_auf(entity_ref("urn:de:hilbert:case:point:D"), entity_ref("urn:de:hilbert:case:line:ab")));
expect result_kind == PROPOSITION;
expect truth_status == NEITHER;
```

Point D is not declared as belonging to the configuration. So the rule PunktGeradePaare does not create the pair of D and ab, and the closure does not apply to it. There is neither a positive entry that D lies on ab nor a negative ground. The answer is NEITHER.

Scenario 05 takes the next step. In it, D is added to the configuration; the entry essential for the domain looks like this:

```
assert punkt_der_figur(entity_ref("urn:de:hilbert:case:point:D")) { id "hilb-t05-d1"; origin case_input; }
```

The complete file also contains the point's label via punktname. What is essential for bringing the pair into the domain is exactly the entry punkt_der_figur: line ab already belongs to the configuration. No information that D lies on ab has been added yet. Now the closure covers this pair and yields FALSE_ONLY.

Compare the three cases:

Case	Pair belongs to the closure domain	Positive liegt_auf entry	Answer
C and ab, scenario 01	yes	no	FALSE_ONLY
D and ab, scenario 04	no	no	NEITHER
D and ab, scenario 05	yes	no	FALSE_ONLY

The transition from the second row to the third is caused by new information: we brought D into the configuration for which completeness is already declared.

That changed what the missing entry means. Adding an object to the configuration brings new pairs into the closure's domain. Do this deliberately: it can create negative conclusions about those pairs.

If instead the case states that D lies on ab, this pair will gain a positive ground. On a new run the closure will not create a negation from lack of support, since support exists. On that run the answer will become TRUE_ONLY.

What the scenario check confirms

Run the supplied scenario 01:

```
law engine test \
  corpus/laws/doctrine/math/hilbert-grundlagen/tests/01-gerade-meidet-punkt.lawt
est \
  --program corpus/clir/de-hilbert-grundlagen.lawir.json --imports \
  contexts/de-hilbert-grundlagen.json
```

The check passes²¹. Besides the answer's state, the file states three more expectations:

```
expect evaluation_status == COMPUTED;
expect issue(CONSTRAINT_VIOLATED);
expect issues_within(info, CONSTRAINT_VIOLATED);
```

The first requires exactly `COMPUTED` for this incidence query. The second, `issue(CONSTRAINT_VIOLATED)`, checks that a message about a violated constraint is present. The third admits messages above level `info` only with the named code. It does not bound the number of such messages, nor confirm that no other undetermined requirements remain.

Thus test `PASS` means the result matches the recorded expectations. It does not mean that the drawing satisfies all the axioms. In this scenario the violation is expected and forms part of the checked result. For scenarios 04 and 05 the same expectations of execution status and diagnostic messages are used.

An expected violation is also checked

`PASS` confirms the recorded expectations. An independent incidence query can receive `COMPUTED` together with `CONSTRAINT_VIOLATED` diagnostics; this does not certify that the whole configuration satisfies the axioms.

A complete list is not yet a model of geometry

Which requirements does the configuration violate? The description contains a triangular configuration of three points, but no shared plane is named. Under the package's closures, the absence of the required plane violates the constraint `AxiomI3`.

The constraints `AxiomII2Innen` and `AxiomII2Fortsetzung` check for an intermediate point and a continuation past a segment's end. Our list contains no corresponding information, and the closures turn their absence into negative grounds. These requirements are violated as well.

²¹Historical scenarios 01: C and ab, 04: D outside the configuration, and 05: D included. The runtime companion preserves these files and records their migration. Its copies expect `COMPUTED` for the independent incidence query while retaining the original truth-state and constraint-diagnostic expectations.

An ordinary finite drawing and a complete geometrical model are different subjects. In picturing a line, we do not draw all its points one by one. But in the presented finite description the package considers the listed objects and the explicitly stated domains of completeness. The axioms' conditions may require objects and links that this description does not supply.

So the negative answer about C and ab must be read together with the execution status. It shows the consequence of declaring completeness for the given pair. The violated constraints show that the configuration as a whole does not meet the checked requirements. The incidence answer and the constraint diagnostics must both be read.

Other requirements may remain undetermined. In I.7, for instance, the presence of three non-collinear points in a plane and four non-coplanar points in space is checked. If only a plane or a spatial configuration is declared, without the needed information about points, this package's corresponding requirements receive UNDETERMINED. The `liegt_auf` closure does not establish completeness at once for all such properties.

These results concern this particular description of the configuration. It does not mean that the mathematical question of whether the needed points exist is unsolvable, or that the axiom itself is refuted. Here again the sixth chapter's distinction is useful: missing ground and a detected violation are different results.

Try it yourself

Take copies of the supplied files, keeping the header and context. In the test and explain commands substitute the copy's path; the path after `--program` stays the same. Change only the named entries, the question, and the state expectation you can justify. Keep the `COMPUTED` and diagnostics expectations for these incidence queries.

1. In a copy of scenario 01, ask whether A lies on bc. Do not change the source information. Predict the state and the ground's kind: a case entry, a rule, or a closure. Then check the `FALSE_ONLY` expectation and read the explanation.
2. In another copy of scenario 01, ask about B and ab. First keep all entries: `TRUE_ONLY` is expected. Then delete only the assertion `hilb-t01-14` reporting that B lies on ab. Predict the new answer, check the old expectation, and only then change it. Compare the grounds' identifiers before and after deletion.
3. In a copy of scenario 05, add a positive `liegt_auf` assertion for D and ab with the new name `book-d-ab`. First keep the `FALSE_ONLY` expectation, then explain the mismatch and record a new one. Why does the negation by this closure not arise on the new run?

Check your predictions

In the first exercise the result will be `FALSE_ONLY` on the ground of `InzidenzAbschluss`. In the second, the source `TRUE_ONLY` rests on `hilb-t01-14`; after deletion, `FALSE_ONLY` will appear on a closure ground. In the third, the answer will become `TRUE_ONLY` and will rest on the case's new entry. The named diagnostic expectations still hold under these changes.

In the first part the absence of an entry left the question open. Now we can find the extra condition that gives absence a meaning: declaring a certain relation complete in a certain domain. The next step is to read rules with exceptions and see how new grounds can change an earlier conclusion.

PART III

Examining the Grounds of the Answer

CHAPTER 08

When Support Is Set Aside

We have already seen how a change of information changes the answer. If a rule lacks a premise, that rule cannot derive its conclusion. If the information is declared complete, the absence of a record can become a ground for denial. Now consider one more case: the rule's premises are established and the rule applies, but an exception sets aside its support for the conclusion.

Until now we have used strict rules, `strict`. Their support cannot be set aside by a defeasible rule or a defeater. That does not mean the answer stays the same under any change of data: a strict rule's conditions must still hold. This chapter introduces rules whose support can be set aside by a separate ground even when their conditions are met.

When a rule applies, and when its support survives

When its conditions are satisfied, a defeasible rule becomes a candidate for deriving a conclusion. Exceptions and priorities determine whether its support survives in the final answer.

The starting presumption

The first example comes from a study package built around a mushroom field guide. Its source is an author's exposition with a limited list of species and identifying features²². We study the conditions of that formalization; the study scenario's result does not establish the edibility of a mushroom found in nature.

The object type here is called `Specimen`. The relation `specimen(s)` reports that the specimen is named in the case. The relation `food_barred(s)` means that eating the specimen is barred under the presumption adopted in this model. By presumption we mean the starting assumption that applies until a specified ground defeats it.

In the package that assumption is recorded by a short rule:

```
@source(MYCO_ART4, MYCO_ART9)
rule PresumeInedible defeasible { for s: Specimen; when specimen(s); then food_barred(s); }
```

²²The author's exposition and its articles; the verdict, presumption, and defeater rules. Scenarios 02 and 03 check the presumption's two sides.

The condition is familiar: the specimen is named. The conclusion is `food_barred(s)`. The new word `defeasible` stands where `strict` stood. It means that the rule can apply while its support is later set aside. The word itself does not specify when this happens. We will see two ways to state it: a defeater that blocks support, and a priority that resolves a conflict between rules.

Start with a scenario containing a single input assertion:

```
test "urn:query:myco-t02-book-presumption-stands" {
  given {
    context { decision_time @2026-08-31T12:00:00Z; knowledge_time @2026-08-31T12:00:00Z; legal_time @2026-08-31; timezone "Europe/Moscow"; }
    assert specimen(entity_ref("urn:case:myco:t02")) { id "myco-t02-1"; origin case_input; }
  }
  evaluate truth(food_barred(entity_ref("urn:case:myco:t02")));
  expect result_kind == PROPOSITION;
  expect truth_status == TRUE_ONLY;
  expect evaluation_status == COMPUTED;
  expect applied(PresumeInedible);
}
```

The rule applies, and there is no ground to lift its support. So `food_barred` receives `TRUE_ONLY`, and the expectation `applied(PresumeInedible)` checks that the named rule was applied in the justification graph.

Run the check from the reader companion directory:

```
law engine test \
  corpus/laws/doctrine/myco/opredelitel/tests/02-book-presumption-stands.lawtest \
  --program corpus/clir/myco-opredelitel.lawir.json --imports \
  contexts/myco-opredelitel.json
```

All expectations pass. A defeasible rule can yield an ordinary positive answer; its difference will show when a ground for an exception appears.

What sets support aside

The second rule links the lifting of the presumption to another established property, `edible_verdict(s)`:

```
@source(MYCO_ART9)
rule FullIdentificationDefeatsBar defeater { for s: Specimen; when edible_verdict(s); defeat food_barred(s); }
```

The word *defeater* denotes a rule that blocks a defeasible conclusion. Instead of then it carries defeat, followed by the conclusion it acts upon. In our example, once *edible_verdict* is established, it lifts the support of *food_barred* that came from the presumption.

defeat food_barred(s) by itself neither concludes *not food_barred(s)* nor creates a permission. That is a different step: what is set aside is a particular ground of the positive conclusion²³. If the assertion has other surviving grounds, they too must be taken into account in reaching the answer.



A muted voice does not become a voice against. Likewise, a defeater by itself does not derive a negation.

Where does *edible_verdict* come from? In this package, strict rules first establish the species from the full set of identifying features. The rule *EdibleOnlyAfterIdentification* requires an established species, its assignment to the class *Edible*, and the negation of *deadly_listed* for that species. The last condition rests on declaring the list complete, using the closure introduced in the last chapter. The mere absence of the species from the list, without the remaining conditions, is not enough for the verdict.

The species' class is stated by the package's own information. Such entries take part in the calculation alongside the entries of a single case. So the verdict's explanation must be traced back to the recorded features and the model's adopted information about the species.

²³The Arxo specification, §§104, 107–108: applicability, defeat, and surviving support. A defeater attacks defeasible support; strict support cannot be lifted that way.

Support is removed, but the question remains

The supplied scenario 03 contains the full set of features specified by one entry in the identification key used in this example. Its values, such as `Present` or `DarkBrown`, are named values from the vocabulary. All starting entries live in the scenario file; below are only the two questions and their expectations:

```
evaluate truth(edible_verdict(entity_ref("urn:case:myco:t03")));
expect truth_status == TRUE_ONLY;
expect applied(EdibleOnlyAfterIdentification);
evaluate truth(food_barred(entity_ref("urn:case:myco:t03")));
expect truth_status == NEITHER;
expect evaluation_status == COMPUTED;
```

The first answer is `TRUE_ONLY`: the verdict rule's conditions hold. The second is `NEITHER`: the presumption's support is lifted, and there is no ground here for an explicit denial of `food_barred`. Compare with the third chapter: `NEITHER` and `FALSE_ONLY` mean different states.

The premise remains; the support is withdrawn



For: none · Against: none

NEITHER

Scenario 03 of the field guide: both premises are established. The defeater does not derive not `food_barred(s)`; there are no other surviving grounds for this question.

Let us check both questions:

```
law engine test \
  corpus/laws/doctrine/myco/opredelitel/tests/03-book-verdict-lifts-bar.lawtest \
  --program corpus/clir/myco-opredelitel.lawir.json --imports \
  contexts/myco-opredelitel.json
```

Now examine the reason for the second answer. Copy the scenario 03 file used in the command above to the reader companion directory under the name `bar.lawtest`. Keep the header and the whole given block. Delete the first `evaluate`, the one about

edible_verdict, and the two expect lines after it. One question about food_barred will remain, which can be handed to the explanation command:

```
chapter8_why_not=$(mktemp -d)
python3 reader_tools/prepare_question.py bar.lawtest \
  --program corpus/clir/myco-opredelitel.lawir.json --imports \
  contexts/myco-opredelitel.json --out "$chapter8_why_not"
law engine why-not "$chapter8_why_not"
```

Below is the result with the rule-application identifier shortened; the closing section on kinds of diagnostics is omitted:

```
статус результата: NEITHER

блокираторы (1):
  [defeated rule] urn:proof:apply:PresumeInedible:58c20198...
    побеждено по причине defeater; заключение употребление экземпляра запрещено
    презумпцией несъедобности(t03)
```

Author's paraphrase of the engine output above (the engine's text is in Russian and is quoted verbatim; what follows is the author's English retelling): the result status is NEITHER, with one blocker — the application of PresumeInedible stands defeated by defeater: the presumption-based bar on eating the specimen (t03) contributes no support to the final answer.

The line `defeated rule` points at the set-aside application of `PresumeInedible`, and the reason `defeater` explains why its support takes no part in the final answer. The premise `specimen(s)` is still there. The rule applies, but a specified exception defeats its support for the conclusion.

The question about `food_barred` remains well-defined: we asked it and received `NEITHER`. That answer does not mean the absence of the bar is established. It reports that, once the exception is taken into account, there is no surviving ground either for or against the assertion.

When there is a ground for denial

Consider a second example: one rule supplies negative support, and a priority sets aside the opposing positive support. In the package on the Scrum Guide there is a type `Team` and three one-argument relations: `scrum_team` — the team is declared a Scrum team, `work_visible` — the work is visible to those performing and receiving it, `work_hidden` — the work is hidden from them.

Here it matters most to separate the text from the formalization. The Guide requires the work to be visible. That requirement by itself does not prove the work is actually

visible. The presumption of visibility and the way it meets concealment are recorded in the model as further assumptions. We will examine how these assumptions work. Whether they are well-grounded calls for a separate comparison against the source.

«The emerging process and work must be visible to those who perform the work and those who receive it.»

Ken Schwaber, Jeff Sutherland · Scrum Guide · 2020, “Transparency”

The edition's English rendering; consult the source for its wording. © 2020 Ken Schwaber and Jeff Sutherland. Adapted under CC BY-SA 4.0; changes: English rendering.

The source states a requirement. The visibility presumption and the priority under concealment are added in the study model and call for independent justification.

The first rule gives the starting support for visibility:

```
@source(SG_S3_1)
rule WorkPresumedVisible defeasible {
  label en official "Transparency: the work of a Scrum Team is presumed visible until the contrary is established";
  for t: Team;
  when scrum_team(t);
  then work_visible(t);
}
```

The second yields the opposite conclusion once concealment is established:

```
@source(SG_S3_1)
rule HiddenWorkBreaksTransparency defeasible {
  label en official "Transparency: work hidden from those performing or receiving it is not visible";
  for t: Team;
  when scrum_team(t) and work_hidden(t);
  then not work_visible(t);
}
```

Both rules are defeasible. The second carries then not, not defeat: it creates support for the denial of `work_visible`. If a case establishes both `scrum_team` and `work_hidden`, both rules' conditions hold. It remains to determine how to resolve their conclusions.

A priority between rules

For this pair the package holds a separate declaration:

```
@source(SG_S3_1)
priority HiddenWorkBeatsPresumptionOfVisibility {
  label en official "an established concealment prevails over the presumption
of visibility";
  prefer HiddenWorkBreaksTransparency over WorkPresumedVisible;
  reason explicit_exception;
}
```

The word `priority` introduces a named declaration for resolving a conflict between rules. The line `prefer ... over ...` gives the concealment rule precedence over the presumption of visibility. `reason explicit_exception` names the reason stated in the model: an explicit exception. That is a justification mark, not an automatic proof that the source states exactly this priority²⁴.

In scenario 07 the team is named, and nothing is reported about concealment. The presumption yields `TRUE_ONLY`. In scenario 08 `work_hidden` is added, and the visibility question receives `FALSE_ONLY`. The presumption's positive support is set aside by `priority`, while the second rule's negative support survives.

The second case can be checked with this command:

```
law engine test \
  corpus/laws/org/scrum/guide/tests/08-book-hidden-work-prevails-by-priority.law
test \
  --program corpus/clir/scrum-guide.lawir.json --imports contexts/scrum-guide.js
on
```

The reason the presumption was set aside will be shown by:

```
chapter8_why_not=$(mktemp -d)
python3 reader_tools/prepare_question.py corpus/laws/org/scrum/guide/tests/08-bo
ok-hidden-work-prevails-by-priority.lawtest \
  --program corpus/clir/scrum-guide.lawir.json --imports contexts/scrum-guide.js
on \
  --out "$chapter8_why_not"
law engine why-not "$chapter8_why_not"
```

Unlike the field guide, the final status here is `FALSE_ONLY`, and the set-aside rule's line carries the reason `priority`. These are two differences to be read together: what remains in the answer, and why the other ground was set aside.

²⁴Fragment `SG_S3_1` records the visibility requirement; `02-theory.law` states the formal rules and the priority. Their action is checked by scenarios 07 and 08.

What happens without the priority

Let us check the third declaration's role with a separate trial. As in the sixth chapter, make a copy of the package. The commands run from the reader companion directory in one terminal window:

```
chapter8_work=$(mktemp -d)
cp -R corpus/laws/org/scrum/guide "$chapter8_work/guide"
```

In the file `02-theory.law` inside the copy, delete only the declaration `HiddenWorkBeatsPresumptionOfVisibility` together with its `@source(SG_S3_1)` line. Keep both rules. Then prepare the changed model and repeat the starting scenario 08:

```
law engine lower "$chapter8_work/guide" > "$chapter8_work/without-priority.lawir.json"
law engine test \
  corpus/laws/org/scrum/guide/tests/08-book-hidden-work-prevails-by-priority.law
test \
  --program "$chapter8_work/without-priority.lawir.json" --imports \
  contexts/scrum-guide.json
```

The `FALSE_ONLY` expectation will fail: the result is `BOTH`. In this trial both opposing supports survive, since the preference between them is removed.

The order of the rules in the file does not replace a priority.

If you copy the scenario and change the expected status to `BOTH`, the check will pass. It will confirm the changed model's behaviour, not establish that the model faithfully reads the Guide. Restoring the priority and running `lower` again returns the starting scenario's result.

Compare the cases considered:

Three outcomes after comparing the grounds

Case	Surviving grounds	Answer
The verdict lifts the presumption in the field guide	none for or against <code>food_barred</code> defeater: positive support lifted; no negative support arose.	NEITHER
Concealment has priority in the Scrum model	only against <code>work_visible</code> priority: the presumption's support set aside; the negative support survives.	FALSE_ONLY
The same pair of rules without priority	for and against <code>work_visible</code> Both opposing applications survive.	BOTH

The first case asks about `food_barred`. The second and third ask one question about `work_visible` on the same information; between them only the priority declaration is changed.

These are the results of particular sets of rules and information. The mere presence of a defeater never promises NEITHER, nor does a priority always promise FALSE_ONLY: the outcome depends on all surviving grounds for the stated assertion.

Check every surviving ground

A defeated rule application is only part of the explanation. Other grounds for the same assertion may survive. Support for its negation must also be checked separately.

Try it yourself

1. In a copy of the field guide's scenario 03, delete only the `myco-t03-6` entry about the spore print. Predict both answers, then run the check with the old expectations. After accounting for the mismatches, correct the expectations. To examine the answer about `food_barred`, keep only the second question in a separate copy, as done above.
2. In a copy of the Scrum scenario 08, delete only the `sg-t8-b` entry about the work's concealment. Predict the state of `work_visible` and the applied rule. First check the old expectations, then change them for the revised case.
3. After the trial without priority, restore the deleted declaration in the package copy, run `lower` again, and start the untouched scenario 08. Explain which support is now set aside and which survives.

Check your predictions

In the first exercise the full set of identifying features is no longer established, so the species is not identified. `edible_verdict` will receive NEITHER, and the expectation of applying `EdibleOnlyAfterIdentification` must go. `food_barred` will be TRUE_ONLY: the defeater's condition fails, and the presumption's support survives.

In the second exercise visibility will receive `TRUE_ONLY`; replace the expectation of applying the concealment rule with `applied(WorkPresumedVisible)`. In the third, the restored priority leaves the negative support standing again, and the starting check with `FALSE_ONLY` passes.

We have learned to distinguish a rule that did not apply, a support that was set aside, and support for a denial. In all three cases the explanation leads to different grounds, even when the answer states sometimes coincide. The next chapter adds one more source of differences: a decision that must come not from the model but from a person named in it.

The English wording of the Scrum rules and priorities in this chapter is quoted verbatim from the package, which follows the English original of the Guide; the surrounding prose retelling is the translator's own.

CHAPTER 09

The Answer That Needs a Human

In the last chapter we changed the rules and watched which grounds kept their force. Now leave the rules in place. Imagine that everything needed to consider a request is known, but the decision itself must still come from a human.

No further
calculation can replace that decision.

Arxo has a special declaration for this situation. It lets us name the question that calls for an outside decision and say who must make it. Before reading that entry, let us separate two questions that blend easily into one in ordinary speech: whether the way is open, and whether entry is permitted.

The gates are open

In Franz Kafka's parable "Before the Law" a man comes to the doorkeeper and asks to be let in. The gates stand open. The doorkeeper says he cannot let him in now, but allows for the possibility later. The man stays to wait.

Take the model of this scene from the corpus. One distinction interests us: the state of the gates is known from the text, while permission to enter in the chosen formalization depends on a separate decision of the doorkeeper. That is one way of questioning the parable; it does not exhaust its meaning and invites discussion²⁵.

The package's relation names are German. `weg_frei` means that the way through the gates is free, and `einlass_gestattet` that entry is permitted. In both cases the argument is the man. A constant `Mann` is already declared for the petitioner:

```
const Mann: Person = entity_ref("urn:de:kafka:clir:vor-dem-gesetz#Mann") {
  label de official "der Mann vom Lande";
  label ru official "проситель из деревни";
}
```

²⁵Franz Kafka, "Before the Law" ("Vor dem Gesetz"), first printed in *Selbstwehr*, 1915. The pinned German text and the fragment registry. The open gates and the free way are described in `01-tor.law`.

After `const` come the name and the type, after the equals sign the reference to the particular object. So in the examples we can write `Mann` instead of the long `entity_ref(...)`: the same petitioner is meant. `DerTuerhueter` likewise denotes the doorkeeper, and `Tor` the gates.



The way is open. Permission to enter is a separate question: in this model it needs other grounds.

Some starting information already lives in the package. In particular, the request for entry is recorded thus:

```
@source(VDG_ANKUNFT)
assert eintritt_erbeten(Mann) { evidence VDG_1915_TEXT; }
```

The familiar `assert` states a fact. The `@source` mark binds it to a source fragment, and `evidence VDG_1915_TEXT` points at the evidence the entry carries. Here that is the parable's text. We will examine the use of evidence in detail in the eleventh chapter. For now note that these entries are already available when the scenario runs: the request need not be repeated in every given block.

The premise a human must report

The assertion "the doorkeeper has permitted this petitioner to enter" is recorded by the relation `tuerhueter_gewaehrt_einlass`. Let us read its declaration:

```
@source(TO_JETZT_NICHT, TO_SPAETER_MOEGLICH)
external judgment relation tuerhueter_gewaehrt_einlass(m: Person) {
  authority Tuerhueter;
  label de official "der Türhüter gewährt diesem Mann den Einlaß";
  label ru official "привратник даёт этому просителю впуск";
}
```

The relation form is familiar to us: the name, the argument *m* of type *Person*, then the declaration body. The new words stand before the declaration: *external judgment*. They mark out the relation set aside for an outside judgment. Here that is the entry decision the model must receive from outside.

The line *authority Tuerhueter* names the one to whom the question is addressed: the doorkeeper²⁶. Here *Tuerhueter* is the name of the authorized person in the judgment's declaration. The constant *DerTuerhueter* we met above denotes the character in the parable's facts. These names play different roles in the model: *Tuerhueter* names the authority in the judgment declaration; *DerTuerhueter* can be an argument in an assertion about the character.

The declaration by itself does not yet establish permission. For that the package holds a rule:

```
@source(TO_JETZT_NICHT)
rule EinlassGestattet strict {
  label de official "Ob der Mann eintreten darf, entscheidet der Türhüter: er allein kann den Eintritt gewähren oder versagen";
  label ru official "Войти ли просителю – решает привратник: только он даёт вход и только он его не даёт";
  for m: Person;
  when eintritt_erbeten(m) and tuerhueter_gewaehrt_einlass(m);
  then einlass_gestattet(m);
}
```

The long labels are left as recorded in the package. The conclusion's meaning is set by the *when* and *then* lines: if the man asked for entry and the doorkeeper permitted him to enter, then entry is permitted to him. The labels explain the rule; they add no conditions or conclusions of their own.

The first premise is already supported by the fact of the request. The second calls for a decision. The rule stays *strict*, *strict*: new here is the origin of one of its premises.

A strict rule, an outside premise

strict describes the mode of inference. *external judgment* describes the origin of one of the premises. These characteristics belong to different parts of the model.

²⁶The Arxo specification, §47.3: *external judgment*; the entry declaration and rule. The occasions without a decision, with consent, and with refusal are VG-01–VG-03.

Before the decision is supplied

Start with the scenario VG-01. It states a context but no further assertions. Leave the date values as in the starting example: for now we change only the information about the decision.

```
test "VG-01 выпуск без ответа привратника: REQUIRES_JUDGMENT, орган Tuerhueter"
{
    given {
        context {
            decision_time @1000-01-05T12:00:00Z;
            knowledge_time @1000-01-05T12:00:00Z;
            legal_time @1000-01-05;
            timezone "UTC";
        }
    }

    evaluate truth(einlass_gestattet(Mann));
    expect evaluation_status == REQUIRES_JUDGMENT;
    expect truth_status == NEITHER;
    expect judgment_requests(1);
    expect judgment_request(tuerhueter_gewaehrt_einlass);
}
```

The first two expectations check different sides of the answer. `truth_status == NEITHER` means there is no support for the assertion of permitted entry, either for or against. `evaluation_status == REQUIRES_JUDGMENT` refines the reason: the derivation depends on a missing premise that calls for an outside decision.

The last two lines check the request itself. `judgment_requests(1)` expects one judgment request; `judgment_request(...)` checks which relation it concerns. So the test demands naming the missing decision, not only returning `NEITHER`.

All scenarios of this file can be run from the reader companion directory:

```
law engine test corpus/laws/fiction/kafka/vor-dem-gesetz/tests/vor-dem-gesetz.la
wtest \
    --program corpus/clir/de-vor-dem-gesetz.lawir.json --imports \
    contexts/de-vor-dem-gesetz.json
```

To see the explanation of a single question, use a separate file containing the same case:

```
law engine explain \
    corpus/laws/fiction/kafka/vor-dem-gesetz/tests/01-entry-requires-the-doorkeepe
rs-judgment.lawtest \
    \
    --program corpus/clir/de-vor-dem-gesetz.lawir.json
```

The start of the explanation looks like this; the node's long address is omitted here:

```
der Einlaß ist ihm gestattet(Mann): NEITHER
judgment der Türhüter gewährt diesem Mann den Einlaß(Mann)
```

Author's paraphrase of the engine output above (the engine's text is in German and is quoted verbatim; what follows is the author's English retelling): "entry is permitted to him" (Mann) — NEITHER; the missing piece is a judgment — "the doorkeeper grants this man entry" (Mann).

The judgment node shows which decision is missing. That is a description of the request. Running the command `neither` contacts the doorkeeper nor obtains a decision. Asking the human and recording the decision still lie ahead.

The request still needs to reach a person

A `JudgmentRequest` identifies the question and its addressee. Someone must still deliver the question and record the answer.

Supplying consent

Now consider a hypothetical version of the scene: the doorkeeper has agreed to let the petitioner in. We change the example's input information; we ascribe no such continuation to the parable. In the scenario `VG-02` an entry appears inside given:

```
assert tuerhueter_gewaehrt_einlass(Mann) {
  id "assert-adj0";
  origin adjudicated;
}
```

`origin adjudicated` marks the assertion as a supplied decision. It concerns the same man the question is asked about. Now both premises of the `EinlassGestattet` rule are supported, and the expectations change:

```
evaluate truth(einlass_gestattet(Mann));
expect evaluation_status == COMPUTED;
expect truth_status == TRUE_ONLY;
expect judgment_requests(0);
```

The conclusion receives `TRUE_ONLY`, the calculation `COMPUTED`, and there are no more judgment requests. The explanation shows the rule applied on two grounds: the request from the package and the decision from given.

The adjudicated mark reports how the entry is presented. By itself it does not certify who entered it, nor that this person is truly entitled to decide the question. In the shown example we supply a decision; checking its provenance remains a separate

task. The authority declaration lets us name the required participant, but it does not replace such a check²⁷.

Supplying a refusal

In the scenario VG-03 one word changes:

```
assert not tuerhueter_gewaehrt_einlass(Mann) {
  id "assert-adj0";
  origin adjudicated;
}
```

not supplies an explicit negative answer: the doorkeeper does not grant entry. This is a refusal, not an absence of a decision. Before reading the expectations, ask yourself: what has become known about the doorkeeper's decision, and what now follows about permitted entry?

The judgment itself has received negative support. But the question still concerns a different relation, `einlass_gestattet`:

```
evaluate truth(einlass_gestattet(Mann));
expect evaluation_status == COMPUTED;
expect truth_status == NEITHER;
expect judgment_requests(0);
```

The result is `NEITHER` again. The rule concludes permission from consent; under refusal its positive premise fails. The package holds no rule that would conclude `not einlass_gestattet(m)` from that refusal.

Denying the premise does not automatically turn into denying the conclusion.

The calculation still has the status `COMPUTED` and no longer calls for the doorkeeper's decision: the decision is already supplied. If you run `why-not` for separate scenario 03, the explanation will point at `EinlassGestattet` and the premise's `FALSE_ONLY` state. Here it is known why the rule did not fire.

Compare the three cases for the same permission question:

²⁷§47.3.1 tells apart an answer with a judgment object from the compatible form without one. In the first, authority and payload form are checked; the shown entry uses the second. On the provenance of assertions, see §73.

Same support, different reasons

Supplied decision	Support of the answer	Calculation status	Decision requests
No decision Judgment: NEITHER	NEITHER	REQUIRES_JUDGMENT	1
Consent Judgment: TRUE_ONLY	TRUE_ONLY	COMPUTED	0
Refusal Judgment: FALSE_ONLY	NEITHER	COMPUTED	0

The judgment is `tuerhueter_gewaehrt_einlass(Mann)`; the question is `einlass_gestattet(Mann)`. Under refusal, the denial is established for the premise, but there is no rule carrying that denial over to the conclusion.

Two identical NEITHERs hide different situations. In the first case a decision is still called for; in the third it is known but gives support to neither answer to our question. The value COMPUTED also promises neither "yes" nor "no": a calculation can finish with no support on either side of the conclusion being queried.

What exactly the text says

A natural doubt remains: the doorkeeper did say "not now". Why does the first scenario wait for a decision at all?

«"Maybe," says the doorkeeper, "but not now."»

Franz Kafka · "Before the Law" · Selbstwehr, 1915, p. 2

The edition's own English translation of the German first printing.

The character's words are kept in the package as separate facts. Linking them to an outside judgment is possible only in an explicitly stated way; no such link is recorded here.

In the model his words are kept as a separate fact:

```
@source(TO_JETZT_NICHT, TO_NOCH_NICHT)
assert einlass_jetzt_versagt(DerTuerhueter, Mann) { evidence VDG_1915_TEXT; }
```

The relation `einlass_jetzt_versagt` records the refusal of entry for now. The words about a later possible entry are recorded by one more relation, `einlass_spaeter_moeglich`. No rule links these facts to the outside judgment `tuerhueter_gewaehrt_einlass`.

So the first scenario means exactly the absence of a supplied judgment in this formalization. From it one cannot conclude that the character said nothing or made no decisions at all. Another formalization can be proposed that reads his words as

a negative judgment. Then the links between the facts and the answer will change; those links are what should be discussed when comparing readings.

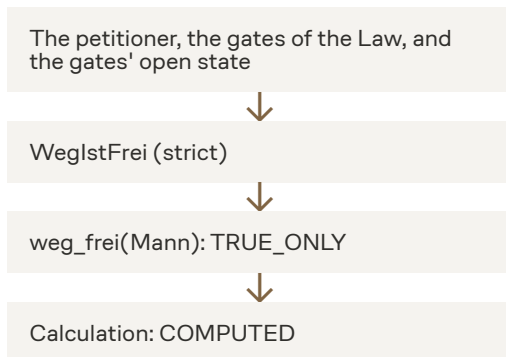
And the open gates stay open meanwhile. A separate scenario 04 asks about the free way:

```
evaluate truth(weg_frei(entity_ref("urn:de:kafka:clir:vor-dem-gesetz#Mann")));
expect result_kind == PROPOSITION;
expect truth_status == TRUE_ONLY;
expect evaluation_status == COMPUTED;
```

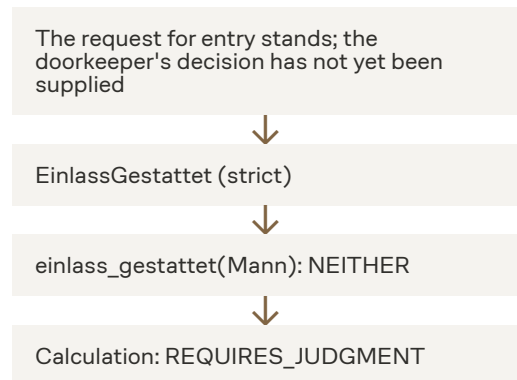
The rule `WegIstFrei` rests on the information about the petitioner, the gates of the Law, and their open state. The doorkeeper's decision is not among its conditions. So this question receives `TRUE_ONLY` and `COMPUTED` without a supplied judgment. Waiting for a decision belongs to one particular question; it does not block every answer of the model.

Two questions about one petitioner

Is the way free?



Is entry permitted?



This is the initial case, before a judgment has been supplied. The request concerns permission to enter; whether the way is open is computed independently.

Try changing the decision

Work with a copy of the file `vor-dem-gesetz.lawtest`, keeping the file header and one test `VG-02` in it. For each trial start from that copy, so changes do not accumulate.

1. Delete from given the assertion with `origin adjudicated`. Keep the expectations at first and run the test. The check must point out a mismatch in the answer state, the calculation status, and the number of requests. Account for each result, then record the expectations for the case without a supplied decision.
2. Restore the assertion and add `not` before the relation. Predict the permission question's result. Then replace the question itself with

`truth(tuerhueter_gewaehrt_einlass(Mann))`. Why is `FALSE_ONLY` now expected, although the permission question gave `NEITHER`?

3. Compare the explanation of the open way from scenario 04 with the decision-free permission's explanation from scenario 01. Find the first conclusion's grounds and the second's missing premise. Which information does each of these questions need?

We have learned to reserve a specific question for a human decision and to continue the derivation once that decision is supplied. The next step is to distinguish an act from its normative position: whether it is permitted, required, or prohibited. Open gates, permission to enter, and actually entering already give us three different subjects of conversation.

No single published English translation of Kafka's "Before the Law" was adopted for this chapter: the German quotations above stay in German, each with a brief English gloss beside it, and all English renderings of Kafka's text in this chapter are the translator's own.

CHAPTER 10

What Is Permitted, What Is Required, What Happened

In the last chapter we separated an open way from permission to enter. If the petitioner then crosses the threshold, another record may appear: that the entry took place. By itself that record says nothing about whether the entry was permitted.

Such distinctions reach far beyond law. A pronunciation rule says how a word should be read. A communications standard imposes requirements on a program's behaviour. In both cases we must ask separately what is prescribed and what happened. We begin with a relation whose meaning includes a prohibition. We then add a way to say who bears a duty, what state it requires, and how reported behaviour bears on its fulfilment. The new construction here is a duty; permission remains a separate question, as in the preceding chapter.

What "may not" means in a relation's name

In §33 of the "Söz sazy" (The Melody of Speech) section of Rabigha Syzdyq's *Қазақ тілінің анықтағышы* (A Handbook of Kazakh, Elorda, 2000), full and shortened verb forms are considered. In reading, the form chosen in the text must be kept. For example, a written "кеп" ("kep", shortened) may not be replaced in pronunciation with the full "келіп" ("kelip"). Where the full form is given, shortening it is likewise not allowed²⁸.

Consider the first half of this rule in the corpus package:

```
@source(YQSH_F_S0Z_33)
rule YqshamdyTolyqAituBolmaidy strict {
  label kk-KZ official "$ 33: ықшам жазылғанды толық етіп айтуға болмайды";
  label ru-KZ translation "$ 33: написанное сокращённо нельзя произносить полн
ым";
  for w: EtistikTulgasy;
  when yqsham_turinde_berilgen(w);
  then yqshamdy_tolyq_etip_aituga_bolmaidy(w);
}
```

The variable *w* denotes a verb form of the type *EtistikTulgasy*. The premise *yqsham_turinde_berilgen(w)* reports that the form is given shortened. The conclu-

²⁸Rabigha Syzdyq, *Қазақ тілінің анықтағышы* (Elorda, Astana, 2000), "Söz sazy" section, §33: the source handbook, the verb-form rules and the scenarios.

sion `yqshamdy_tolyq_etip_aituga_bolmaidy(w)` means that it may not be pronounced in full ("bolmaidy" — "may not, is not allowed"). The translation mark on the Russian label denotes a translation; the mark itself does not certify its accuracy.

The first scenario states the form "кеп" ("kep"):

```
assert yqsham_turinde_berilgen(entity_ref("urn:kz:grammar:clir:yqsham-tulgalar#Kep")) {
  id "assert-a0";
  origin case_input;
}
```

The question asks whether the prohibition has support:

```
evaluate truth(yqshamdy_tolyq_etip_aituga_bolmaidy(entity_ref("urn:kz:grammar:clir:yqsham-tulgalar#Kep")));
expect evaluation_status == COMPUTED;
expect truth_status == TRUE_ONLY;
```

The words "may not be pronounced" are part of the relation's meaning. This is still an ordinary one-argument assertion: the rule derives it, and `truth` checks its support. `TRUE_ONLY` means that the prohibition is supported for this form. It says nothing about how anyone actually pronounced the word.

A prohibition can have positive support

`TRUE_ONLY` is not the same as permission. Here it supports an assertion that prohibits a particular pronunciation.

The model identifies neither a particular reader nor an act of reading aloud. To establish a violation, we would need to name the person bound by the prohibition, record how that person pronounced the word, and state rules linking the two. That is this vocabulary's boundary: it lets us reason about word forms without describing individual acts of reading.

The Kazakh material in this section stays in Kazakh, with a Latin-script transliteration beside each form; all English renderings of Syzdyq's text are the translator's own.

Whom the requirement concerns

Now take BCP 14 — the documents RFC 2119 and RFC 8174 on the meaning of key words in standards. To begin we need one word: **MUST**. In section 1 of RFC 2119

²⁹RFC 2119, §§1 and 3 defines **MUST** and **SHOULD**; RFC 8174, §2 refines the meaning of writing in capitals. The requirement's link to the implementation's conformance claim is set by `package.law`.

it marks an unconditional requirement of the specification²⁹. The wording below follows the English original of RFC 2119.

«This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.»

Scott Bradner · RFC 2119 · §1, MUST · March 1997

The edition's English rendering; consult the source for its wording.

The definition sets the requirement's force. Its link to a particular implementation and the information about fulfilment are represented in the model by separate relations.

The package binds such a requirement to a program that claims conformance to the document. In the vocabulary it is called an implementation, `Implementation`; the requirement has the type `Requirement`. In the relation `subject_to(i, r)` the first argument is the implementation, the second the requirement that applies to it. `absolute_requirement(r)` reports that the requirement is ranked at the MUST level or that of an equivalent word.

Both conditions meet in the rule's familiar opening:

```
@source(RFC2119_§1)
rule AbsoluteRequirementBindsTheImplementation strict {
  label en official "an absolute requirement of the specification binds every
  implementation which claims conformance";
  for i: Implementation; for r: Requirement;
  when subject_to(i, r) and absolute_requirement(r);
  then duty SatisfyAbsoluteRequirement {
    bearer i;
    goal maintenance {
      condition requirement_satisfied(i, r);
      window [@1997-03-01, infinity);
    }
  };
}
```

The new construction begins after `then`. In place of an assertion stands duty `SatisfyAbsoluteRequirement`: a named duty with a body of its own. When the conditions hold for particular values of the variables, a duty arises for that implementation with respect to that requirement. This duty is a normative position: it has a bearer and a state of its own, distinct from the rule that creates it and the assertion it requires to hold.

Let us read the body in order:

Entry	What it states
bearer <i>i</i>	The implementation that bears the duty
goal maintenance	The goal: continued maintenance of a state
condition <code>requirement_satisfied(i, r)</code>	The state to be maintained: this implementation satisfies this requirement
window <code>[@1997-03-01, infinity)</code>	The interval during which that state is to be maintained

Unlike a one-time achievement, maintenance requires keeping the condition.

One successful check does not yet prove it was kept before and will always be kept.

The interval's start is included; no end bound is stated. We take the date from the model; we will examine ways of recording and checking time in the thirteenth chapter³⁰.

Creating the duty does not establish the truth of `requirement_satisfied`. That condition is still to be checked against information about the implementation's behaviour.

From recorded behaviour to a violation

In the study scenario there is a server and a requirement denoted by the address `allow`. The given records in order: the document uses BCP 14; the requirement lives in that document; the server claims conformance to it; the requirement is marked with the word **MUST** written in capital letters; the server's behaviour departs from the requirement.

`Must` and `AllCapitals` are values in the vocabulary for the word and its capitalization. The remaining participants are named by `entity_ref` links. Let us read the whole scenario:

³⁰The Arxo specification: §123 — kinds of goals, §137 — keeping a state. The BCP 14 rules are in `O2-requirement-levels.law`; the examples in scenarios O1: **MUST** violation and O4: departure justification.

