portage-ng

A declarative reasoning engine for large scale software configuration management

Pieter Van den Abeele

pvdabeel@mac.com

September 2026

Contents

Chapter 1

Introduction

1.1 The growing complexity of software

portage-ng is not a package manager. It is a **reasoning engine** for software configuration management at scale.

To understand why this distinction matters, consider operating systems as examples of complex software systems. An operating system is assembled from thousands of interdependent components — libraries, compilers, language runtimes, desktop environments, system services — each of which evolves independently. The challenge of keeping all those components working together has grown dramatically over the past three decades, and the tools we use to manage that challenge have had to evolve with it.

1.1.1 Binary software distribution

In the earliest model — still the dominant one for distributions like Debian, Red Hat, and Ubuntu — packages arrive as pre-built archives. The distribution maintainers compile each package, test it against a fixed set of other packages, and ship the result. The package manager's job is essentially logistics: download the right set of archives and unpack them into the filesystem. Order barely matters; as long as every required archive is present at the end, inter-package linkage is correct and the system works.

This model is simple and reliable. Because everyone receives the same binaries, configurations are easy to reproduce, and commercial software vendors can build and certify against a known, fixed set of packages. The trade-off is that the user has little control: configuration choices are made upstream, and customisation is limited to what the distribution maintainers decided to provide.

1.1.2 Source-based systems

Before Gentoo, installing software from source on Linux was a manual undertaking. Projects like **Linux From Scratch** (started 1999) documented the process step by step, but every command was the user's responsibility: download, patch, configure, compile, install — by hand, for every package.

The **FreeBSD Ports** system (Jordan Hubbard, 1994) was the first to automate source-based package management. **Gentoo** (originally Enoch, created by Daniel Robbins in 1999, inspired by FreeBSD Ports) brought this idea to Linux. Gentoo 1.0 (March 2002) was the first Linux

distribution to provide fully automated source-based package management as its primary mode of operation.

In its early days, Gentoo targeted a single platform (IA-32) and the dependency problem was tractable: the package manager walked the dependency graph, figured out which packages needed compiling, ordered them, and built them one by one. Compiling from source meant binaries were optimised for the exact hardware — no lowest-common-denominator builds — and the performance advantage was real.

Tools like **Portage** navigated these dependency graphs with an imperative, trial-and-error approach: try a combination, detect conflicts, adjust, and try again. For a single-platform distribution with a moderately sized tree, this approach is adequate.

1.1.3 From packages to knowledge

As Gentoo grew — thousands of packages, a dozen architectures, multiple operating systems via **Gentoo Prefix** — simple graph traversal was no longer sufficient. Each package now has **build-time options** (USE flags) that interact multiplicatively: different CPUs to target, different compilers to use, different versions of those compilers, different optional feature combinations. The space of valid configurations is not merely large — it is combinatorially vast, and every point in that space must be internally consistent.

This is the **metadistribution** concept. Gentoo does not distribute binaries; it distributes **knowledge** — recipes (ebuilds) that describe how to build every component of a system, and configuration parameters (USE flags, keywords, profiles) that let the user tailor the result. The word "package manager" does not do justice to what this entails. Putting pre-built archives together is logistics. Ensuring that thousands of packages, across multiple platforms, with user-specified feature selections and hardware constraints, form an internally consistent system — that is **configuration management**, and it requires more than a graph traversal algorithm.

Yesterday the system worked; today, after a routine update, it does not — and the answer is buried somewhere in the interaction of thousands of constraints across hundreds of packages. A trial-and-error search loop can tell you it *failed*, but not *why*.

1.1.4 From searching to proving

Solving this problem requires more than a better search loop. It requires **reasoning** — the ability to derive consequences from rules, to detect why a configuration is inconsistent, and to explain what must change to make it consistent again.

portage-ng approaches the problem this way. Instead of *searching* for a plan, it *proves* one. Every build plan portage-ng produces is a formal proof — a Prolog term that records, for every package, which rule justified its inclusion and under what constraints. When no fully valid plan exists, portage-ng does not give up: it makes explicit assumptions (flag changes, keyword acceptance, unmasking), proves a plan under those assumptions, and presents the assumptions as actionable suggestions. The proof answers not only "what should I install?" but also "why does this work?" — and when something breaks, "what changed?"