```

test "urn:query:bcp14-test-must-breached" {
  given {
    context { decision_time @2026-08-30T12:00:00Z; knowledge_time @2026-08-30T12:00:00Z; legal_time @2026-08-30; timezone "UTC"; }
    assert adopted_bcp14(entity_ref("urn:ietf:rfc:9110")) { id "bcp14-t1-a"; origin case_input; }
    assert stated_in(entity_ref("urn:ietf:rfc:9110#allow"), entity_ref("urn:ietf:rfc:9110")) { id "bcp14-t1-b"; origin case_input; }
    assert claims_conformance(entity_ref("urn:case:impl:server"), entity_ref("urn:ietf:rfc:9110")) { id "bcp14-t1-c"; origin case_input; }
    assert keyword_written(entity_ref("urn:ietf:rfc:9110#allow"), Must, AllCapitals) { id "bcp14-t1-d"; origin case_input; }
    assert behaviour_deviates(entity_ref("urn:case:impl:server"), entity_ref("urn:ietf:rfc:9110#allow")) { id "bcp14-t1-e"; origin case_input; }
  }
  evaluate positions();
  expect result_kind == NORM_POSITION;
  expect evaluation_status == COMPUTED;
  expect normative_status == VIOLATED;
}

```

Here is a new question: `positions()`. It asks for the normative positions that have arisen. So the answer's kind is `NORM_POSITION`, and the field checked is `normative_status`. In this scenario one duty arises, and its status is expected to equal `VIOLATED`, "violated".

How is that result reached? Other rules in the package, not printed here, use the first three entries to derive `subject_to` and the fourth to establish the requirement's level. Those conclusions suffice for the duty to arise. The last entry takes part in a separate rule:

```

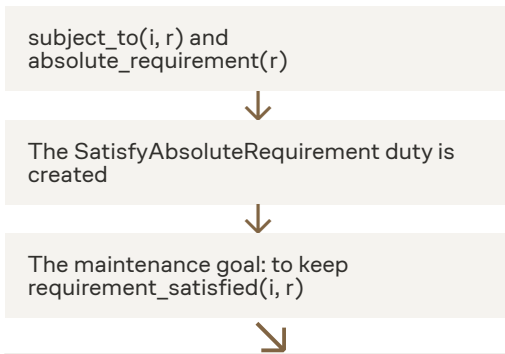
@source(RFC2119_S1)
rule DeviatingBehaviourFailsTheRequirement defeasible {
  label en official "observed deviating behaviour fails the requirement";
  for i: Implementation; for r: Requirement;
  when subject_to(i, r) and behaviour_deviates(i, r);
  then not requirement_satisfied(i, r);
}

```

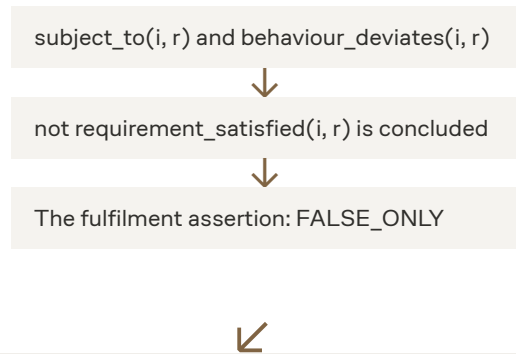
The conclusion carries an explicit denial: `not requirement_satisfied`. So in this case the duty's condition stands refuted. That is a ground for establishing a violation of the standing duty. The chain holds two distinguishable steps: first the duty appears, then the behaviour information lets us assess its state.

A duty and the ground for its violation

How the duty arises



The behaviour information



The duty is created; its condition stands refuted in the checked window.

VIOLATED

Scenario 01 of BCP 14. `norm_creation` explains how the duty arose. The violation's ground lies in another chain — at the denial of `requirement_satisfied`.

In a copy of the scenario, replace the question and its expectations. Now ask the ordinary truth question, whether the requirement is satisfied:

```

evaluate truth(requirement_satisfied(entity_ref("urn:case:impl:server"), entity_ref("urn:ietf:rfa:9110#allow")));
expect result_kind == PROPOSITION;
expect evaluation_status == COMPUTED;
expect truth_status == FALSE_ONLY;
  
```

`FALSE_ONLY` concerns support for the assertion of fulfilment; `VIOLATED` concerns the duty's state. These answers fit together here, but they do not mean the same thing. Neither status can stand in for the other.

Three fields, three different questions

`truth_status` describes support for the assertion; `normative_status` describes the duty's state; `evaluation_status` describes the calculation's status.

Run the scenario about the duty with this command from the reader companion directory:

```

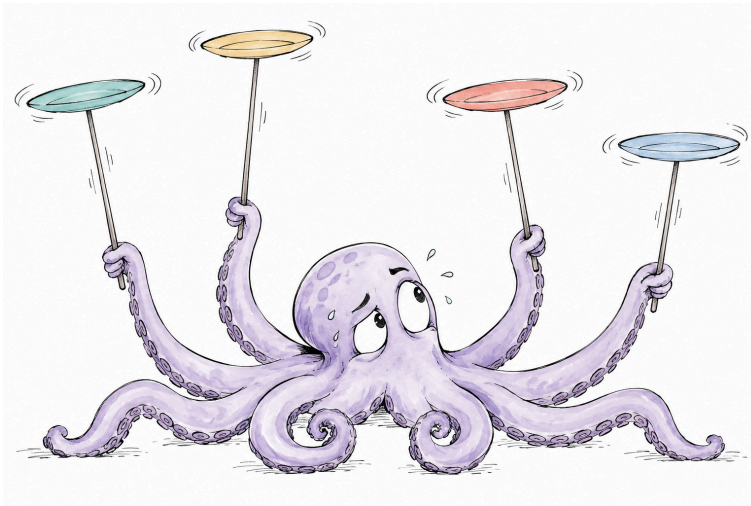
law engine test corpus/laws/org/ietf/bcp14/tests/01-must-breached-is-violated.la
wtest \
  --program corpus/clir/ietf-bcp14.lawir.json --imports contexts/ietf-bcp14.jso
n
  
```

If `test` is replaced with `explain`, the explanation begins with the line `позиций: 1` ("positions: 1") and the node `norm_creation`. This tree shows why the duty arose: why the server is bound by the requirement and why the requirement counts

as unconditional. It does not by itself show why the maintenance condition was refuted. To inspect that second chain, run explain on the copy containing the truth question; the path leads to the denial of `requirement_satisfied`.

The duty stands, but there is no information about fulfilment

Delete only the `behaviour_deviates` entry. The old `VIOLATED` expectation no longer passes: the position receives `ACTIVE`. The duty's conditions for arising survive, but this case no longer contains information about its violation.



The plates are still spinning. Keeping them that way takes continuing effort, just as a maintenance duty continues to apply.

The question about `requirement_satisfied` then receives `NEITHER`. The duty stands, while the requirement's fulfilment is neither established nor refuted. `ACTIVE` does not mean the behaviour was checked and judged conforming.

Now report `behaviour_conforms` instead of a departure, with the same two arguments: the observed behaviour conforms to the requirement. `TRUE_ONLY` appears for `requirement_satisfied`, but the duty stays `ACTIVE`. The requirement to maintain the state remains in force; a positive observation does not complete it.

Compare the three versions of the same case:

A continuing duty and evidence of compliance

Behaviour information	Is the requirement fulfilled: truth_status	Duty: normative_status
Departure reported	FALSE_ONLY	VIOLATED There is a violation ground.
No information	NEITHER	ACTIVE Fulfilment is unknown.
Conformance reported	TRUE_ONLY	ACTIVE The duty to maintain the condition continues.

The conditions that create the duty and the context stay the same in all three cases. ACTIVE is compatible with both missing information and a positive observation.

In all three variants the calculation has the status `COMPUTED`. What differs is the assertion's support and the duty's state.

Now change the case at an earlier point. Restore the report of deviating behaviour, but delete `claims_conformance`, the server's claim of conformance to the document. This case no longer yields `subject_to`, so no duty is created. `explain` shows позиций: `0` ("positions: 0"). When the old `VIOLATED` expectation is checked, the overall answer will be `UNDETERMINED`; the position list shows this is about a duty's absence, not a new state of the old one.

That is the chosen model's result. By itself the absence of a conformance claim does not prove the server has no other duties in a real situation.

When a departure calls for justification

The word `SHOULD` has another sense. Section 3 of RFC 2119 admits departing from a recommendation for weighty reasons in particular circumstances; the implications must be understood and carefully weighed before choosing another path.

The bare fact of a departure does not create those grounds.

Here the last chapter comes in handy. The relation `valid_reason_to_ignore(i, r)` is declared an outside judgment. In the model the question is addressed to the Implementor, the implementation's developer. That is the model's chosen assessor; the RFC itself institutes no body of that name.

A separate relation `implications_understood_and_weighed` records that the implications were considered. Together these premises serve the rule:

```
@source(RFC2119_S3)
rule DeviationFromRecommendationIsJustified strict {
  label en official "ignoring a recommended item is justified when valid reasons exist and the full implications have been weighed";
  for i: Implementation; for r: Requirement;
  when subject_to(i, r)
    and implications_understood_and_weighed(i, r)
    and valid_reason_to_ignore(i, r);
  then deviation_from_recommendation_justified(i, r);
}
```

In scenario 04 it is already reported that the implications were weighed, but the decision about the reasons' weight is not supplied. The question about the departure's justification receives the familiar pair:

```
evaluate truth(deviation_from_recommendation_justified(entity_ref("urn:case:impl:server"), entity_ref("urn:ietf:rfc:9110#allow")));
expect result_kind == PROPOSITION;
expect truth_status == NEITHER;
expect evaluation_status == REQUIRES_JUDGMENT;
```

This is a question about the grounds of a possible departure. It does not report that the server already departed: this scenario holds no such fact. `REQUIRES_JUDGMENT` concerns exactly the requested justification.

Other scenarios supply the missing positive judgment. With `subject_to` established, that judgment and the record that the implications were understood and weighed let the rule derive that a departure is justified. Rules and priorities not printed here then set aside the ground that makes the recommendation binding, so the matching duty is not created. The judgment alone is not enough: the implications must also have been weighed. The possibility of departure thus has conditions of its own, distinct from a report that a departure actually occurred.

Try separating the questions

Work with a copy of the separate BCP 14 scenario 01, keeping the file header. Start each trial anew.

1. Ask about `requirement_satisfied` instead of `positions()`. Before running, record the expected kind and state of the answer. Then consider `explain`: which rule linked the observed behaviour to the denial of fulfilment?

2. Delete `behaviour_deviates`, then in a separate copy replace it with `behaviour_conforms`. For each variant check both the position and the fulfilment assertion. Explain why two `ACTIVES` come with different truth answers.
3. Delete only `claims_conformance`. Check the number of positions through `explain` and find the missing premise of the rule that establishes `subject_to`.
4. Return to the `Syzdyq` example. To consider reading by a particular person, name the duty's bearer, the required behaviour, and the information about how that person read the word aloud. Mark what is taken from the text and what must be added to describe that instance of reading.

Now we can separately record a requirement, its bearer, and the information about behaviour. One question remains that was so far taken as settled: where did the information itself come from, and on what ground is it trusted? The next chapter is devoted to that.

CHAPTER 11

Who Reported It and What It Confirms

In the last chapter the information about behaviour was already present in the case. We asked what follows from it and how it bears on the duty. Now take one step back: why should the starting assertion be accepted at all?

Imagine an employee asking to be reimbursed for a purchase. He presents an approval, a receipt, and a bank confirmation. At first glance everything is in place. But the confirmation concerns another purchase. The document may be genuine, its content truthful, and yet it may leave the payment we need to establish unconfirmed.

«Checking a credential does not imply assessing the truth of the statements it contains.»

W3C · Verifiable Credentials Data Model 2.0 · §2

The edition's English rendering; consult the source for its wording. © 2025 W3C. Commentary for this software tutorial; the document's own licence applies.

The standard describes digital credentials. What is useful here is the distinction itself between checking a document and assessing assertions; the Arxo mechanism is arranged separately.

The purchase we are checking

Take the study package expense–evidence. It states a rule for a hypothetical organization: an expense meets the reimbursement conditions if the purchase is approved, its amount and payment are established, the employee who incurred the expense is known, and the amount does not exceed 10,000 roubles³¹.

A purchase has the type `Purchase`, an employee `Employee`, and a monetary amount the type `Money`, which pairs an amount with a currency. The rule looks like this:

³¹The organization's conditional example: the package description and the Reimbursement rule in `expense-evidence.law`. The money limit and conditions are stated for this teaching case only.

```
rule Reimbursement strict {
  for p: Purchase;
  for e: Employee;
  for a: Money;
  when approved(p)
    and expense_amount(p, a)
    and paid(p)
    and incurred_by(p, e)
    and a <= REIMBURSEMENT_CAP;
  then reimbursable(p, e);
}
```

REIMBURSEMENT_CAP is a constant with the value 10000 RUB. The relation incurred_by(p, e) links the purchase to the employee. Three other premises speak of approval, amount, and payment: approved, expense_amount, paid.

The conclusion `reimbursable(p, e)` means the expense matches the recorded reimbursement conditions. It does not report that the money has already been transferred to the employee. As in the last chapter, the ground for acting and the act itself differ.

In our example the purchase has the identifier `urn:demo:purchase:42`, the employee `urn:demo:employee:7`. For brevity we will call them purchase 42 and employee 7. The link between them is recorded with an ordinary assert `incurred_by(...)`. Approval, amount, and payment require documentary grounds.

A document is not yet a payment confirmation

Consider the bank document B92 from the right-payment scenario. Inside given it is described by a separate block:

[illegible]

After evidence come the name B92 and the evidence kind BankConfirmation. The body lists its particulars. `id` states the entry's persistent identifier, `issuer` names the document's issuer, `content_hash` holds the fingerprint of its content. The dates say when the information was observed and recorded; `valid` states the period during which the document is in effect.

The last four fields describe the content: a bank confirmation, purchase 42, employee 7, the amount of 10,000 roubles. That is the study document's structured entry; no scanned text is recognized in this example. The hashes are left in their original form so the excerpts can be matched exactly against the files.

The document already carries status verified: it is marked as checked. But that one field is not enough for acceptance. The verification result is recorded separately:

```
verification VB92 {  
    evidence B92;  
    document_hash "sha256:92929292929292929292929292929292929292929292929292929292929292929292929292929292929292";  
    verifier "urn:demo:verifier:audit-desk";  
    outcome verified;  
    recorded @2026-02-10T10:00:03Z;  
}
```

verification VB92 refers to B92 and names the verifier through `verifier`. `outcome` states the positive result, recorded the moment of recording. The `document_hash` value must match the document's `content_hash`. That is how the verification result is tied to particular content.

The model works with the supplied verification result: it does not itself call the bank or certify the signature. Besides the hash match, what counts is whether the verification record was available at the knowledge time. The admissible verifier is named by the package's rules³².

Now state what B92 is offered as evidence for:

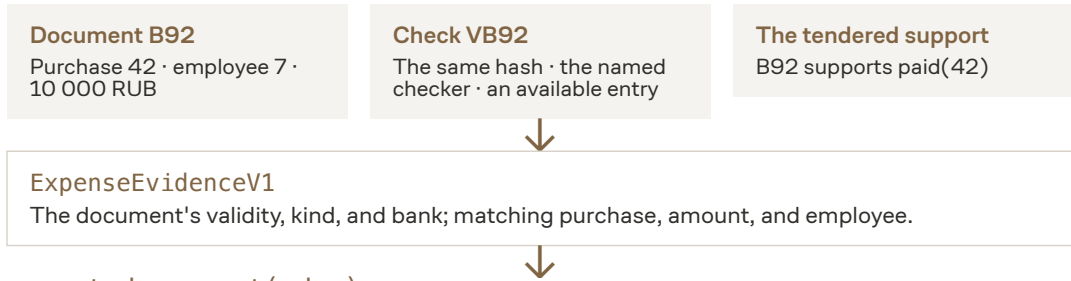
```
support B92 supports paid(entity_ref("urn:demo:purchase:42"));
```

This declares a support link for the paid assertion about purchase 42. After support comes the document; after supports, the assertion it is offered to support. So far we have only proposed the link: "this document is offered to confirm this purchase's payment". Whether to admit or reject such a support is still to be decided.

³²The Arxo specification, §78.2.2–78.2.4: tying a verification result to a document's content, knowledge time, and admission policy. The external verification result enters the model as input information.

We now have three separate entries: what the document says, the result of verifying it, and the assertion it is offered to support. Because they are separate, each link can be examined on its own.

From a document to admitted support



The arrows show the checking of links, not chronology. The final state of paid also depends on other admitted supports.

How a support link is admitted

The package holds an evidence-admission policy, ExpenseEvidenceV1. A policy here means a declared set of requirements for admissible supports. The scenario chooses it in the context:

```

context {
  legal_time @2026-03-01;
  decision_time @2026-03-01T00:00:00Z;
  knowledge_time @2026-03-01T00:00:00Z;
  timezone "UTC";
  evidence_policy ExpenseEvidenceV1;
}
  
```

Leave the dates as in the original example. The new line `evidence_policy ExpenseEvidenceV1` says which admission policy applies in this calculation.

In the policy declaration's body there are two lines stating its main task. An excerpt follows:

```

accepts accepted_support;
protects approved, expense_amount, paid;
  
```

`protects` lists the protected relations. Under the chosen policy they cannot be confirmed by an ordinary assert entry: an admitted support is needed. `accepts` names the relation expressing the admission decision. If the policy's rules conclude `accepted_support` for the given support, it is allowed to back the assertion.

For such rules to read the document, the policy binds its fields and the support's properties to relations in its own vocabulary. In the full declaration the lines `input`, `field`, `arg`, and `target` serve that purpose. Here an edge is the support link itself. Use the following table as a map while reading the next two rules; the names fall into four groups:

Names	Which information they stand for
ev_edge, ev_status, ev_authentic, and other ev_...	The support's link to the document, its status, verification, and availability
doc_kind, doc_purchase, doc_amount, doc_employee	The document's kind and content
edge_is_paid, edge_purchase	What exactly this support is offered to confirm
claim_amount, claim_employee	The amount and employee established for the purchase by a matching receipt

First the document itself is checked:

```
rule SoundDocument strict {
  for d: Text;
  when ev_available(d)
    and ev_current(d)
    and ev_status(d, "verified")
    and ev_authentic(d, TRUSTED_VERIFIER, "verified");
  then sound_document(d);
}
```

It must be available at the knowledge time, be valid on the chosen date, carry the verified status and a verification result from `TRUSTED_VERIFIER`. In this package the constant names the audit-desk verifier. The document's status is one of the premises; it does not replace the other three.

Then the document's link to the payment is considered:

```

rule AcceptPayment strict {
  for edge: Text;
  for d: Text;
  for p: Purchase;
  for a: Money;
  for e: Employee;
  when ev_edge(edge, d, "supports")
    and edge_is_paid(edge)
    and edge_purchase(edge, p)
    and sound_document(d)
    and doc_kind(d, "bank_confirmation")
    and doc_issuer_ok_bank(d)
    and doc_purchase(d, p)
    and doc_amount(d, a)
    and claim_amount(p, a)
    and doc_employee(d, e)
    and claim_employee(p, e);
  then accepted_support(edge);
}

```

The variable `edge` denotes the support's address, `d` the document's. The first three premises establish that the document is tendered as confirmation of the payment for purchase `p`. The next ones check its validity, kind, and issuing bank.

Then comes the crucial match. The variable `p` in `doc_purchase(d, p)` is the same variable as in `edge_purchase(edge, p)`. So the purchase in the document must coincide with the purchase in the assertion. Likewise one variable `a` links the amounts, and `e` the employee. Both amount and employee must also match those established for this purchase by the receipt.

Only once all conditions hold does the rule conclude `accepted_support(edge)`. A particular support is admitted: one and the same document may serve for one assertion and not for another.

Admission applies to a particular link

`accepted_support` concerns a particular support `edge`. The purchase, amount, and employee are matched for the asserted claim; the document does not become a universal confirmation.

A genuine document about someone else's purchase

In the wrong-payment scenario, approval A17, receipt R25, and bank confirmation B91 are tendered. The first two documents concern purchase 42. B91 carries a positive verification result, but its purchase field states purchase 99. And the support offers it to confirm paid for purchase 42.



The image may be genuine. But does it show this patient?

The `AcceptPayment` rule fails at the purchase match: `doc_purchase(d, p)` cannot hold for purchase 42 because B91 names purchase 99. The B91 support is not admitted. The `paid(42)` assertion stays `NEITHER`, as does the reimbursement conclusion.

A mismatched document does not prove the payment never happened.

In `right-payment`, B92 about the needed purchase is added. Its support passes the rule. Now the payment receives `TRUE_ONLY`, and together with the remaining premises it lets us conclude `reimbursable(42, 7)`.

The package's demonstration shows each support's admission and the refusal's reason. Start it from the reader companion directory:

```
python3 packs/demo/expense-evidence/run_demo.py
```

In the first scene find B91 and the mismatched `doc_purchase`. In the second, the admitted B92 support. The following scenes show the confirmation's removal and the case where, after one document's removal, another matching support survives.

The tests themselves start like this:

```
law engine test packs/demo/expense-evidence/tests/reimbursement.lawtest --program \
  packs/demo/expense-evidence/clir/expense-evidence.lawir.json --imports \
  contexts/expense-evidence.json
```

For a separate explanation, keep in a copy the file header and one test right-payment. Replace the reimbursement question with the payment question:

```
evaluate truth(paid(entity_ref("urn:demo:purchase:42")));
expect result_kind == PROPOSITION;
expect truth_status == TRUE_ONLY;
expect evaluation_status == COMPUTED;
```

Hand that copy to the `explain` command with the same program. The output's start will be as follows; further grounds are omitted here:

```
paid(42): TRUE_ONLY
  support_admission paid(42)
```

Author's paraphrase of the engine output above (the package declares no English labels, so the output holds bare identifiers; what follows is the author's free retelling): payment for purchase 42 is established (TRUE_ONLY); the support_admission node marks the support's admission — the tendered bank confirmation passed the AcceptPayment rule with its premises, including the entries from the bank confirmation and the receipt.

The path to `paid(42)` therefore runs through both the tendered document and the rule that admits its support³³.

When confirmation is missing or conflicts

Delete from the right-payment copy only verification VB92. B92 itself and its support stay. Even the status `verified` inside the document survives. Yet `SoundDocument` now lacks the separate verification record, and the payment returns to `NEITHER`.

The third starting test tries to establish the payment directly, with no bank confirmation:

```
assert paid(entity_ref("urn:demo:purchase:42")) {
  id "assert-bare-paid";
  origin adjudicated;
}
```

Even `origin adjudicated` does not replace an admitted support for a protected relation. In this case `paid` and the reimbursement stay `NEITHER`. The policy imposes requirements on the grounds, and this assertion is subject to them too.

³³The whole `SoundDocument`, `AcceptPayment`, and `AcceptPaymentReversal` rules are in the package model; the document compositions in the scenarios. The admission report and its grounds are described in §78.2.6–78.2.9.

A conflict is also possible. The rule `AcceptPaymentReversal` from the same package admits a bank document of the `bank_reversal` kind as a refutation of payment. For such a link refutes is recorded instead of supports. If both supports are admitted for one purchase, `paid` receives `BOTH`.

That is this package's way of presenting a payment's reversal. The model keeps both supports of one assertion; the events' order and the later state's choice are not described here. So a refutation's appearance does not automatically delete the earlier confirmation.

Refutation and a change of state

BOTH means two admitted supports are present. To describe "paid before, reversed now", separate rules of time and state change would be needed.

Which grounds are admitted

Payment's ground	paid	Reimbursement conditions
Only a document about someone else's purchase The purchase does not match.	NEITHER	NEITHER
A matching admitted support The remaining reimbursement premises are established too.	TRUE_ONLY	TRUE_ONLY
A matching document with no verification entry SoundDocument does not apply.	NEITHER	NEITHER
Confirmation and refutation both admitted The paid(p) premise requires TRUE_ONLY.	BOTH	NEITHER

The comparison concerns one teaching case. Neither the absence of a support nor a conflict creates a denial of the reimbursement conditions.

In the last row the reimbursement rule's positive premise `paid(p)` is not established without contradiction. Not deriving the conclusion does not amount to denying reimbursable. We return to the familiar four states, but now we can trace them to documents.

Try checking the ground

Start each trial with a fresh `right-payment` copy, keeping the file header and the remaining documents.

1. Delete verification VB92, keeping the old TRUE_ONLY expectation. Account for the mismatch. Then check the expected NEITHER separately for payment and reimbursement.
2. Restore the verification and delete only the support B92 line. Why does the document with correct particulars no longer establish the payment?
3. Add a separate document about a payment reversal of the bank_reversal kind, with its own name, address, and matching verification record. Keep the purchase, amount, and employee. Use AcceptPaymentReversal as your checklist: check its document and matching premises, including the verifier and the hash linking the verification to the new document. Tender it through refutes for paid. Predict both result lines: the payment and the reimbursement conditions. Then delete B92 with its verification and support: which state remains for the payment?
4. Return to Kafka's facts with evidence VDG_1915_TEXT, such as the assertion that the man requested entry. They state evidence but hold no policy decision shown here about admitting a support. Which links would have to be described to check such a fact's grounds in a similar way?

The number of documents does not replace checking their content.

Each support must answer its own question and pass the stated admission conditions. One more link remains to be considered in detail: exactly where in the source we can find the text that supports the model. That is the subject of the next chapter.

PART IV

Back to the Source

CHAPTER 12

Where Does the Source Say That?

In the previous chapter we checked whether a document confirms a fact. But rules need grounds too. When someone receives a model, they may ask: where does the source say what appears here as a condition or a conclusion?

In Chapter 4 we already traced an explanation to a textbook fragment. Now we follow that path all the way to the saved text. We need to separate four things: the document itself, its edition, its publication, and the chosen place inside it. Then we can check the reference, discuss the rule's meaning against the shared source, and ask how much of that source the rules cover.

From the rule to the words

Let us return to BCP 14 from Chapter 10. Here is the rule that links a requirement containing MUST to the specification's absolute requirements:

```
@source(RFC2119_S1)
rule MustDenotesAnAbsoluteRequirement defeasible {
  label en official "MUST: this word means that the definition is an absolute
  requirement of the specification";
  for r: Requirement;
  when keyword_used(r, Must);
  then absolute_requirement(r);
}
```

The opening `@source(RFC2119_S1)` names the ground of the rule. It points to a declared fragment. It tells us which text to match the rule against; on its own it does not prove the match is correct³⁴.

In `sources.law`, find that name:

³⁴The Arxo specification, §§27–28: document, edition, publication, and fragment. The BCP 14 declarations live in `sources.law`; the `@SOURCE` reference names the ground for matching against the rule.

```

fragment RFC2119_S1 in RFC2119_EN {
  kind section;
  locator "section/1";
  text en official ""1. MUST   This word, or the terms "REQUIRED" or "SHALL",
mean that the
  definition is an absolute requirement of the specification.""";
  content_hash "sha256:5cab6a6cdaacb5846bf7f4fe518a564097de37d01907ec17a13aa9b
1b34e2dee";
}

```

fragment declares a fragment, RFC2119_S1 names it, and in RFC2119_EN ties it to one edition. Inside are the fragment's kind section, the address section/1, and the exact English text in triple quotes. content_hash holds the checksum of that text.

In plain words: the word MUST, and also the terms REQUIRED and SHALL, mean that the definition states an absolute requirement of the specification. This paraphrase only aids reading; the English fragment stays the rule's original ground.

Note a small difference in scope. The section names three equivalent words, while the rule above handles only Must. The package holds neighbouring rules for Required and Shall with the same reference.

One fragment can ground several rules.

A single reference does not mean that every part of the fragment has been expressed in a rule.

The document and its edition

The fragment has led us to RFC2119_EN. To see which text stands behind that name, read two linked declarations:

```

source RFC2119 {
  kind best_current_practice;
  jurisdiction IETF;
  authority IETF;
  number "RFC 2119, BCP 14";
  label en official "Key words for use in RFCs to Indicate Requirement Levels"
;
}

```

source presents the document as a work: here RFC 2119 with its number and title. kind states its type; jurisdiction and authority both name the IETF. This entry identifies the document, but it contains no file to read.

```

edition RFC2119_EN of RFC2119 {
  label en official "RFC 2119 (S. Bradner)";
  language en;
  officiality official;
  adopted @1997-03-01;
  in_force [@1997-03-01, infinity);
  materialization_status PINNED_OFFICIAL_BYTES;
}

```

edition identifies a particular version of the document in a given language. The of RFC2119 link states which document it belongs to. Here language en sets English, and officiality official marks official status.

The adopted and in_force fields hold the model's dates. RFC 2119's own header states March 1997, without a day. The first of the month is an explicit formalization convention here. That entry does not show that the source itself gives the date as 1 March exactly. The next chapter examines how dates affect the answer; for now note where the dates come from.

materialization_status PINNED_OFFICIAL_BYTES means the edition claims to keep the bytes of the official publication. This mark describes the origin of the pinned material. Check it against the publication and the file itself: the status name alone does not certify the source.

Where the text is

The concrete published form is set by publication:

```

publication RFC2119_EN_TEXT of RFC2119_EN {
  media_type "text/plain; charset=us-ascii";
  uri "https://www.rfc-editor.org/rfc/rfc2119.txt";
  retrieved_at @2026-08-30T09:00:00Z;
  content_hash "sha256:3c2ceb7bfc84cd34720f4a5271338ab9d8280d34bdd1eb250c64306202f2ed8b";
  local_path "sources/rfc2119/en.txt";
}

```

The of RFC2119_EN link points at the edition. media_type describes the file's format, uri stores the publication's address, retrieved_at the recorded moment of retrieval. local_path shows where the saved copy lies inside the package.

Different formats of the same content have separate publication records. The text file and the PDF can present the same edition yet differ byte for byte. Each file has its own checksum.

For our file, run from the reader companion directory:

```
python3 - <<'PYTHON'
from hashlib import sha256
from pathlib import Path
source = Path("corpus/laws/org/ietf/bcp14/sources/rfc2119/en.txt")
print(sha256(source.read_bytes()).hexdigest())
PYTHON
```

Compare the result with the publication's `content_hash`, without the `sha256:` prefix. A match confirms that the file agrees with the stated fingerprint. It does not establish that the edition was chosen correctly or that its words were read correctly.

Two checks

Computing SHA-256 checks whether a file matches its recorded fingerprint. Checking the meaning requires comparing the rule's premises and conclusion with the source passage.

Do not confuse the two checksums in this chapter. The publication's checksum covers the whole file, headers and layout included. The fragment's checksum covers only the text between the triple quotes. They answer different questions, so they differ³⁵.

Document, edition, publication, and fragment

source · document
RFC2119
Which work is named



edition · edition
RFC2119_EN
Which content and language are chosen

The publication and the fragment belong to RFC2119_EN:

publication · file
RFC2119_EN_TEXT
sources/rfc2119/en.txt
SHA-256 of the whole file

fragment · excerpt
RFC2119_S1
section/1
SHA-256 of the fragment text

`@source(RFC2119_S1)`

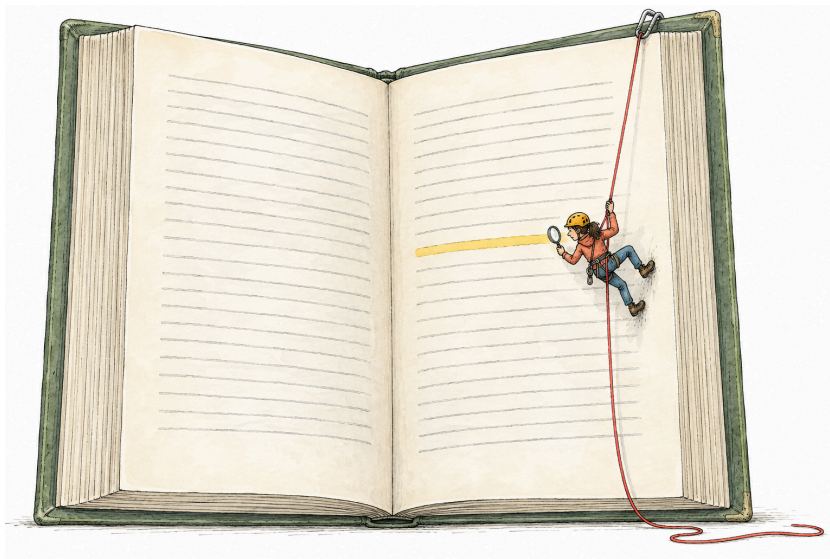
The rule refers to the fragment. Its text is checked against the saved publication.

The diagram shows links between declarations. The file's checksum and the excerpt's checksum refer to different byte sequences.

What counts as a place in the text

Open the saved `en.txt` and find section 1. The `RFC2119_S1` text should occur in it verbatim. The `section/1` address names the place, and the matching words let us check the excerpt itself.

³⁵The pinned RFC 2119 publication and the official text. The `sync_bcp14_fragments.py` script with the `--check` flag compares the declarations against the local publications' fragments without writing.



Finding the book is the first step. Now find the particular line.

The fragment's boundaries also need attention. In the RFC the document numbers the sections itself. In other materials we have already read, the division into fragments works differently:

Material	Source unit chosen in the package
RFC 2119	Section 1, locator section/1
Syzdyq	Paragraph §33, locator article/33
Hilbert	Axiom I, 1, locator section/1
Kafka	The selected passage about the open gates, locator erzaehlung/4

The same word section in a locator does not turn an axiom into an RFC section. It is a convention for locating passages within the package. In Kafka the passage number likewise belongs to the model's markup: the parable has no authorial section "erzaehlung/4".

An address inside the model

A locator is read together with the edition and the markup convention. section/1 alone is not enough: the same locator may occur in different documents.

When handing over the model, explain both the unit's name and how its boundaries were chosen. If a translation is used, name its language and edition. For example, the package's Hilbert fragment points to E. J. Townsend's English translation, while

the Syzdyq fragment keeps the Kazakh text. The book's English paraphrase helps read them but does not replace the pinned ground.

The BCP 14 package supplies a check that the excerpts agree with the saved publications:

```
python3 corpus/tools/sources/sync/sync_bcp14_fragments.py --check
```

It compares the declaration file against fragments extracted from the local texts. The check matters precisely because a mismatch is possible. A successful run reports that the checked files agree; whether the rule faithfully expresses the text is a separate judgment, still to be made against the source.

Two sources in one package

In BCP 14, RFC 8174 stands next to RFC 2119. Its header states Updates: 2119. It refines the use of the key words, while RFC 2119 stays a separate document. So the package has two sources, two English editions, and two text publications.

Section 2 of RFC 8174 quotes the old and the new text of RFC 2119's paragraphs. The refinement: the special BCP 14 meanings belong to words in all caps. Lowercase forms keep their ordinary English meaning.

Consider the rule for the lowercase form:

```
@source(RFC8174_S2)
rule LowercaseKeywordCarriesNoDefinedMeaning defeasible {
    label en official "when these words are not capitalized, they have their nor
mal English meanings and are not affected by this document";
    for r: Requirement; for k: Keyword;
    when keyword_written(r, k, Lowercase);
    then not keyword_used(r, k);
}
```

The reference now leads to RFC8174_S2 — a fragment of the other document. Within RFC 2119 alone, this rule has no ground.

Note precisely what the conclusion is about. The rule denies keyword_used: the word's special BCP 14 meaning. It does not mark every lowercase must as optional; the word keeps its ordinary English meaning. RFC 8174 itself explains that normative text can do entirely without these key words.

«Normative text does not require the use of these key words.»

B. Leiba · RFC 8174 · §2

The edition's English rendering; consult the source for its wording.

Key words are only one way to express a requirement. That is why the result about keyword_used cannot be carried over to the normative sense of the whole text.

A separate scenario 03 asks about the special meaning of the lowercase word and receives FALSE_ONLY:

```
law engine test \
  corpus/laws/org/ietf/bcp14/tests/03-lowercase-keyword-carries-no-force.lawtest
\
  --program corpus/clir/ietf-bcp14.lawir.json --imports contexts/ietf-bcp14.jso
n
```

The package also holds an explicitly selectable disputed reading, PreRfc8174Practice. Selecting it changes some answers. Chapter 14 examines that construction; here it is enough to separate the saved text from the rules that express one reading.

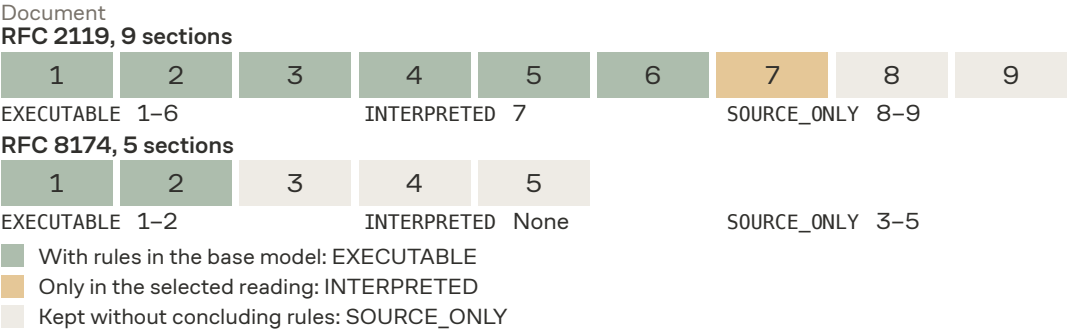
How much of the source the rules cover

Now we can ask about coverage. First, define what we count: the numbered sections of each RFC. RFC 2119 has nine, RFC 8174 five. Headers, annotations, and page layout fall outside this count. Still, RFC 2119's annotation is kept as a separate fragment, because RFC 8174 updates its text.

A section-by-section check gives this picture³⁶. EXECUTABLE marks sections with rules in the base model, INTERPRETED those with rules only in the selected reading, and SOURCE_ONLY those kept as text without concluding rules:

³⁶The unit of counting is the numbered section. The reasons for keeping without rules are listed in the accounts of RFC 2119 and RFC 8174. The classification by links with rules is computed by `check_formalization_depth.py`; a share of sections does not measure the completeness of their meaning.

Coverage across numbered sections



One cell is one numbered section. The colour shows the manner of presentation, not the correctness of interpretation.

For RFC 2119 the first group's share is 6 of 9, about 66.7%. For RFC 8174 it is 2 of 5, or 40%. These are shares of numbered sections linked to the base model's rules. They do not measure how much meaning was read correctly, nor how completely each section was analysed.

Here are the remaining sections. RFC 2119's section 7 covers security matters; its rules apply only under a separate reading. Sections 8 and 9 hold acknowledgements and the author's address. In RFC 8174, sections 3 and 4 report that no IANA actions are required and there are no related security matters, and section 5 holds the normative reference. These parts are kept and named, so that the boundary of the formalization is stated rather than left silent.

A coverage percentage does not replace this list.

Even a section in the EXECUTABLE group may state a condition the rule misses. And a missing rule for acknowledgements has a different reason than a missing rule for a substantive requirement. To judge coverage, look at both sides: which provisions appear, and what stayed outside the boundary.

Try walking the route yourself

1. Find `MustNotDenotesAnAbsoluteProhibition`. From its `@source` go to the fragment, then find the same words in the saved publication. Explain the text's meaning and how the rule records it, each separately.
2. Copy RFC 2119's `en.txt` into a temporary file. In section 1 replace `absolute` requirement with `optional` requirement. Compare the checksums of the original and the copy. Will the unchanged `RFC2119_s1` text still occur in the altered file?

3. Match RFC 2119's sections 7–9 against the coverage table. How does selecting a reading differ from keeping a section without rules? Why is one number not enough?
4. Take a short text from the prologue. Name the document itself, the specific edition, and the available publication. Choose one fragment. Mark which boundaries and dates come from the source and which you propose for the model.

We have traced the path from the rule back to the words that both sides of a discussion can read. One question still needs to be made precise: as of which date does this text ground the answer? The next chapter starts there.

As of Which Date?

In the previous chapter we found the edition and the saved text. But a rule taken from that text may apply only within a particular interval. The same question can get different answers before and after a definition changes.

Take the kilogram. In the 8th edition of the SI Brochure the kilogram is defined through the mass of the international prototype. The new definition, built on the fixed value of the Planck constant, took effect on 20 May 2019. Imagine asking from one fixed later moment which definition applied on the day before that date and on the day of the change itself. The moment at which we ask stays the same; the date we ask about changes³⁷.

Dates and moments in time

The case context already gave us three time fields. Read them one by one:

Field	What it specifies
<code>legal_time</code>	The date for checking which rules and editions apply
<code>knowledge_time</code>	The cutoff time for information available to the calculation
<code>decision_time</code>	The time of the decision represented in the context

Despite the name `legal_time`, this example concerns the definition of a unit of measurement. The field specifies the date for checking which editions and rules apply; it is not confined to legal texts.

In our examples `legal_time` is a calendar date: `@2019-05-19`. The other two fields hold a moment, for example `@2026-09-08T12:00:00Z`. After the `T` come hours, minutes, and seconds; `Z` denotes UTC. A calendar day and a moment inside that day are different values.

We will first change `legal_time` while holding the other fields fixed. Then we will return to the payment example and change `knowledge_time` to see when a verification becomes available.

³⁷Resolution 1 of the 26th CGPM (2018) sets the start of the new SI's effect from 20 May 2019. The definitions are kept in the 8th edition, §2.1.1.2 and the 9th edition, §2.3.1.

One definition on both sides of the boundary

The SI package has two relations for our question. `defined_by_reference` links the unit to a physical artefact; `defined_by_constant` links it to a defining constant. The names `Kilogram`, `InternationalPrototypeKilogram`, and `PlanckConstant` are already declared in the vocabulary.

Start with the prototype question. Take a copy of scenario 01a and hold `decision_time` and `knowledge_time` at 8 September 2026. Set `legal_time`, the date we are asking as of, to 19 May 2019:

```
test "urn:query:si-book-prototype-before" {
  given {
    context {
      decision_time @2026-09-08T12:00:00Z;
      knowledge_time @2026-09-08T12:00:00Z;
      legal_time @2019-05-19;
      timezone "UTC";
    }
  }
  evaluate truth(defined_by_reference(Kilogram, InternationalPrototypeKilogram));
  expect truth_status == TRUE_ONLY;
  expect evaluation_status == COMPUTED;
  expect applied(KilogramByInternationalPrototype);
}
```

`given` carries no extra assertions. The needed information already sits in the package, in the table of base units. Here is its kilogram row:

```
assert base_unit(Kilogram, Mass, "kg") { id "t2-kg"; }
```

It links the kilogram to mass and the symbol `kg`. On 19 May the rule `KilogramByInternationalPrototype` applies and supports the `TRUE_ONLY` answer. The `applied(...)` expectation checks that application.

Now change only `legal_time` in the copy:

```
context {
  decision_time @2026-09-08T12:00:00Z;
  knowledge_time @2026-09-08T12:00:00Z;
  legal_time @2019-05-20;
  timezone "UTC";
}
```

The question, the program, and the available information stay the same. Yet on 20 May the prototype answer becomes `NEITHER`. The old expectations then fail in two

places: there is no positive answer and no application of the rule. This is not a failed calculation: its status stays `COMPUTED`.

This does not claim the prototype's mass changed or the kilogram ceased to exist. We asked whether the unit is defined through that artefact as of the chosen date.

The ground for such a conclusion no longer holds.

The example has no separate negative rule, so the result is `NEITHER`, not `FALSE_ONLY`.

The edition's interval

See where the time boundary is written. The package keeps the English texts of two editions:

```
edition SI_BROCHURE_8_EN of SI_BROCHURE {
  label en official "The International System of Units (SI), 8th edition, 2006
– English text of the bilingual edition";
  label ru unofficial "8-е издание (2006), английская часть двуязычного издани
я";
  language en;
  officiality official;
  adopted @2006-01-01;
  in_force [@2006-01-01, @2019-05-20];
  materialization_status PINNED_UNOFFICIAL_COPY;
}
```

```
edition SI_BROCHURE_9_EN of SI_BROCHURE {
  label en official "The International System of Units (SI), 9th edition, 2019
, version 4.01";
  label ru unofficial "9-е издание (2019), версия 4.01, английский текст";
  language en;
  officiality official;
  adopted @2018-11-16;
  in_force [@2019-05-20, infinity);
  materialization_status PINNED_UNOFFICIAL_COPY;
}
```

In `in_force` the left square bracket includes the start; the closing parenthesis excludes the end. So the eighth edition's interval ends before 20 May. The ninth edition's interval starts that day; `infinity` means no set end.

On the boundary itself the intervals do not overlap.

adopted and `in_force` do different jobs: a change can be adopted before it takes effect. The saved Brochure reports agreement in November 2018 and effect from 20 May 2019. For our question the second date matters.

«...from 20 May 2019, the International System of Units, the SI, is the system of units in which...»

26th CGPM · Resolution 1 (2018)

The edition's English rendering; consult the source for its wording.

The resolution goes on to fix the values of the seven defining constants. The date “from 20 May” sets the start of the new SI's effect; the moment the document is read may be different.

Distinguish the dates assigned by the model from the dates reported by the publication. @2006-01-01 on the eighth edition does not show that the Brochure appeared on 1 January. And the saved version 4.01 of the ninth edition contains later updates: its interval does not mean all that text already existed in May 2019. Here we check only the kilogram definition's boundary.



The night shift ends, the morning shift begins. The boundary decides whose turn it is to act.

The rule's interval

The prototype rule also carries a time boundary:


```
@source(SI8_DEF_KILOGRAM)
rule KilogramByInternationalPrototype strict {
  label en official "8th ed., §2.1.1.2: the kilogram is equal to the mass of t
he international prototype of the kilogram (until 20 May 2019)";
  label ru unofficial "8-е изд., §2.1.1.2: килограмм равен массе международног
о прототипа килограмма (до 20 мая 2019)";
  for q: BaseQuantity;
  for s: Text;
  when base_unit(Kilogram, q, s);
  then defined_by_reference(Kilogram, InternationalPrototypeKilogram);
  effective [-infinity, @2019-05-20);
}
```

The new effective line sets this rule's applicability interval. The left `-infinity` leaves its start open; the right boundary excludes 20 May 2019. The link to the source's edition still holds: that open start does not automatically make the rule applicable before the edition's own interval³⁸.

The conditions must hold together

All three are needed here at once: an applicable edition of the source, an effective interval of the rule, and satisfied premises. `effective` does not cancel the source's time limits.

The new definition's rule mirrors it:

```
@source(SI9_DEF_KILOGRAM, SI9_APP1_2018_INDEX)
rule KilogramByPlanckConstant strict {
  label en official "$2.3.1: the kilogram is defined by taking the Planck cons
tant h to be 6.626 070 15 × 10-34 J s (from 20 May 2019)";
  label ru unofficial "$2.3.1: килограмм определён фиксацией постоянной Планка
h = 6,626 070 15 × 10-34 Дж·с (с 20 мая 2019)";
  for q: BaseQuantity;
  for s: Text;
  when base_unit(Kilogram, q, s);
  then defined_by_constant(Kilogram, PlanckConstant);
  effective [@2019-05-20, infinity);
}
```

In both rules, `base_unit(Kilogram, q, s)` finds the familiar table row. The conclusion, the source, and the time interval differ. The date tells us which ground applies; it changes nothing about the constant or the artefact.

Check both questions on both dates:

³⁸The Arxo specification, §89 — context, §92 — edition applicability. In the example the intervals of the editions and the rules are reconciled; the result does not isolate the effect of only one of these constraints.

One boundary, two different questions



An empty dot means the end is excluded. A filled dot means the start is included.

Applicability date	19 May 2019	20 May 2019
Through the international prototype	TRUE_ONLY	NEITHER
Through the Planck constant	NEITHER	TRUE_ONLY

decision_time and knowledge_time: 08.09.2026, 12:00 UTC.

The applicability boundary of 20 May is shown. The results concern two relations and carry the COMPUTED status; the new definition is not a negation of the earlier relation.

All four results carry the COMPUTED status. These are answers to two different relations, not positive and negative support for one claim. So the neighbouring definition alone does not turn NEITHER into FALSE_ONLY.

Checking and explaining the boundary

The companion's SI package already includes its pinned unit definitions. Make a working copy for the date experiments, starting in reader/:

```
mkdir si-study
cp -R corpus/laws/org/bipm/si-brochure/. si-study
```

Run the file containing the four kilogram-redefinition scenarios:

```
law test si-study --file tests/01-kilogram-redefinition.lawtest
```

--file selects this scenario file from the package's declared checks. The four original scenarios should pass. In them the other time fields change together with the applicability date. The copies above separate them, isolating the effect of legal_time.

For your own experiment, edit the copy at si-study/tests/01-kilogram-redefinition.lawtest. Keep the header and one test, then change the named time field and run the same command again.

To examine why a definition gives no support, use the question preparer from Chapter 4 on that single-test copy, with the companion's SI program and contexts/bipm-si-brochure.json. Run law engine why-not on the prepared question. For the prototype on 20 May, the explanation names KilogramByInternationalPrototype

and the SI_BROCHURE_8_EN edition, closed as of that date. For the Planck-constant question on 19 May, it points at SI_BROCHURE_9_EN: that edition has not yet taken effect.

The base-unit premise is not refuted here. The reason lies in the source's applicability. A rule outside its applicability interval differs from a rule missing a premise, even when both yield the same NEITHER.

When the information became available

Now keep the applicability date and change another field. Return to the purchase from Chapter 11. In the right-payment scenario the verification of the B92 bank confirmation reads:

```
verification VB92 {  
    evidence B92;  
    document_hash "sha256:929292929292929292929292929292929292929292  
9292929292";  
    verifier "urn:demo:verifier:audit-desk";  
    outcome verified;  
    recorded @2026-02-10T10:00:03Z;  
}
```

Its result was recorded on 10 February 2026 at 10:00:03 UTC. Take the purchase information as of 6 February and keep the decision moment at 1 March. For the first copy set this context:

```
context {
  legal_time @2026-02-06;
  decision_time @2026-03-01T00:00:00Z;
  knowledge_time @2026-02-10T10:00:02Z;
  timezone "UTC";
  evidence_policy ExpenseEvidenceV1;
}
```

Document B92 is already available and valid on the chosen date, but its verification is recorded one second later. The policy cannot use that result yet³⁹. Payment and reimbursement both receive NEITHER.

Now change only knowledge_time to @2026-02-10T10:00:03Z. The verification record is available, the support is accepted, and both answers become TRUE_ONLY. The payment itself does not happen again: what changes is the body of information available when answering about that purchase.

³⁹The Arxo specification, §78.2.3 — policy inputs and verification availability. In the right-payment scenario entry VB92 has recorded 10.02.2026, 10:00:03 UTC; in the experiment only the knowledge moment changes.

One second changes the available grounds

knowledge_time · 10 February 2026

What is available and what is derived	10:00:02 UTC	10:00:03 UTC
Document B92	Available	Available
Check VB92	Not yet available	Available
paid	NEITHER	TRUE_ONLY
reimbursable	NEITHER	TRUE_ONLY

legal_time: 06.02.2026 · decision_time: 01.03.2026, 00:00 UTC.

At 10:00:03 the VB92 entry becomes available. The payment and the case date are preserved; the document check's result appears.

For this pair use the right-payment copy from the previous chapter and its program packs/demo/expense-evidence/clir/expense-evidence.lawir.json. The SI vocabulary plays no part here.

These are two different changes. In the kilogram question, legal_time changed while the available materials stayed the same. In the payment question, knowledge_time changed while the case date stayed the same. The verification's recorded time remained fixed; the later cutoff made that record available.

In these experiments decision_time remains the stated time of the decision. It cannot automatically stand in for the other two fields. Nor must the context match the computer's clock at launch: we reproduce a stated situation.

Reproducible context

The scenario sets the time values. A rerun with the same context keeps the stated moments; the computer's current date does not replace them.

Try separating the time fields

1. In the prototype-question copy keep legal_time at 19 May 2019 and move knowledge_time from 8 to 7 September 2026. Then, in a separate copy, move only decision_time to 9 September. Why does the answer stay TRUE_ONLY in both cases?
2. Ask about the Planck constant on 19 and 20 May, keeping the other fields the same. Find the first answer's reason in why-not and the rule's application for the second in explain.
3. In the purchase case compare the knowledge moments 10:00:02 and 10:00:03. Check paid first, then reimbursable. Find the entry that became available between the two calculations.

4. Find the French-language version of the ninth edition of the SI Brochure. Compare its language and interval with the English one. Why does the language difference alone not call for another effective date of the definition?

We can revisit an earlier date while keeping the stated body of knowledge. But time does not settle every disagreement about meaning. If one text admits different readings, name them and consider each separately. That is the next chapter's subject.

CHAPTER 14

One Word, Several Definitions

In the previous chapter the kilogram's definition changed with the date. But different definitions can also coexist. Then the date alone no longer tells us which meaning we intend.

Return to chemistry. In Part I we recorded that a particle donates or accepts a proton. Now ask whether it counts as an acid. Finding the definition in the textbook seems enough. Yet Chemistry 2e presents several definitions, and each requires different information.

Definition	What the model requires to classify a particle as an acid
Arrhenius	The particle dissolves in water and yields hydrogen cations
Brønsted–Lowry	The particle donates a proton
Lewis	The particle accepts an electron pair

This table retells the model's conditions briefly. The textbook explains them through the behaviour of reaction participants; a donor, for example, passes a proton to another particle. We still do not derive that behaviour from the chemical formula. The case states it as information⁴⁰.

«A particle that accepts an electron pair is a Lewis acid.»

OpenStax · Chemistry 2e · §15.2

The edition's English rendering; consult the source for its wording. © OpenStax, Rice University. Adapted under CC BY-NC-SA 4.0; noncommercial use; changes: English rendering.

This definition checks the acceptance of an electron pair. Proton donation belongs to the Brønsted–Lowry definition; their conditions must not be mixed in one question.

So make the question "is it an acid?" precise: under which definition? In Arxo the chosen definition becomes an explicit part of the context, stored alongside the answer.

⁴⁰Chemistry 2e: §14.1 — Arrhenius and Brønsted–Lowry, §15.2 — Lewis. The package's rules apply to reported properties; the chemical formula itself does not produce these premises.

First give the definitions different names

The familiar vocabulary lives in the `openstax.chemistry_acids` package. The `chem.acid_base` package builds its acid and base definitions on it. Here is one of those rules:

```
@source(openstax.chemistry_acids::OS_14_1_BRONSTED)
pub rule BronstedAcidDonatesProton strict {
  label ru unofficial "отдающая протон частица – кислота Брёнстеда–Лоури";
  label en unofficial "a compound that donates a proton is a Brønsted–Lowry acid";
  for s: openstax.chemistry_acids::Species;
  when openstax.chemistry_acids::donates_proton(s);
  then bronsted_acid(s);
}
```

After `when` comes the familiar `donates_proton` relation; after `then`, the new `bronsted_acid`. Once the particle's proton donation is established, the rule derives `bronsted_acid` for that particle.

The long name `openstax.chemistry_acids::donates_proton` says which package the relation comes from. The double colon separates the package name from the declaration name. Chapter 16 explains dependencies; for now recognize the old vocabulary behind this entry. The `@source` reference leads to the pinned fragment of the textbook's section 14.1.

The model holds three separate relations: `arrhenius_acid`, `bronsted_acid`, and `lewis_acid`. Ask about each by name. Explicit links may also connect them: a separate rule, for example, concludes `bronsted_acid` from `arrhenius_acid`. The fact that these names all end in `_acid` creates no such link by itself.

The general question is `is_acid`. Its English label reads: "the species is an acid under the selected reading".

A reading supplies the link from this general question to one specific definition.

How to declare a reading

A reading is a named way of understanding the source, expressed through the model's rules. Here it states which definition answers the general acid question.

Below is the start of the `BronstedReading` declaration and its first inner rule. The other three rules are omitted: `BaseIsBronstedBase` links the base definition to the

general base question; `NotAcidUnderBronsted` and `NotBaseUnderBronsted` provide the negative links. All appear in the package's `02-readings.law` file.

```
pub interpretation BronstedReading of openstax.chemistry_acids::OS_14_1_BRONSTED
{
  status authoritative;
  label ru unofficial "прочтение Брёнстеда–Лоури: кислота – донор протона";
  label en unofficial "Brønsted–Lowry reading: an acid is a proton donor";
  source_anchor openstax.chemistry_acids::OS_14_1_BRONSTED;

  @source(openstax.chemistry_acids::OS_14_1_BRONSTED)
  rule AcidIsBronstedAcid strict {
    label ru unofficial "по прочтению Брёнстеда кислота – кислота Брёнстеда"
;
    label en unofficial "under the Brønsted–Lowry reading, an acid is a Brøn
sted–Lowry acid";
    for s: openstax.chemistry_acids::Species;
    when bronsted_acid(s);
    then is_acid(s);
  }

  // Three more rules are omitted from this excerpt.
}
```

The word `interpretation` introduces the reading, `BronstedReading` names it, and of points at the fragment it construes. The `source_anchor` field records the same source link inside the declaration body. Both identify the text being interpreted; neither selects the reading or supplies the inner rule's premise. That rule says: once `bronsted_acid` is established, conclude `is_acid`.

This rule joins the derivation only when `BronstedReading` is selected. The `BronstedAcidDonatesProton` definition above stands outside the reading. It can conclude `bronsted_acid` whatever is selected. The selection matters for the step to the general `is_acid` question.

The status `authoritative` line records the model's appraisal of the reading. It does not prove the formalization correct, nor does it select the reading on its own. Here all three readings carry that mark, yet a choice is still required.

The readings are gathered into a group for that choice:

```
interpretation_group AcidDefinition {
  alternatives ArrheniusReading, BronstedReading, LewisReading;
  selection exactly_one;
}
```

`alternatives` lists the options, and `selection exactly_one` requires exactly one. This is the package's condition for answering the general question. It does not claim

the definitions exclude each other in chemistry: one particle may satisfy several of them. To compare them, ask the same question several times, selecting a different reading each time⁴¹.



One object, three purposes. Before judging it, name the criterion.

Follow the information to the answer

Take the package's complete check example. The case reports that the hydrogen-chloride particle under study donates a proton. The reading is selected inside context:

⁴¹The Arxo specification, §§152–156: readings, statuses, groups, and selection. The AcidDefinition group and its alternatives are set in 02-readings.law; the mandatory selection concerns the group-dependent question.

```

language "law.core" version "0.2";
package chem.acid_base version "0.1.0";
namespace "urn:chem:clir:acid-base";

test "urn:query:chem-acid-base-02-hcl-bronsted" {
  given {
    context {
      decision_time @2026-09-06T12:00:00Z;
      knowledge_time @2026-09-06T12:00:00Z;
      legal_time @2026-09-06;
      timezone "UTC";
      interpretation BronstedReading;
    }
    assert openstax.chemistry_acids::species_in_question(entity_ref("urn:cas
e:chem:hcl")) { id "ab02-q"; origin case_input; }
    assert openstax.chemistry_acids::donates_proton(entity_ref("urn:case:che
m:hcl")) { id "ab02-1"; origin case_input; }
  }
  evaluate truth(is_acid(entity_ref("urn:case:chem:hcl")));
  expect truth_status == TRUE_ONLY;
  expect evaluation_status == COMPUTED;
  expect applied(BronstedAcidDonatesProton);
  expect interpretations(BronstedReading);
}

```

The `interpretation BronstedReading;` line fixes the question's meaning. `species_in_question` presents the object under study, and `donates_proton` reports its property. The `is_acid` conclusion itself is not among the starting assertions.

The result is the following chain — a simplified diagram, not a literal printout of the explanation:

<code>donates_proton(hcl)</code>	starting information
→ <code>bronsted_acid(hcl)</code>	definition rule
→ <code>is_acid(hcl)</code>	reading rule

In the explanation the last rule is named `BronstedReading/AcidIsBronstedAcid`. That name shows which reading linked the definition to the answer.

The first expectation checks `TRUE_ONLY`, the second the `COMPUTED` status. The next two check the definition rule's application and the selected reading's role. The passing test confirms these expectations on the stated information. The report of the particle's behaviour stays a case condition: the test does not turn it into a chemistry experiment's result.

If no reading is selected

The changes below can first be followed as thought experiments. To run them, use the temporary-copy setup under “Try changing the question” at the end of the chapter.

Delete one line from the example's copy:

```
interpretation BronstedReading;
```

Leave the dates, assertions, and question as they are. For `is_acid` we get `truth_status = NEITHER` and `evaluation_status = INTERPRETATION_REQUIRED`. The conclusion is not established, and the status names exactly what is missing: the selection of a reading.

Contrast this with a case where a reading is selected but the information it needs is missing. Select `ArrheniusReading` in the same case, keeping the proton-donation report. That report does not establish that the particle dissolves in water and releases hydrogen cations. The answer is `NEITHER` with `COMPUTED` status: the question has a definite meaning, but the needed premises are not established.

No chosen meaning, or missing premises?

What was changed in the original example	truth_status	evaluation_status
Kept BronstedReading	TRUE_ONLY	COMPUTED Proton donation suffices for the selected definition.
Deleted the selection	NEITHER	INTERPRETATION_REQUIRED Answering <code>is_acid</code> requires choosing one alternative of the group.
Selected ArrheniusReading	NEITHER	COMPUTED The needed information about behaviour in water is not established.

The facts about `hcl` and the dates are the same in all rows. The same `NEITHER` comes with different reasons and different computation statuses.

In the unselected case `why-not` lists the unselected readings. The missing selection blocks an answer to the general `is_acid` question; the original proton-donation information remains in the case.

Which questions require a choice of reading?

Without choosing BronstedReading, the independent bronsted_acid question may still receive TRUE_ONLY and COMPUTED. The general is_acid question depends on the AcidDefinition group and requires a choice.

Same information, different definitions

Now change the example. In the boron-trifluoride case record that the particle accepts an electron pair and does not donate a proton. Below are all three starting assertions of scenario 10, inside given after the context:

```
assert openstax.chemistry_acids::species_in_question(entity_ref("urn:case:chem:bf3")) { id "ab10-q"; origin case_input; }
assert openstax.chemistry_acids::accepts_electron_pair(entity_ref("urn:case:chem:bf3")) { id "ab10-1"; origin case_input; }
assert not openstax.chemistry_acids::donates_proton(entity_ref("urn:case:chem:bf3")) { id "ab10-2"; origin case_input; }
```

Leave these entries and the dates unchanged. Change only the reading-selection line. For the same is_acid question we get:

One BF₃ case, three separate calculations

Recorded: the particle accepts an electron pair; proton donation is explicitly refuted. No information about behaviour in water is given.

Reading
LewisReading

Answer
TRUE_ONLY

Ground
Electron-pair acceptance
established

Reading
BronstedReading

Answer
FALSE_ONLY

Ground
Proton donation explicitly
refuted

Reading
ArrheniusReading

Answer
NEITHER

Ground
The required information about
behaviour in water is missing

In each calculation exactly one reading is selected; COMPUTED everywhere. These answers do not add up to BOTH: the contexts differ.

In all three cases the calculation status is COMPUTED. The answers differ because the selected definitions set different conditions. Do not merge the positive Lewis answer and the negative Brønsted answer into BOTH: they come from different contexts.

The negation in the second case needs explanation. The model holds a separate rule:

```

@source(openstax.chemistry_acids::OS_14_1_BRONSTED)
pub rule NotBronstedAcidWithoutDonation strict {
  label ru unofficial "предъявленная частица с опровергнутой донорностью прото
на – не кислота Брёнстеда";
  label en unofficial "a presented species whose proton donation is refuted is
not a Brønsted-Lowry acid";
  for s: openstax.chemistry_acids::Species;
  when openstax.chemistry_acids::species_in_question(s)
    and refuted(openstax.chemistry_acids::donates_proton(s));
  then not bronsted_acid(s);
}

```

The familiar refuted requires an established negation: `FALSE_ONLY` on the premise. Together with `species_in_question` it concludes `not bronsted_acid`. Then the selected reading's `NotAcidUnderBronsted` rule uses that refutation, together with `species_in_question`, to conclude `not is_acid`. If the case simply omits whether the particle donates a proton, this chain does not fire.

The textbook cites boron trifluoride as a Lewis-acid example. But our table checks what follows from the case's specific entries. In particular, missing information about water is a gap in this case. It does not prove the substance never interacts with water or was never studied.

Contradictory information under a single reading

Keep `BronstedReading` and consider a hypothetical particle `x`. Now the case holds two opposite entries:

```

assert openstax.chemistry_acids::donates_proton(entity_ref("urn:case:chem:x")) {
id "ab17-yes"; origin case_input; }
assert not openstax.chemistry_acids::donates_proton(entity_ref("urn:case:chem:x"
)) { id "ab17-no"; origin case_input; }

```

They concern one object in one context. Unlike the previous table, here the starting assertions diverge, not the definitions.

Check each link separately:

What was asked about	State
<code>donates_proton(x)</code>	BOTH
<code>bronsted_acid(x)</code>	NEITHER
<code>is_acid(x)</code>	NEITHER

All three queries carry `COMPUTED` status. The positive rule needs `TRUE_ONLY` on `donates_proton`; the negative one needs `FALSE_ONLY`. Under BOTH neither condition

holds. This case has no other grounds for acidity, so the conflict does not carry through to the conclusion⁴².

That is how these rules behave. It does not mean every contradiction under a reading yields NEITHER. Check the state on each claim: on our original premise it is exactly BOTH. And selecting another definition does not delete the contradictory entries from the case, though it may make them irrelevant to the new question.

Comparing contexts

BOTH concerns support and refutation of one claim in one context. Answers under different readings are kept as separate results.

When a reading is tied to a disagreement about the text

In the chemistry example, different definitions from one textbook stand side by side. In BCP 14 from Chapter 12, the situation differs: the model checks a disputed reading of the document's words.

The base rules see no special BCP 14 key-word meaning in a lowercase `must`. The `PreRfc8174Practice` reading admits one. Its rule adds a premise for that conclusion, and the package's separate priority lets it defeat the opposing rule in the base model.

With the same information about a lowercase `must` and the same dates, `keyword_used` receives `FALSE_ONLY` in the base model and `TRUE_ONLY` with the reading selected. Here `PreRfc8174Practice` carries `status disputed`. Selecting it explores that position's consequences; it does not prove that the position is historically correct.

No group here demands that reading's selection. Its absence therefore keeps the base answer rather than giving `INTERPRETATION_REQUIRED` as in the acid example. Also, `keyword_used` asks strictly about the key word's special meaning. A negative answer does not strip the sentence of all normative sense.

The name `PreRfc8174Practice` does not move the case into the past either. The date stays a date, and a separate line selects the reading.

Changing one field never substitutes for the other.

⁴²Scenario 17 states opposite information about donorship. The `BronstedAcidDonatesProton` and `NotBronstedAcidWithoutDonation` rules in `01-definitions.law` call for `TRUE_ONLY` and `FALSE_ONLY` respectively; under BOTH both chains in this case are stopped.

Try changing the question

The companion includes the acid-definitions package with its pinned chemical vocabulary. Make a working copy from reader/:

```
mkdir acid-study
cp -R corpus/laws/doctrine/chem/acid-base/. acid-study
```

Run its checks:

```
law test acid-study
```

The seventeen scenarios should pass. The command builds the package with its received dependency; Chapter 16 explains how that pinning is checked.

For each exercise, edit `acid-study/tests/chem-acid-base.lawtest` and run `law test acid-study` again. Predict the result before changing expectations, and begin each exercise with an unchanged copy of the scenario file.

1. In scenario 02 replace `BronstedReading` with `ArrheniusReading`. What information would be needed for a positive answer? Check `NEITHER`, `COMPUTED`, and the new reading; drop the old definition rule's application expectation.
2. In scenario 10 delete the negative proton-donation assertion. Why should the old `FALSE_ONLY` change to `NEITHER`? Drop the negative rule's application expectation, keeping the selected reading and `COMPUTED`.
3. In a separate copy of the same scenario name a hypothetical particle `exercise-x`, select `LewisReading`, and leave only the object's presentation and the explicit denial of electron-pair acceptance. Check `FALSE_ONLY` and `COMPUTED`. Which negative rule must now be seen in the explanation? Do not carry this exercise's condition over to boron trifluoride's real properties.

When someone else hands you an answer, ask not only about the date and the source but also about the selected definition. If the answers diverge, compare these grounds first. The next chapter adds one more refinement: the units a number is written in, and what survives conversion.

PART V

Assemble and Hand Over

CHAPTER 15

A Number With Its Unit

In Chapter 5 we compared the lengths of a triangle's sides. For that example bare numbers were enough: every length came in one shared unit. Now imagine one person reporting a distance in metres and another in centimetres. Before comparing their numbers, check whether the entries denote the same length.

The entry 50 m has two parts: the number 50 and the unit metre. 5000 cm looks different but denotes the same length. But 50 s denotes time. Equal numbers alone do not establish equal quantities; different numbers alone do not establish different quantities.

«The value of a quantity is generally expressed as the product of a number and a unit.»

BIPM · SI Brochure, 9th edition · §2.1

The edition's English rendering; consult the source for its wording.

The number refers to the chosen unit. When moving from metres to centimetres, the number changes while the length is preserved.

This chapter follows one small calculation with units. It then changes the unit used for an input, supplies an incompatible unit, and compares a rule that converts explicitly with one that does not. Precision and rounding come separately at the end.

What the calculation receives

Take constant-force work from the supplied package `calc.mechanics`. With a force of 6 newtons and a displacement of 50 metres along the force, the work is 300 joules:

$$W = F \cdot d$$

$$6\text{ N} \cdot 50\text{ m} = 300\text{ J}$$

This formula has applicability conditions: the force stays constant, and the displacement points along it. Any other direction needs the angle taken into account. Our rule receives no angle and does not check that the force is constant⁴³. Whoever supplies the inputs must justify that this calculation fits the problem.

The package vocabulary accepts these inputs:

⁴³The `WorkOfConstantForce` rule takes only force and displacement; its conditions include neither an angle nor a check that the force is constant. `Scenario 04` checks the conclusion for the given inputs.

```

pub relation given_force_and_distance(c: Calculation, f: Quantity, d: Quantity)
kind empirical {
    label en official "the constant force and the distance along which it acts"
;
    label ru unofficial "постоянная сила и путь, вдоль которого она действует";
}

```

For now we are reading the package's declarations. The next chapter supplies the build commands needed to run this example.

c denotes the calculation, *f* the force, *d* the displacement. The last two arguments have type *Quantity*: a number together with a unit. The type is general; it does not require force for *f* or length for *d*. That dimension — the kind of physical quantity — is checked by the conversion operation inside the rule.

This allows writing 5000 cm instead of 50 m. But it also means that the relation's declaration alone does not yet guarantee that what was passed into the force argument is indeed a force.

How the rule uses units

```

rule WorkOfConstantForce strict {
    label en official "W = Fd; newton times metre is the joule";
    label ru unofficial "W = Fd; ньютон на метр даёт джоуль";
    for c: Calculation;
    for f: Quantity;
    for d: Quantity;
    when given_force_and_distance(c, f, d);
    then work(c, quantity_of(unit_convert(magnitude(f), "N")
                                * unit_convert(magnitude(d), "m"), "J"));
}

```

The beginning is familiar: the rule reads the input relation and derives a new one. The computation sits in the argument of the *work* relation. Let us read it in order.

magnitude(f) supplies the numeric magnitude together with the unit information needed by the next operation.

It does not leave only a bare number.

Then *unit_convert(..., "N")* converts the force to newtons, and a second such operation converts the displacement to metres.

The resulting values are multiplied. Finally, `quantity_of(..., "J")` expresses the result in joules, checking that the unit agrees with the computed product. This is no license to stick an arbitrary label on an arbitrary number⁴⁴.

The `units.si` package defines the relationships between units. In ordinary mathematical notation the newton is written as $\text{kg}\cdot\text{m}/\text{s}^2$, and the joule as $\text{kg}\cdot\text{m}^2/\text{s}^2$. Multiplying by metres gives $(\text{kg}\cdot\text{m}/\text{s}^2) \cdot \text{m} = \text{kg}\cdot\text{m}^2/\text{s}^2$: the product of force and displacement has the dimension of work. The next chapter shows how to connect that package.

Dimension and applicability of the formula

Consistent units allow the calculation to run. They do not establish that the force is constant and acts along the displacement. Those conditions need separate justification.

The full example

Below is a scenario from the mechanics check set. This runnable copy updates only the historical test-language header to 0.2 for the npm CLI. It can be saved as a separate `.lawtest` file:

```
language "law.core" version "0.2";
package calc.mechanics version "0.1.0";
namespace "urn:law:calc:mechanics";

test "urn:query:calc-mech-t04-work" {
  given {
    context { decision_time @2026-09-06T12:00:00Z; knowledge_time @2026-09-06T12:00:00Z; legal_time @2026-09-06; timezone "UTC"; }
    assert given_force_and_distance(entity_ref("urn:case:calc:m4"), 6 N, 50 m) { id "c4a"; origin case_input; }
  }
  evaluate truth(work(entity_ref("urn:case:calc:m4"), 300 J));
  expect truth_status == TRUE_ONLY;
  expect evaluation_status == COMPUTED;
  expect applied(WorkOfConstantForce);
}
```

The case reports force and displacement. The question checks whether 300 joules of work follow. The expectations demand, in order, a positive answer with no negative support, a completed computation, and the application of `WorkOfConstantForce`.

The arithmetic is checked independently: $6 \cdot 50 = 300$. The explanation shows the rule and the input assertion `c4a` on which it fired. That lets us link the numeric answer to the particular input quantities.

⁴⁴§49.1 of the Arxo specification describes Magnitude, explicit conversion, and the check of the result's unit; the unit definitions live in the SI registry. Operations use the unit information together with the number.

Change the unit, keep the quantity

In a copy of the scenario, replace 50 m with 5000 cm. Leave the expectations alone. The rule converts centimetres to metres, and the test passes again: the work is 300 J.

The same length, the same work

Step	Quantities	What happens
Input information	6 N; 5000 cm	The force and the displacement belong to one calculation.
Explicit conversion	6 N; 50 m	$5000 \times 1/100 = 50$. The force is expressed in N, the displacement to m.
Product and result	$300 \text{ N}\cdot\text{m} = 300 \text{ J}$	After multiplication, quantity_of expresses the work in J.

The steps of WorkOfConstantForce, written mathematically. The original 6 N and 50 m also give 300 J. Both tests return TRUE_ONLY and COMPUTED.

The conversion uses two different properties of a unit:

Characteristic	What it lets us establish
Dimension	Whether one unit can be converted into another at all
Scale	How to recalculate the number in such a conversion

Metre and centimetre share the dimension of length. A centimetre is 1/100 of a metre. Seconds cannot be converted into metres: their dimensions differ.



One distance, different strides. Name the unit.

Another mechanics scenario starts with an initial velocity of 36 km_per_h, zero acceleration, and an elapsed time of 1 h. It asks for the final velocity of 10 m_per_s.

With zero acceleration the velocity stays the same; its written form changes in both number and unit: $36 \cdot 1000 / 3600 = 10$. The rule explicitly converts velocity, acceleration, and time to consistent units.

When the number is right but the unit is wrong

Replace 6 N with 6 J in the work example. The number stays the same, but energy goes in instead of force. Conversion to newtons is impossible. The query yields NEITHER with the status `RUNTIME_ERROR`; the reason is `UNIT_DIMENSION_MISMATCH`.

That result does not mean "the work is not 300 joules". The computation never completed: the input dimension does not fit. What needs fixing is the force input, not the expected numeric answer.

The OpenStax measurement model provides a contrast: a rule that compares quantities without first converting them. It checks a claimed product of quantities. Here is the rule's condition:

```
when claimed_product(e, first, second, claimed)
    and magnitude(first) * magnitude(second) == magnitude(claimed);
```

This is an excerpt from `ClaimedProductMatches`⁴⁵. Here `e` identifies the equation being checked; `first` and `second` are the factors, and `claimed` is the proposed result. The rule multiplies the first two quantities and compares the result with the third. There is no explicit conversion before the comparison here.

⁴⁵The product-check rule and scenarios 02a–c. In both comparison failures the full document reports `UNIT_MISMATCH` and `RUNTIME_ERROR`; when converting J to N the reason is different — `UNIT_DIMENSION_MISMATCH`.

The same diagnostic, different fixes

Factors	Claimed result	Check outcome
2 m_per_s and 3 s	6 m	TRUE_ONLY, COMPUTED Comparison completed The units of the product and of the claimed result coincided.
The same	6 s	NEITHER, RUNTIME_ERROR UNIT_MISMATCH Length and time are incompatible. Correct input data is needed.
The same	0.006 km	NEITHER, RUNTIME_ERROR UNIT_MISMATCH The length is the same, the scale differs. This rule has no explicit conversion.

Outcomes of the ClaimedProductMatches rule. A refused comparison is not a negative answer: the computation did not finish.

In the last two cases the reason is UNIT_MISMATCH: the comparison needs identical units. But the two errors mean different things. Metres and seconds clash in dimension. Metres and kilometres agree in dimension, but the scale needs recalculating before comparing. 0.006 km still equals 6 m physically; this particular rule just skips the needed conversion.

So a quantity can be convertible, yet a particular rule may still not convert it. Those are different things. The work rule writes unit_convert explicitly. The product-check rule leaves it out.

Precision and rounding

An exact calculation does not make the initial measurement exact.

If a length carries a measurement error, multiplying by force does not remove it. Our simple case records no measurement errors; the answer does not establish that the measurements were error-free.

For arithmetic the language distinguishes decimal numbers from exact fractions. For example, another mechanics scenario divides the work 200 J by the heat quantity 500 J and obtains the dimensionless ratio 2/5. The fraction needs no prior replacement by an approximation.

The model can also round when the problem requires it. In `.law` you write rounding explicitly. For example, the following expression rounds a decimal value to two decimal places under `HALF_UP`:

```
round(1.235, 2, "HALF_UP")
```

This is a bare expression, not a complete model file. Its result is 1.24. The arguments give the number of places and how to handle the midpoint. The rounded value is then used in the rest of the calculation.

When to round is part of the calculation

Rounding 1.235 to 1.24 first and then doubling gives 2.48. Doubling first and then rounding to two places gives 2.47. In both cases the `HALF_UP` policy is chosen.

Shortening a number for display is different. Showing fewer places alone must not change the value the rules use. An exact result, an explicitly rounded result, and its display must be kept apart.

Check the example's boundaries

The next chapter gives the build commands for this scenario and its dependency. Prepare the changes in tasks 1 and 2 now, then run them there using a copy of the full example. Task 3 can be reasoned through from the table:

1. Replace metres with centimetres, keeping the length. Explain why the `300 J` expectation stays the same.
2. Pass joules instead of newtons. Check not only `NEITHER` but also `RUNTIME_ERROR` with the reason `UNIT_DIMENSION_MISMATCH`. The old rule-application expectation must be removed: the rule could not produce a result.
3. Compare the two failures from the product table. For which of them is an explicit unit conversion enough, and for which must the input data itself be fixed?

Now the number carries a unit, the conversion lives in a rule, and the formula carries applicability conditions. For someone else to repeat this calculation, they need the same formulas and the same unit definitions. The next chapter shows how to hand over both together.

CHAPTER 16

Reusing a Package

In the previous chapter the work rule used the newton, the metre, and the joule. Their definitions lived in a separate package. The acid chapter split things the same way: one package set the chemical vocabulary, another built definitions on it. Time to work out how these parts join.

Start by identifying what the package provides: its public names, the inputs it requires, and the result it promises. A package used by another package is a dependency; the package that uses it is the consumer. We will use chemistry to examine the names, then mechanics to check a pinned dependency and repeat the familiar calculation.

The package name and the names inside it

The chemical vocabulary opens with three lines:

```
language "law.core" version "0.2";  
package openstax.chemistry_acids version "0.1.0";  
namespace "urn:openstax:clir:chemistry-acids";
```

The first line names the language and its version; the second, the package and its version. The third sets the namespace, the shared beginning from which the declarations' addresses are built.

In an explanation a rule's address may look like this: urn:openstax:clir:chemistry-acids#AmphiproticDonatesAndAccepts. In this address the namespace comes before # and the rule's name after it. A label helps us recognize the rule; the full address identifies the declaration that took part, without guessing from a similar name.

We have often met pub before declarations. It marks a name as available to other packages. A declaration without pub may take part in the work of its own package, but other packages cannot refer to it as a public name⁴⁶.

A public name does not establish that its definition is sound or fits the case at hand. pub is about availability.

⁴⁶ §§23–24 of the Arxo specification tell apart import, declaration visibility, and a dependency's participation in execution. A name's publicness does not replace connecting the needed program contents.

We still need to read the declaration's meaning and constraints.

What an import means

The acid-definitions package connects the vocabulary with the line:

```
import openstax.chemistry_acids version "0.1.0";
```

After that it may refer to the vocabulary's public declarations. For example, the general question about acids uses the type `Species` from the connected package:

```
pub relation is_acid(s: openstax.chemistry_acids::Species) kind institutional {  
  label ru unofficial "частица признаётся кислотой по выбранному прочтению";  
  label en unofficial "the species is an acid under the selected reading";  
}
```

In the name `openstax.chemistry_acids::Species`, the package stands to the left of the double colon, the declaration to the right. The consumer refers to the vocabulary's own `Species` type; it does not declare a second type with a similar name and assume the two will be treated as one.

But an import line by itself runs nothing. To read the consumer's declarations, names must be resolved against the dependency's public declarations. To compute with them, its definitions and rules must also be present in the assembled program. The `law test` command prepares both when it builds the pinned world. Recognizing names and computing with their definitions are separate steps, even when one command handles them.

Mechanics has a similar line connecting `units.si`. Here the dependency provides the unit definitions the calculation's operations use. Recognizing the symbol `n` and computing a quantity with a known dimension are two different requirements on the assembled set.

One dependency, two roles

When reading the entry

The dependency's pub declarations

↓
Import context
↓

Permitted names and types

Species belongs to the chemical vocabulary.

When running the calculation

The program and the dependency's contents

↓
law test
↓

A program with definitions and rules

The units.si definitions take part in the work calculation.

Resolving names and building the dependency world are separate steps. The package-level law test command handles both.

What the package provides—and assumes

Take the work calculation from the last chapter. Before use, it helps to write down its conditions in a few lines:

What to establish	For this calculation
Which inputs are needed	Force and displacement for one calculation
Which units are admissible	Units of force and length convertible to N and m
Which answer results	Work, written in J
What is assumed	Constant force, displacement in its direction
What the calculation does not check	Whether these conditions hold; measurement errors

This table summarizes the package's contract with its user in ordinary language; it is not part of the program. The relation's signature gives the inputs' types and order, and the conversion operations check their dimensions. Neither establishes that the force in the real experiment was constant or directed along the displacement. The person applying the model must check that.

Try writing the same summary for the chemical vocabulary: what comes in, what follows, and what remains unchecked? A `donates_proton` message counts as input information. The definitions package builds its conclusion on that message but does not investigate the substance on its own. A dependency carries these applicability conditions together with its capabilities.

Why a fingerprint is needed next to the version

An import names a version, but repeatability requires pinning the dependency's exact contents. The `law.lock` file does this job.

It records the chosen packages, their versions, and content fingerprints. The version names the intended release; `artifactHash` identifies the compiled dependency, while `sourceHash` identifies its source snapshot. Both belong to the current lock format. `resolutionHash` fingerprints the outcome of resolving the dependencies together, rather than one package alone⁴⁷.

These values follow the package format's rules. Do not replace them with an arbitrary hash of the printed text or a hand-shortened string. An explanation may show the fingerprint's start; checking needs the full fingerprint in the file.

A matching fingerprint does not establish quality

A matching fingerprint confirms that the copy has the pinned contents. It does not establish whether the rules are correct or the model suits a new question.

The recipient must receive both the pinning and the dependencies themselves. The `law.lock` file tells what to look for but does not contain the dependency's program itself. Given the dependency files, the recipient can prepare the calculation without asking the author for the right version.

When a dependency file and its recorded fingerprint disagree, first investigate the cause.

Overwriting `law.lock` to fit whatever is at hand means accepting a new set, not restoring the earlier calculation.

Run the calculation with its dependency

The companion already contains mechanics, its unit definitions, and the lock that pins their contents. First make a working copy. Start in `reader/` and choose a directory name that does not already exist:

```
mkdir mechanics-study
cp -R packs/primitives/natural-sciences/mechanics/. mechanics-study
```

⁴⁷§11: `law.lock` sets the pinned information and separates updating dependencies from execution. The `mechanics` manifest declares the `units.si` dependency.

`mechanics-study` is now your copy of the package. It contains the rule files, scenarios, `law.lock`, and `deps/`. Check that its dependency files agree with the received lock:

```
law install --project mechanics-study --offline
```

`--project` names the package. `--offline` uses the files already received. A successful check confirms that the pinned dependencies are available; it does not run the work calculation or choose a newer version.

Now run the scenarios:

```
law test mechanics-study
```

The ten scenarios should pass, including the 300-joule work calculation. `law test` reads the package's manifest, builds the program with its pinned dependencies, and compares each answer with the scenario's expectations.

A program is not yet an answer

The program needs a case, a question, and a context before it can answer. `law test` supplies these from each recorded scenario and compares the result with its expectations.

For tasks 1 and 2 from Chapter 15, edit the work scenario in `mechanics-study/tests/calc-mechanics.lawtest`, then run `law test mechanics-study` again. Begin each exercise with an unchanged copy of that file. Replacing `50 m` with `5000 cm` keeps the result. After replacing `6 N` with `6 J`, use the dimension-error expectations specified in Chapter 15, including removal of the rule-application expectation.

Check the pinning separately

Make a second copy for the deliberate error:

```
cp -R mechanics-study mechanics-bad-pin
```

In `mechanics-bad-pin/law.lock`, find the dependency record named `units.si`. Change only its `artifactHash` to `sha256:` followed by 64 zeros. The dependency files themselves stay unchanged. Try the same installation check:

```
law install --project mechanics-bad-pin --offline
```

The command refuses the altered lock. The recorded fingerprint no longer agrees with the received set. Read the refusal before changing anything else. Restore the lock from the untouched copy and check again:

```
cp mechanics-study/law.lock mechanics-bad-pin/law.lock
law install --project mechanics-bad-pin --offline
law test mechanics-bad-pin
```

The installation check should succeed and the ten scenarios should pass again. The mechanics rules needed no repair: the deliberate error was in the pinning⁴⁸.

The chemical vocabulary allows the same check. The refusal reason stays the same: the pinned fingerprint and the dependency copy in hand disagree.

Three states of one set

What was presented	Installation check	What is established
Original law.lock and dependency	Successful	The copy matches the pinning.
64 zeros were written into artifactHash	Refused	The fingerprint does not match the previous copy of the dependency.
Restored the original law.lock	Successful	Consistency restored without changing the rules.

The dependency files and rule sources stay unchanged. Restoring the original lock restores the consistent set.

What to hand over together

This is the dependency part of the handover; Chapter 18 returns to the full transfer. With the rule files, enclose the manifest `law.toml`, which names the package and declares its dependencies, the dependency pinning, the dependencies themselves, and sample questions. If the model rests on a text, the saved source materials and their bindings will be needed too.

⁴⁸This edition's experiment changes only the dependency's artifactHash in a copied lock; the dependency files stay unchanged. The npm CLI refuses the altered lock. Returning the saved lock restores installation and all ten mechanics scenarios. The historical specification link above describes the earlier contentHash format; the companion's migration note records the current sourceHash/artifactHash format.



The nest travels together with the little tree that holds it.

Equally important are brief instructions: which file to run, which question is asked, what is expected, and which conditions the user must check before calculating. The prepared set must let anyone repeat the answer without remembering how the author assembled it on their own machine.

«...any party can recreate bit-by-bit identical copies of all specified artefacts.»

Reproducible Builds · definition of a reproducible build

The edition's English rendering; consult the source for its wording.

The definition assumes identical sources, environment, and build instructions. A repeated 300 J answer by itself does not yet prove bit-by-bit reproducibility of all artefacts.

Now we can connect a package, check its pinned copy, and examine its role in the computation. The fingerprint check establishes agreement with the pinning; the scenarios check particular answers. Neither establishes that every rule is right. A wrong rule can pass tests that never expose its difference from the intended rule. The next chapter builds checks that do.

CHAPTER 17

A Check That Tells Models Apart

In Part I we recorded two pieces of information about water and derived amphiproticity. Then we removed one piece, and the conclusion disappeared. These two cases check different sides of one promise: two premises suffice, and one does not.

Now imagine an error slipping into the rule — say, the proton-acceptance condition is dropped by accident. The two-premise scenario still gives the right answer. So one lucky example is not enough: we need a case where the right and the wrong rules behave differently.

First name the promise

Let us return to the familiar rule:

```
pub rule AmphiproticDonatesAndAccepts strict {
  label ru unofficial "способная и отдать, и принять протон частица амфипротна (14.1)";
  label en unofficial "a species capable of either donating or accepting protons is amphiprotic (14.1)";
  for s: Species;
  when donates_proton(s) and accepts_proton(s);
  then amphiprotic(s);
}
```

This rule carries at least three checkable requirements: both properties are needed; they must concern one particle; to test the inference, the conclusion must be derived from the inputs through this rule, rather than supplied in the case.

For each requirement we can pick a small change to the model that breaks it:

Requirement	Error the scenario must notice
Amphiproticity follows from two pieces of information	The rule is removed
Both pieces of information are needed	One condition is dropped from the rule
The information concerns one particle	The second premise names another variable

That states in advance what each check case is for.

The expectation comes from the rule's meaning and the source.

The run then tests that expectation — the run's own output must not become the expectation's sole ground.

What the two-premise example notices

Scenario 14 of the chemical vocabulary reports both properties of water and expects `TRUE_ONLY`. If the rule `AmphiproticDonatesAndAccepts` is removed, the same information yields `NEITHER`. The scenario catches the lost conclusion.

Scenario 15 reports only proton donation and expects `NEITHER`. It passes both on the original model and on the model without the rule. The outcome matches, but the reasons differ: in one case a premise is missing; in the other, the very path to the conclusion is gone.

That does not make scenario 15 useless. It catches a different error: dropping the proton-acceptance condition. After that edit, a report of proton donation alone suffices, and `TRUE_ONLY` arrives instead of the expected `NEITHER`. One scenario may catch one error and miss another.



Both umbrellas worked in the sun. Rain revealed the difference.

For experiments we can use the `law engine mutate` tool. It proposes changes to the program, for example `remove_rule`, `drop_conjunct`, and `freshen_var_occurrence`. The

last replaces one occurrence of a variable with a fresh one. Such altered variants are usually called mutants⁴⁹.

«A mutant is considered “killed” if a test case distinguished it from the original program.»

A. Jefferson Offutt, Jie Pan · Using Constraints to Detect Equivalent Mutants · 1994, §1.2
The edition's English rendering; consult the source for its wording.

This is about a difference detected by a test. Matching answers on one input does not yet mean the programs are equivalent on all inputs.

Here they serve as deliberately wrong models to compare against the original. If a change makes the model invalid, the language check catches it. The harder case is what matters here: the model stays valid, but the reasoning changes.

Keep the expectation

Keep the original expectation when testing a mutant. Replacing NEITHER with TRUE_ONLY to match the erroneous rule stops the test from checking the original promise.

An error the earlier examples missed

Imagine that instead of one variable in two premises, the rule requires proton donation from a particle *s* and acceptance from some particle *t*. That is already a different statement: the properties may belong to different objects.

In the original water example the error stays invisible. Water both donates and accepts a proton, so either variable can bind to it. What is needed is a case that splits the properties.

Below is the author's check based on scenario 15. One piece of information about another object is added. The header and context are kept so the example can be run as a separate `.lawtest`:

⁴⁹The CLI's `mutation notes` applies local changes to the CLIR program and checks that the result is valid. The original chemical set holds 22 scenarios. The count of passed cases concerns a particular mutation, not the model's completeness.

```

language "law.core" version "0.2";
package openstax.chemistry_acids version "0.1.0";
namespace "urn:openstax:clir:chemistry-acids";

test "urn:query:openstax-chem-acids-15-book-water-one-role-no-conclusion" {
  given {
    context {
      decision_time @2026-09-06T12:00:00Z;
      knowledge_time @2026-09-06T12:00:00Z;
      legal_time @2026-09-06;
      timezone "UTC";
    }
    assert donates_proton(entity_ref("urn:case:chem:h2o")) { id "k15-1"; ori
gin case_input; }
    assert accepts_proton(entity_ref("urn:case:book:other")) { id "book-othe
r-accepts"; origin case_input; }
  }
  evaluate truth(amphiprotic(entity_ref("urn:case:chem:h2o")));
  expect truth_status == NEITHER;
  expect evaluation_status == COMPUTED;
}

```

Donating belongs to h2o, accepting to other. We ask about h2o's amphiproticity. On the original model the answer is NEITHER: information about another object does not satisfy the premise about water.

On the altered model, where the acceptance premise carries a substituted variable, the answer becomes TRUE_ONLY and the test fails. That is exactly the difference we wanted to catch.

All 22 earlier scenarios still pass on that altered rule. The new case exposes a specific coverage gap: none of them splits the properties across objects in a way that fits this error. A count of passing checks alone does not reveal that gap.

The mutant passed — what does that mean?

A passing mutant may reveal a gap in the tests, or it may behave like the original program. Here, splitting the properties between two objects gives a concrete case in which the answers differ.

For the symmetric error a mirror check is needed: water accepts a proton, and another object donates a proton. Then substituting the variable in the donation premise becomes visible too.

Which check notices the error

Scenario	Rule removed	Acceptance condition dropped	Acceptance variable substituted	Donation variable substituted
14: both properties in water TRUE_ONLY	FAIL	PASS	PASS	PASS
15: only donation in water NEITHER	PASS	FAIL	PASS	PASS
Water donates, other accepts NEITHER	PASS	FAIL	FAIL	PASS
Water accepts, other donates NEITHER	PASS	PASS	PASS	FAIL

On the original model, all four tests pass. FAIL here means: the test detected the error in the altered model. PASS means that case did not tell it apart. Under the scenario name, the expected truth_status is given.

The same answer may come another way

Scenario 14 has one more expectation:

```
expect applied(AmphiproticDonatesAndAccepts);
```

It checks that the named rule took part. To see its use, take a temporary copy of the model. Give the original rule an extra condition that the case leaves unknown, and add a second rule deriving amphiproticity from the same two premises. Before running the changed model, predict the truth status and the rule-application check separately.

The water question still yields TRUE_ONLY, so the answer-only check passes. But the applied(AmphiproticDonatesAndAccepts) expectation fails: the conclusion came through a different rule. The explanation exposes the substitution.

At the same time, applied does not prove the named rule was the answer's only ground, nor that its premises faithfully express the source. To assess that, we must read the rule itself and the matching branch of the explanation. The expectation has one particular job⁵⁰; do not stretch its meaning further.

The same holds when no conclusion is established. On NEITHER, check what was missing: information, a reading choice as in Chapter 14, an applicable edition as in Chapter 13, or a rule. A matching status alone does not mean matching reasons.

⁵⁰Scenario 14 separately checks truth_status and applied. The original rule links both premises with one variable; the alternative rule in the experiment is a study change to a temporary copy.

Check the intermediate steps

The model on Gauss's *Disquisitiones Arithmeticae* checks a supplied trace — a step-by-step record of a modular exponentiation calculation: raising a number to a power and finding its remainder after division by a fixed number. To follow the example, it is enough to know that a remainder is what is left after division.

The case reports base 2, divisor (or modulus) 341, and several "exponent — remainder" pairs. The model checks the links between them: when the exponent doubles, the remainder must match the square of the previous remainder taken modulo the same modulus.

In scenario 10, exponent 5 is recorded with remainder 33, and exponent 10 with remainder 1. Let us check the pair independently:

$$\begin{aligned} 33^2 &= 1089 \\ 1089 &= 3 \cdot 341 + 66 \end{aligned}$$

The independent calculation gives remainder 66, not 1.

(Author's paraphrase of the reported engine result; the full output is not reproduced here.) The rule `VestigiumFalsumInDuplicatione` detects the inconsistency and derives `vestigium_falsum`. The query yields `TRUE_ONLY`, `COMPUTED`, and the explanation points to these two recorded steps.

The model also checks adjacent exponents: when the exponent increases by one, multiply the previous remainder by the base and take the remainder modulo the same divisor. This explains the second error, from remainder 16 at exponent 4 to remainder 33 at exponent 5: $16 \cdot 2 = 32$, already below 341, so the next remainder must be 32. The rule `VestigiumFalsumInGradu` checks that link.

Scenario 12 removes a link needed to build the checked chain. Here it is not enough to check equalities for the pairs at hand: we must also establish that the chain reached the declared final exponent 340.

(Author's paraphrase of the reported engine result; the full output is not reproduced here.) A separate rule does this job. It reads the declared limit and `not_known(catena(...))`: no checked chain reaching it is established. The result is `vestigium_ruptum`: the trace breaks. Here 340 is the exponent the checked chain must reach.

That is how the check tells apart a wrong link and a missing path to the goal. Neither result on its own proves Fermat's little theorem or that 341 is prime. The check's subject is narrower: the consistency of the presented computation trace⁵¹.

⁵¹The trace-check rules, scenario 10, and scenario 12. The first rule compares the presented remainders, the second checks the absence of an established catena up to the declared limit.

A wrong link and a path that did not reach the goal

Scenario 10: the wrong link

Exponent 5: remainder 33
Exponent 10: remainder 1

$$33^2 \bmod 341 = 66 \neq 1$$

VestigiumFalsumInDuplicatione

vestigium_falsum

In both scenarios: TRUE_ONLY, COMPUTED.

These are two separate questions about the presented trace. Detecting a wrong step and the absence of a path to the goal are not mutually exclusive properties.

Scenario 12: the broken chain

Declared final exponent: 340
No checked chain reaching it established

not_known(catena(..., 340))

VestigiumRuptum

vestigium_ruptum

What repeating a calculation checks

So far we asked whether the model behaves as its stated rules require. One question remains: did the recipient repeat the calculation they were handed?

The calculation document holds the input manifest, the answers, and the explanation graph. The `resultHash` field serves as the fingerprint of its canonical contents. Comparing fingerprints catches a difference between documents even when both show `TRUE_ONLY`.

But a matching fingerprint does not check whether the author read the source rightly. It establishes which particular calculation was reproduced. Substantive checks and repetition checks solve different problems; the last chapter runs both.

Try picking the missing check

Work with a copy of the chemical package. First predict the outcome of scenarios 14 and 15 if `accepts_proton(s)` is removed from `AmphiproticDonatesAndAccepts`. Then make that edit and prepare the program with the `law engine lower` command and run the unchanged tests with `law engine test --program`. Scenario 14 must pass, and 15 must catch the error.

Then take the two-object check above and swap the relations: water accepts, another object donates. Confirm the explanation above by checking which of the two variable substitutions this case catches.

Finally, for one rule of your own, write down its promise and two examples: where it must produce a conclusion and where a nearby case must not yield that conclusion. If both examples pass after removing an essential condition, the check needs refining.

The final scenario set does not prove the model complete.

It shows which differences we can detect. Hand over that knowledge with the model too: what was checked, by which case, and which boundary stays unchecked so far.

CHAPTER 18

A Model You Can Hand Over

Imagine sending the model to a colleague. They open the files, but no one stands beside them who remembers where the dependency lives, which case was chosen, and why this particular number or statement is expected in the answer.

A well-prepared model lets them recover all of that from the set itself. The recipient repeats the question, checks the result, then changes one ground and explains why the answer changed or stayed the same.



If you have to send the author along, the set is not ready yet.

We assemble such a set using the water example from the start of the book. It needs no new rules. We separate the familiar input information, the question, and the test's expectations.

Keeping the case separate from the test

In `.lawtest`, `given`, `evaluate`, and `expect` stand side by side: inputs, question, expectation. For handing over, keep the information in a separate `.lawcase` file. Different questions can then be asked about the same case without building an expected answer into it.

The book's example is called `kniga.voda`. Its manifest, `law.toml`, identifies the package and its dependencies:

```
[package]
name = "kniga.voda"
version = "0.1.0"
language = "0.2"
namespace = "urn:kniga:clir:voda"

[dependencies]
"openstax.chemistry_acids" = "0.1.0"

[[cases]]
name = "Voda"
path = "cases/voda.lawcase"
```

The [dependencies] section names the chemical vocabulary. The [[cases]] section links the name Voda to the case file. That is a pointer to the information the query will run on.

At the root lies package.law with the header and the import:

```
language "law.core" version "0.2";
package kniga.voda version "0.1.0";
namespace "urn:kniga:clir:voda";

import openstax.chemistry_acids version "0.1.0";
```

And the full cases/voda.lawcase file looks like this:


```

language "law.core" version "0.2";
package kniga.voda version "0.1.0";
namespace "urn:kniga:clir:voda";

import openstax.chemistry_acids version "0.1.0";

case Voda {
  context {
    legal_time @2026-09-08;
    knowledge_time @2026-09-08T12:00:00Z;
    decision_time @2026-09-08T12:00:00Z;
    timezone "UTC";
  }
  assert openstax.chemistry_acids::donates_proton(entity_ref("urn:case:kniga:h
20")) {
    id "voda-1";
    origin case_input;
  }
  assert openstax.chemistry_acids::accepts_proton(entity_ref("urn:case:kniga:h
20")) {
    id "voda-2";
    origin case_input;
  }
}

```

Both assertions concern one object. The relation names are qualified because the case belongs to one package and the vocabulary to another. The context's dates and each assertion's origin are stated explicitly.

Neither assertion states that water is amphiprotic. A rule from the imported vocabulary package must derive it. The case also holds no `expect:` expectations belong to the check, not to the inputs.⁵²

Prepare a copy to hand over

Install npm CLI 0.2.4 as described in the prologue. The companion provides a complete Water package with its case, compiled dependency, source snapshot, and lock. Start in `reader/` and make a working copy:

```

mkdir water-study
cp -R docs/books/postroeniya-alt/examples/voda/. water-study
law install --project water-study --offline

```

The installation check should confirm that the received dependency files match `law.lock`. Keep this lock: it records the author's chosen set. Generating another lock

⁵²A standalone case and the `ask` operation are described in §168.4 of the specification. The `Voda` declaration and both original assertions are given in the example file.

would choose a new set rather than restore this one. The companion's migration note records how the author prepared the package.

The small "Water" package does not define amphiproticity itself; the definition comes from the chemical vocabulary. Its dependency source snapshot includes the readable rules and saved textbook fragments. Keep those materials with the compiled dependency. The program repeats the calculation; the source text lets the recipient assess the formalization.

«The research itself is the complete software development environment and the full set of instructions used to obtain the figures.»

Jonathan B. Buckheit, David L. Donoho · WaveLab and Reproducible Research · 1995, §3.1, p. 5

The edition's English rendering; consult the source for its wording.

This is how the authors phrase John Claerbout's idea. For our model, the handed-over result is likewise accompanied by the means of obtaining it; reproducibility on its own does not justify the rules.

Ask the question

```
law ask water-study --case Voda \
  --query 'evaluate truth(
    openstax.chemistry_acids::amphiprotic(
      entity_ref("urn:case:kniga:h2o")
    )
  );' \
  --query-id amphi --out water-answer
```

--case selects the case. --query carries the familiar `evaluate truth(...)` question. --query-id gives the question a stable name; --out names the output directory.

The saved evaluation includes these three files used in the chapter:

File	What it holds
<code>ask.json</code>	Which package, case, and question were chosen
<code>request.json</code>	The prepared computation input
<code>result.json</code>	Answers, explanation, and the calculation manifest

The CLI also writes `bundle.json` and `index.json` for the saved evaluation's package information. Keep the whole output directory when handing over a replay.

In `result.json` the answer to the amphiproticity question is `TRUE_ONLY`, the status `COMPUTED`. The explanation graph links the conclusion to the rule `AmphiproticDonatesAndAccepts` from the vocabulary and assertions `voda-1`, `voda-2`

from the case. That is how we see the dependency's rule truly took part in the calculation.⁵³

What the recipient checks

The document has several fingerprints:

Field	What it helps compare
<code>manifest.programHash</code>	The program used in the calculation
<code>manifest.caseHash</code>	The prepared case
<code>resultHash</code>	The canonical result document

For the full water handover, collect:

- the prepared package directory, with `law.toml`, `package.law`, the case, `law.lock`, and the compiled dependency in `deps/`;
- the readable dependency sources and saved textbook fragment;
- installation instructions for Node.js 20 or newer and `@arxo/cli@0.2.4`, together with the supported platform and `law version --json` output;
- the question command, its expected answer, and the original `result.json` for comparison;
- the complete saved evaluation directory if the recipient will use `law eval`.

The recipient first installs the CLI with `npm install -g @arxo/cli@0.2.4` and confirms the version. They receive the package files separately from npm. `law install --offline` checks the existing dependency pins and restores from local bytes without choosing new versions; it leaves the author's lock in place.

The local repetition below uses that same recipient route. `water-recipient` is a second directory, containing the package sent to the recipient:

⁵³The conclusion links assertions `voda-1`, `voda-2` to the rule `AmphiproticDonatesAndAccepts` from the chemical package. The rule's application is checked against the explanation graph in `result.json`, not only against the final `TRUE_ONLY`.

```
cp -R water-study water-recipient
law install --project water-recipient --offline
law ask water-recipient --case Voda \
  --query 'evaluate truth(
    openstax.chemistry_acids::amphiprotic(
      entity_ref("urn:case:kniga:h2o")
    )
  );' \
  --query-id amphi --out recipient-answer
cmp water-answer/result.json \
  recipient-answer/result.json
```

The last command compares the two documents byte for byte and stays silent when they match. On the checked set they match. This is only a local rehearsal of handing over — not a run on someone else's computer — but it shows the calculation can be repeated with the npm-installed CLI and the received dependency pins in a different directory.

The environment needed to repeat the calculation

State the tool version and the repository files used. Moving the directory on the same computer checks only part of the handover: the recipient also needs a compatible environment in which to run it.

To replay the saved evaluation directly, run:

```
law eval water-answer
law inspect water-answer/result.json summary
```

`law eval` repeats the saved request and checks its result. `law inspect` summarizes the document without making another claim about the source text.

Hashes cover contents under canonical-form rules — a standard representation for comparison — rather than the JSON's outward formatting. A matching `resultHash` confirms the pinned document was reproduced. It does not establish that the inputs are true in the world or prove that the textbook was interpreted correctly.⁵⁴

Two ways to compare contents

`cmp` compares complete files byte for byte. Changing JSON indentation can change the bytes while preserving the document's canonical contents. Adding an assertion changes the case itself.

⁵⁴The manifest's makeup is described in §170, the canonical-form rules in §208. To explain a divergence, the document's contents are compared; one fingerprint does not tell which particular field changed.

If the fingerprints differ, first compare the program and the case. Then check the question, its name, the context, and the manifest's remaining fields. The two fields `programHash` and `caseHash` do not exhaust every possible reason for a differing result.

Change one ground

In the recipient's copy, delete the `accepts_proton` assertion block with its `id` and `origin`. Keep the first assertion and the whole context. Repeating the question yields `NEITHER`, `COMPUTED`: donation alone does not support amphiproticity. It does not refute it either: one of the rule's needed premises is missing.

Now restore the original case and add one more assertion:

```
assert openstax.chemistry_acids::donates_proton(entity_ref("urn:case:kniga:other")) {
  id "voda-other";
  origin case_input;
}
```

It reports another object's property. The answer about water stays `TRUE_ONLY`. But the case has changed, so the calculation document differs from the original. The two experiments are easy to compare side by side:

The answer can stay the same while the document changes

Change	Answer	programHash	caseHash and resultHash
Repetition without changes	TRUE_ONLY	As before	As before
Water's proton acceptance removed	NEITHER	As before	Changed
The above entry about another object added	TRUE_ONLY	As before	Changed

Comparison against the original calculation. The same answer to one question does not mean the same information was handed over.

In all rows the status is `COMPUTED`. The extra assertion did not support the conclusion about water, but it belongs to the supplied case.

Getting the same logical answer is therefore different from reproducing the same calculation document.

Not every file edit changes the calculation. Reindenting is not adding an assertion. Watch for substantive changes to the prepared case; do not expect a new hash after every keystroke.

Now take your own text

Return to the material chosen in the prologue. Narrow the first question so it can be checked on a small number of cases. There is no need to model the whole document or to use every construction in the book.

First name the source and the fragment on which the model rests. If there are editions, name the needed one. Then spell out the objects, the admissible inputs, and the rule leading to the answer. Mark which conditions stay assumptions and which the model establishes.

Prepare a positive case and a nearby case just beyond the applicability boundary. If the answer depends on a date, a reading, or a human judgment, the context and the explanation must show it. Justify expectations before running.

After that, assemble the set: the source, the readable model, cases, questions, checks, pinned dependencies, and repetition instructions. Enclose the original result document and name one change the recipient can make on their own.

What must reach the recipient

Program and dependencies

```
law.toml · package.law
law.lock · deps/
```

Names, imports, pinnings, and the dependencies' own copies.

Grounds and checks

```
Source · readable rules
Positive and boundary cases
```

Why the model is written this way and what justifies the expectations.

These are the materials to hand over, not a required directory tree. The recipient first repeats the original calculation, then makes a meaningful change.

Case and question

```
cases/voda.lawcase
Voda · amphi · evaluate truth(...)
```

The information, the explicit context, and the exact query. Expectations are kept separately.

Instructions and a sample result

```
Commands · run environment
Original result.json
```

How to repeat the calculation, what to compare it against, and which ground to change.

The set is not yet ready to hand over if any of the following hold:

- the finished outcome is asserted in the case instead of derived;
- the stated fragment does not support the rule;
- missing information is treated as negation with no declared ground;
- the expectation is merely copied from the result;
- repeating the run needs files only the author has.

Different formalizations of one text are allowed. They can be compared on their merits when each shows its choices, grounds, and consequences.

Handing over a model lets another person make that comparison without you. The epilogue gathers the grounds the recipient should be able to ask you to show.

EPILOGUE

Now Show the Grounds

A text lies on the table again. Beside it, a model that we can read, question, and change. In the prologue we did not yet know which differences between them would matter. Now we can trace those differences in the model.

It all started with water, which can give a proton to another particle and accept one. The textbook calls particles that play both roles amphiprotic. We identified the subject, recorded two pieces of information, and watched a rule link them to the conclusion.

Then one piece of information went missing. The conclusion disappeared — but that did not establish the opposite. Conflicting messages, exceptions, dates, sources, and readings entered the picture. The question "what does the answer rest on?" grew sharper.



The height was obvious. Now we can see what supports it.

Now it may sound like this: Who reported this information? Does it concern the same object? Which fragment supports the rule? Does the rule apply as of the chosen date? Which definition are we using? What happens if one of the premises is removed?

These questions now have places in the model: assertion, rule, source reference, context field, explanation branch. Not every path settles the matter. But each gives us a place to start checking a claim or discussing a disagreement.

What remains our responsibility

The model does not relieve us of responsibility for the interpretation. Someone picked a condition out of the text, chose a vocabulary, and judged the information sufficient for the conclusion. Someone judged the formula applicable to the experiment, or accepted a message as a trustworthy ground.

Some of these decisions can be written down explicitly. Others hide in relation names, in the choice of subject, in what the model never asks. So working with grounds does not end when a file is accepted or every test passes.

In the amphiproticity model, we had to explain why the two abilities are checked jointly. In the work calculation, we had to name the conditions the formula needs. In interpreting a document, we had to distinguish a chosen reading from an established claim about the text's history.

An explicit model puts these decisions on the table. Anyone can point at a line, propose another, and compare the consequences. But noticing a missing assumption, checking the source, and judging the applicability limits remains human work.

«The first principle is that you must not fool yourself — and you are the easiest person to fool.»

Richard P. Feynman · Cargo Cult Science · Caltech, 1974

The edition's English rendering; consult the source for its wording.

For the model's author, this is a reason to look also for information that could refute the chosen reading. A successful run does not free one from such checking.

A ground to return to

Talking about the grounds of knowledge offers no promise of an infallible answer. It starts with a readiness to tell apart what we observed, what we accepted, what we derived, and what we have not yet sorted out.

The model helps keep these differences when it is handed over. The recipient sees not only the outcome but the path to it: information, choices, rules, constraints. They can repeat the calculation, dispute a premise, and check what changes after replacing it.

Repeatability does not make a statement true.

It gives further discussion a shared basis: we make sure we are discussing the same model and the same calculation. Then disagreement can narrow — to the source, an assumption, a rule, or the question — instead of dissolving into two incomparable answers.

When you are handed a model, ask to be shown its grounds. When you hand over your own, show its grounds first.

A handover card for the model

Enclose a short card with the set you hand over. It helps the recipient find the essentials before reading every file.

Eight points of reference for the recipient

Field	What to state
Question	What is checked, for which object, and as of which date
Source	Document, edition, saved fragment and its fingerprint; coverage boundaries
Information	Where the inputs are taken from and what the recipient must check themselves
Assumptions	The chosen reading, applicability conditions, explicit constraints
Dependencies	Packages, versions, pinnings, and where their copies are
Repetition	Command, case, and question name; expected answer and status
Document	Where the original result.json with the full resultHash lies
Change	Which ground to remove or replace and what to expect after that

The card points the way to the materials. The full files and fingerprints travel with it.

For the "Water" example the question is about the amphiproticity of `urn:case:kniga:h2o`. The inputs are the supplied assertions about donating and accepting a proton. The expected answer is `TRUE_ONLY, COMPUTED`; after removing the assertion about acceptance — `NEITHER, COMPUTED`. The rule and the source come from the pinned chemical package; the previous chapter gives the commands and shows how to produce the result document.⁵⁵

⁵⁵The full handing-over experiment is given in chapter 18. The original "Water" case holds two assertions; removing the `voda-2` block gives the stated `NEITHER, COMPUTED` answer. This checks the given example.

Give your text a card of its own, perhaps quite short. What matters is that the person across the table can open the set, find the needed files, and continue the reasoning independently.

Colophon

Constructions. The Craft of Formalization

First English edition · Edition 2026.10 · Published 14 October 2026

© 2026 Rifat Jumagulov

Published by Arxo

The original author text is offered under Creative Commons Attribution 4.0 International (CC BY 4.0), <https://creativecommons.org/licenses/by/4.0/>. This licence does not relicense quoted passages, source documents, code from other projects, trademarks or other third-party material. Their own terms continue to apply. In particular, OpenStax Chemistry 2e is licensed under CC BY-NC-SA 4.0; its excerpts are identified at their point of use. The OpenStax-based material is offered under CC BY-NC-SA 4.0 for noncommercial use, with adaptations identified. The Scrum Guide rendering retains CC BY-SA 4.0. The companion's THIRD-PARTY-NOTICES.md records the remaining component terms and original source addresses.

German quotations from Kafka are accompanied by the edition's own English glosses. The English descriptions of Euclid's propositions are the edition's own paraphrases. Russian and Greek labels in quoted code are retained; English explanations of engine output are marked as the author's paraphrases.

The selected cover scene shows glass and paper. Its English typography is composed separately. In-text illustrations are generated artwork, selected for the book; rights to the Arxo name or logo are not granted by this licence.

The companion recompiles selected historical models with npm-installed `@arxo/cli` 0.2.4 and records its adaptations. All 383 selected scenarios and 50 printed shell blocks passed on macOS ARM64. SI test expectations were preserved. Water reproduces its result and proof hashes; four support states and four receipt scenes also passed. Current locks include source and artifact hashes. Equivalence to the retired semantic release and execution on other platforms are not claimed.

The companion excludes Syzdyq's handbook and its long embedded quotations. The runnable model keeps a bibliographic reference with `ABSTRACT_ONLY` source status, preserving its rules and expectations. Source-text coverage is not claimed for this model. The archive supplies no CLI executable, product implementation source or system font files. Editorial and source-language checks were completed with Codex and Astra. A human reader pilot has not been performed. Console excerpts preserve recorded wording; the public CLI may format its output differently.

The edition address is <https://books.arxo.io/constructions/>.