1.1.5 Declarative reasoning

This is inherently a task for **declarative reasoning**. We do not want to prescribe a fixed sequence of imperative steps; we want to state the rules of the domain and let a reasoning engine derive the consequences.

Prolog — an **artificial intelligence** language built on exactly this paradigm — is a natural fit. You specify *what* solution to produce, not *how* to produce it. The runtime — unification, backtracking, and proof search — figures out the "how." Consider a trivial example:

```
os(linux).
os(darwin).
```

```
?- os(Choice).
Choice = linux ;
Choice = darwin.
```

Prolog **automatically** enumerates every valid binding for `Choice`. This built-in **backtracking** — the ability to systematically explore all alternatives — is exactly what a configuration engine needs. Traditional Portage did not originally have backtracking at all; the retry mechanism it acquired later is not designed to enumerate alternatives but to iteratively refine a single solution by accumulating masks across restarts. In Prolog, backtracking over alternatives is not a bolt-on feature — it is a primitive of the language.

The reasoning engine portage-ng implements is not inherently tied to operating system management. We have chosen Gentoo because it captures many of the hardest sub-problems — figuring out the correct USE flag combinations to satisfy user, system, and hardware constraints simultaneously; resolving cyclic dependencies; managing co-installable slots — and any solution that works for Gentoo generalises to simpler domains. But the same proof-based architecture could reason about any domain where entities have capabilities, constraints, and dependencies — from cloud service composition to event-driven automation to hardware design space exploration. Gentoo is the proving ground; the ideas are general.

While source-based configuration management systems like Portage are used by thousands of developers and organisations, the formal concepts behind them — constraint propagation, domain narrowing, proof construction — are understood by only a handful of people. portage-ng aims to push forward the state of the art in this area and, by expressing the resolver as a set of logical rules rather than an opaque imperative algorithm, to make its inner workings more accessible and easier to reason about.

This chapter explains why Gentoo is the right domain, why Prolog is the right language, and how portage-ng's proof-based architecture addresses the problem at a level that imperative package managers cannot reach.

1.2 Why Gentoo?

The previous section closed with a claim: Gentoo captures the hardest sub-problems in software configuration management, so anything that works for Gentoo generalises to simpler domains. This section substantiates that claim — looking at what the knowledge Gentoo dis-

tributes actually contains, how that model has been stressed across architectures and platforms, and where it is deployed at industrial scale — before returning to what all of this means for a reasoning engine.

1.2.1 The metadistribution in practice

The metadistribution concept was introduced in From packages to knowledge: Gentoo distributes the declarative specification of a build process — ebuilds plus configuration parameters — rather than the output of one. What does that specification look like up close? A single Portage tree contains roughly 32,000 ebuilds, and each ebuild declares:

- **Dependencies** — what it needs at build time, run time, and post-install
- **Use flags** — optional features the user can enable or disable
- **Slots** — multiple versions that can coexist
- **Keywords** — which architectures the package is tested on
- **Use constraints** — restricting valid Use flag combinations

None of these declarations stands alone. Enabling a USE flag on one package activates extra dependencies; those dependencies carry their own flags, slots, and keyword restrictions; a slot chosen deep in the graph can invalidate a version chosen near the top. This is not a bug — it is the point. Gentoo's power comes from this configurability. But it also means that reasoning about Gentoo packages is reasoning about a large, richly structured constraint space — not merely walking a dependency graph.

1.2.2 Architectures and keywords

Gentoo was originally built for the IA-32 (x86) architecture. As contributors ported it to other platforms — PowerPC, ARM, SPARC, MIPS, HPPA, and others, often available in 32-bit and 64-bit variants — the project developed the **keyword** system to track per-architecture stability. An ebuild can be marked `amd64` (stable on x86-64), `~arm` (testing on ARM), or carry no keyword for a given architecture (meaning it has not been validated there at all). Keywords turn architecture support into a first-class constraint in the dependency graph: a package that is stable on one architecture may be unstable or unavailable on another, and the resolver must respect those boundaries.

Platforms beyond x86 Linux — such as BSD, Solaris, and others — were handled as regular Gentoo targets with a different kernel and different user-space libraries, using the same Portage machinery and ebuild format. Google's **ChromeOS** is a prominent example of such a different platform delivered and managed entirely by Portage: ChromiumOS maintains a fork of Portage alongside Gentoo-derived overlay repositories (`portage-stable` for unmodified upstream ebuilds, `chromiumos-overlay` for Google-specific packages), and changes flow back to upstream Gentoo regularly.

The **Gentoo Prefix** project (an outgrowth of the Gentoo for Mac OS X effort) addressed a different challenge: installation *within* a pre-built operating system where root binaries cannot be modified. On platforms like Mac OS X, Prefix installs Portage and all packages into a user-defined offset directory rather than the filesystem root, allowing a full Gentoo-managed software stack to coexist with the host system.

1.2.3 Real-world adoption

Gentoo's approach to source-based configuration management has been adopted well beyond the Gentoo community:

- **ChromiumOS / ChromeOS** (Google). ChromiumOS is the open-source project; ChromeOS is Google's proprietary product shipped on Chromebooks. Both are built using Gentoo's Portage, with overlay repositories (`portage-stable` for unmodified upstream ebuilds, `chromiumos-overlay` for Google-specific packages). In 2025, Google confirmed that ChromeOS and Android are merging into a unified platform (codenamed "Aluminium") for 2026, with Android's kernel as the foundation and ChromeOS's desktop interface layered on top.
- **Container Linux** (CoreOS, later Flatcar). CoreOS Container Linux — a lightweight, container-optimized operating system designed for cloud infrastructure — was built on Gentoo foundations, using Portage and ebuilds for its build system. After CoreOS was discontinued in 2020, **Flatcar Container Linux** continued the Gentoo-based lineage and is deployed at scale by organisations including Adobe (18,000+ nodes), Equinix, and numerous managed Kubernetes providers.

These adoptions are not cosmetic. ChromeOS and Flatcar use the same ebuild format, the same Portage dependency resolver, and the same overlay architecture as upstream Gentoo. The fact that this machinery scales from a single developer's workstation to tens of thousands of production nodes is evidence that Gentoo represents state-of-the-art practice in large-scale software configuration management.

1.2.4 Reasoning about software at scale

When you ask "can I install Firefox with Wayland support on this machine?", you are really asking: "does there exist a consistent assignment of package versions, USE flags, and slot choices across my entire dependency graph such that all constraints are satisfied?" That is a **satisfiability problem** over a structured domain.

portage-ng treats the Portage tree as what it truly is: a **declarative knowledge base**. Ebuilds are not build scripts to execute — they are propositions with preconditions. Dependencies are not edges in a graph to traverse — they are logical implications to prove. Configuration choices are not switches to flip — they are constraints to satisfy.

This shift in perspective — from "searching for a working set of packages" to "proving that a consistent configuration exists" — is what makes portage-ng fundamentally different from Portage, Paludis, and pkgcore — the three existing package managers that operate on the same ebuild base.

1.3 A Prolog primer

If you have never used Prolog, this section gives you enough to follow the rest of the book. If you already know Prolog, skip to Why Prolog?.

1.3.1 Facts and rules

Prolog programs are built from **facts** and **rules**. A fact states something that is true:

```
requires(browser, graphics).
requires(browser, networking).
requires(graphics, fonts).
```

This says: a browser requires graphics and networking; graphics requires fonts. Each line is a fact — something the system knows to be true.

A **rule** says something is true *if* certain conditions hold:

```
needs(X, Y) :- requires(X, Y).
needs(X, Y) :- requires(X, Z), needs(Z, Y).
```

Read `:-` as "if." The first clause says: X needs Y if X directly requires Y. The second says: X needs Y if X requires some intermediate Z, and Z in turn needs Y. Together, these two lines define transitive dependency — if the browser requires graphics and graphics requires fonts, then the browser needs fonts.

1.3.2 Queries and unification

You ask Prolog questions by posing **queries**. Prolog answers by finding values that make the query true:

```
?- needs(browser, What).
What = graphics ;
What = networking ;
What = fonts.
```

Prolog found everything the browser transitively needs. It did this through **unification** — matching the variable `What` against terms in the database — and **backtracking** — systematically trying every possibility.

Unification is more powerful than pattern matching. Two terms unify if there exists a substitution that makes them identical:

```
?- package(Name, stable) = package(editor, Status).
Name = editor, Status = stable.
```

Prolog figured out that `Name` must be `editor` and `Status` must be `stable` for both sides to match. This works in both directions — Prolog does not distinguish between "input" and "output" arguments.

1.3.3 Backtracking

When a Prolog query has multiple solutions, the runtime explores them through **backtracking**. Consider:

```
color(red).
color(green).
color(blue).

?- color(X).
X = red ;
X = green ;
X = blue.
```

Each `;` triggers backtracking: Prolog undoes its last choice and tries the next alternative. This search is built into the language — you do not write a search loop.

1.3.4 Compound terms

Prolog terms can be nested, forming structured data without defining classes or schemas. For example, a package entry might look like:

```
package(editor, version(2, 4, 1), [unicode, spellcheck]).
```

This single term captures a package name, a structured version, and a list of enabled features. Because Prolog comparison (`compare/3`) works structurally on compound terms, two versions can be compared directly — no custom comparator needed. portage-ng uses compound terms extensively to represent versions, dependencies, and proof entries.

1.3.5 Lists and association lists

Prolog lists are linked lists built from `[Head|Tail]`:

```
?- [a, b, c] = [H|T].
H = a, T = [b, c].
```

Looking up a value in a plain list requires walking it from head to tail — O(n) in the worst case. When a proof tree contains thousands of entries, this becomes a bottleneck.

SWI-Prolog provides **association lists** (AVL trees) via `library(assoc)` as an efficient alternative. An AVL tree is a self-balancing binary search tree: keys are kept in order, and the tree is rebalanced after every insertion so that no branch is more than one level deeper than its sibling.

To find a key, we do not scan every element. Instead, we compare the target with the current node and follow the appropriate branch — left if the target is smaller, right if it is larger. The following diagram shows how looking up “ssl” in a tree of seven entries requires only three comparisons:

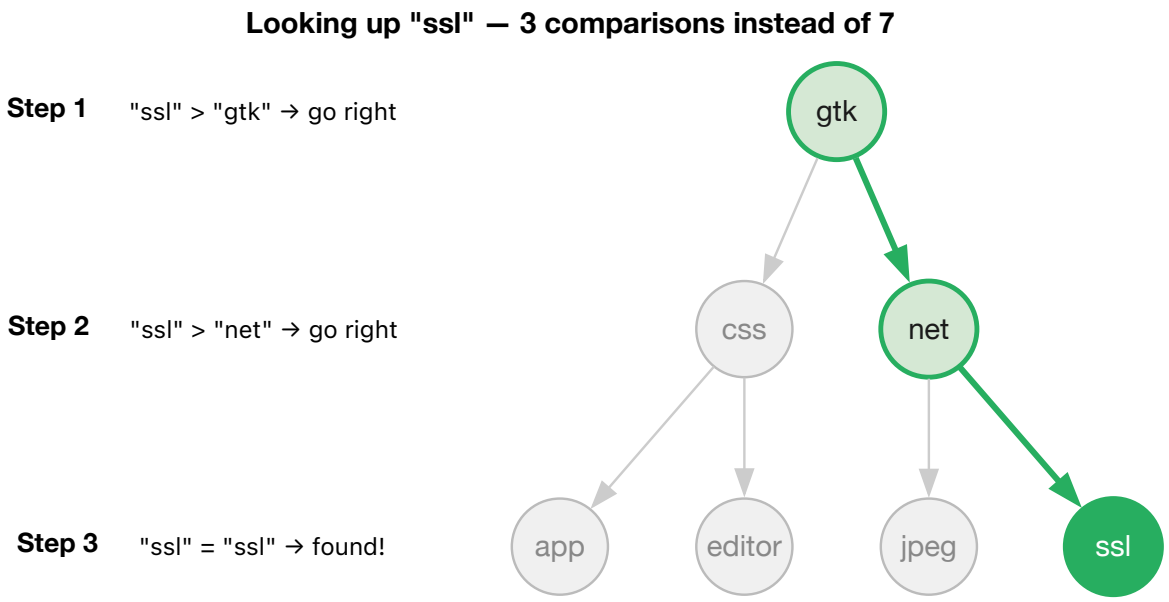


Figure 1 — AVL tree lookup

Because the tree stays balanced, every lookup follows a single path from root to leaf. The length of that path is at most $\log_2(n)$ — with 10,000 entries, an AVL lookup visits at most 14 nodes instead of scanning all 10,000.

portage-ng uses association lists extensively — for the proof, the model, the trigger set, and the constraint store:

```
?- empty_assoc(E),
   put_assoc(editor, E, installed, A1),
   put_assoc(browser, A1, pending, A2),
   get_assoc(editor, A2, Status).
Status = installed.
```

All operations (`get_assoc`, `put_assoc`) are O(log n), which makes them practical for the data structures at the heart of the prover.

1.3.6 Definite Clause Grammars

When portage-ng reads a package's dependency specification, it needs to parse structured text like `>=dev-libs/openssl-1.1:0=` into Prolog terms the prover can reason about. In most languages, writing a parser means writing imperative code — loops, state machines, error handling. In Prolog, you can write the grammar itself as a program.

A **DCG** (Definite Clause Grammar) lets you describe what valid input looks like, declaratively. Prolog takes care of matching the input against the grammar rules. For example, a simple grammar for a greeting:

```
greeting --> [hello], name.
name --> [world].
name --> [prolog].
```

This reads naturally: "a greeting is the word `hello` followed by a name; a name is either `world` or `prolog`." To check whether a sequence matches:

```
?- phrase(greeting, [hello, world]).
true.

?- phrase(greeting, [hello, cat]).
false.
```

The grammar *is* the parser — there is no separate parsing step. portage-ng uses DCGs to parse the **EAPI** (Ebuild API) dependency specification language — EAPI is the versioned interface that defines the syntax and semantics of Gentoo's ebuild format, including version ranges, Use conditionals, slot operators, and choice groups. The result is a parser that reads like a specification of the language it accepts, making it easier to verify, extend, and maintain.

1.3.7 Meta-programming

One of Prolog's most distinctive features is that programs and data are made of the same material: **terms**. A rule like `requires(browser, graphics)` is not just an instruction — it is a data structure that a program can inspect, build, and pass around. This blurs the line between "the program" and "the data it operates on" in a way that is natural in Prolog but awkward in most other languages.

Why does this matter for portage-ng? Because the prover does not just compute a plan — it builds a **proof** that explains why the plan is correct. As the prover works, it constructs a term that records every decision:

```
proof(browser, [
  rule(requires(browser, graphics), [
    proof(graphics, [
      rule(requires(graphics, fonts), [
        proof(fonts, [fact])
      ])
    ])
  ])
]).
```

This proof term says: "the browser is in the plan because it requires graphics, which is in the plan because it requires fonts, which is a base fact." The proof is not a side effect or a log — it is a first-class Prolog term that can be queried, compared, and transformed like any other data.

The same principle applies to assumptions. When the prover cannot satisfy a dependency without, say, accepting a testing keyword, it records that assumption as a term inside the proof:

```
assumed(accept_keywords('~amd64', graphics))
```

At the end, portage-ng can walk the proof, collect all assumptions, and present them to the user as actionable suggestions — precisely because assumptions are data, not scattered side effects.

This capability is called **reification**: turning the process of reasoning into data that can itself be reasoned about. It is what makes the "every plan is a proof" architecture natural in Prolog.

1.4 Why Prolog?

Now that you have seen the basics, here is why Prolog is not just a possible implementation language but the *right* one for building a reasoning engine for software configuration management.

1.4.1 The primitives match the problem

A reasoning engine for configuration management needs a small set of core operations. In Prolog, these operations are built-in primitives rather than library code:

Reasoning need	Prolog primitive
Configuration rules	Horn clauses
Structured data	Compound terms
Search with backtracking	Built-in backward chaining with backtracking
Constraint propagation	Unification and association lists
Specification grammar	Definite Clause Grammars
Proof construction	Terms as data (reification)
Meta-programming	Runtime assertion and introspection

Imperative package managers must re-implement all of these. Portage's `depgraph.py` implements its own retry loop with mask accumulation. pkgcore's `merge_plan` implements its own frame stack and state checkpoints for backtracking. Paludis's `decider.cc` implements its own constraint accumulator with exception-driven restart. All three re-invent mechanisms that Prolog provides as proven primitives. By relying on these primitives, the codebase becomes easier to read and understand — developers can focus fully on declaring domain knowledge rather than maintaining search and backtracking machinery.

1.4.2 Meta-level reasoning

Beyond the primitives, Prolog enables **meta-level reasoning** — the ability to reason about the reasoning process itself. For a configuration management engine, this is the decisive advantage.

Reifying assumptions. When the prover cannot satisfy a dependency, it does not throw an error. It records an assumption as a first-class term in the proof tree. The assumption carries structured metadata: why it was made, what alternatives were tried, and what the user can do about it. This is natural in Prolog because proofs are data.

Consider what happens when a package requires a keyword that is not accepted. An imperative resolver would either fail or silently accept the keyword. portage-ng records:

```
rule(assumed(portage://dev-libs/foo-1.0:run), [])
```

(The real literal's context also carries `assumption_reason(...)` and `suggestion(accept_keyword, ...)` tags.) This assumption appears in the proof, flows through the plan, and is presented to the user as: "I assumed dev-libs/foo-1.0 could be merged. To make this work, I will add it for you to package.accept_keywords."

Inspecting proof trees. The explainer module queries the proof AVL to answer "why is this package in the plan?" without re-running the resolver. In an imperative resolver, this would require instrumenting the search loop with logging and replaying it.

Learning from failures. When a proof attempt fails, the prover extracts a learned constraint — a narrowed version domain — from the failure and carries it into the next attempt. This is analogous to CDCL (Conflict-Driven Clause Learning) in SAT solvers, but expressed as Prolog term manipulation rather than boolean clause generation.

1.4.3 Declarative vs. imperative

The difference is not just stylistic. A declarative specification of configuration rules is:

- **Auditable** — the rules can be read as logical statements
- **Testable** — individual rules can be queried in isolation
- **Extensible** — adding a new dependency type means adding clauses, not modifying control flow

In portage-ng, the entire EAPI specification — hundreds of pages of the PMS (Package Manager Specification, the formal document that defines how Gentoo ebuilds, dependencies, USE flags, slots, and all related metadata must be interpreted) — is captured in a set of DCG grammar rules and Prolog clauses. Each time a new EAPI revision introduces features, supporting them is a matter of adding grammar rules and clauses. The reasoning engine — the prover and the ordering laws — does not change at all, because it is domain-agnostic.

1.4.4 Beyond pure Prolog

Prolog's strengths — backtracking, unification, reification — provide the foundation, but building a large-scale reasoning engine also requires structure that pure Prolog does not offer out of the box. portage-ng extends standard Prolog with two mechanisms: **feature terms** for carrying structured metadata through proofs, and **contexts** for organising code into encapsulated, independently stateful components.

1.4.5 Feature logic and feature terms

In classical logic programming, everything is built from **terms**, and a term is one of three things. An **atom** names a specific individual directly: `openssl`, `amd64`, `stable`. A **variable** — written with an initial capital, like `Version` or `X` — stands for an individual that is not yet known; unification may later bind it to a concrete term. A **compound term** names an individual *by description*: a functor applied to arguments, as in `version(2, 4, 1)` — "the version with components 2, 4 and 1". Just as the phrase "the mother of John" identifies a person without stating her name, the compound term `mother_of(john)` denotes an individual through its relationship to another one. And because arguments may themselves be compound, descriptions nest to arbitrary depth — `mother_of(mother_of(john))` is the mother of the mother of John. With only these

three kinds of terms, a finite program can describe and reason about arbitrarily large — even infinite — domains.

Three kinds of terms are enough for simple reasoning, but when the prover needs to carry *configuration alongside identity* — "this package, with these USE flags, in this slot, at this proof depth" — a flat compound quickly becomes unwieldy. Feature logic, originally developed by Hassan Aït-Kaci and others for computational linguistics, offers a cleaner model: a **feature term** is a structured record of named attributes (features) whose values may themselves be feature terms. Two feature terms can be **unified** (merged) non-destructively, combining their information while checking for consistency.

Andreas Zeller applied feature logic directly to software configuration management, showing that features and feature unification provide a natural formalism for describing and merging software configurations — capturing version selections, build options, and platform constraints as feature terms rather than ad-hoc data structures. portage-ng builds on this insight: USE flags, slot constraints, version domains, and proof context are all represented as feature terms that the prover unifies as it expands the dependency graph.

portage-ng uses the `?{}` notation to attach a feature term to any literal. The syntax reads as "this literal, *qualified by* these features":

```
portage://app-editors/neovim-0.12.0:run?{[]}
```

Here the feature term `{[]}` is empty — no additional constraints beyond the literal itself. As the prover expands dependencies and resolves USE flags, the feature term accumulates information:

```
portage://app-editors/neovim-0.12.0:run?{[nvimpager, naf(test)]}
```

The feature term `{[nvimpager, naf(test)]}` records that the `nvimpager` USE flag is enabled and `test` is disabled (`naf` stands for "negation as failure" — the standard logic-programming notation for default negation).

Feature unification is the operation that merges two feature terms. When the prover encounters the same package from two different dependency paths, each carrying its own feature term, unification combines them:

- **Plain items** (like USE flags) are collected by union.
- **Constrained sets** `{L}` use intersection semantics — both paths must agree.
- **Keyed values** (`feature:value` pairs) are unified recursively.
- If a term and its negation (`naf(X)` vs. `X`) both appear, unification **fails** — this signals a genuine conflict in the dependency graph.

This mechanism is domain-agnostic: the unifier (`Source/Logic/unify.pl`) does not mention USE flags, slots, or ebuilds. It operates on abstract feature terms. The Gentoo domain layer maps USE flags, slot constraints, and version domains onto feature terms; the unifier simply merges them. A domain hook (`feature_unification:val_hook/3`) allows domain-specific value types (e.g. version domain intersection) to participate in unification without modifying the core.

The result is that every literal in a proof carries a complete, machine-readable description of its resolved configuration. The orderer and printer can read this directly — there is no need to re-derive "which USE flags were active" after the fact.

1.4.6 Contextual object-oriented programming

Prolog operates under the **closed-world assumption**: what cannot be derived from the program is considered false. The classic illustration is an airline database: a flight that is not listed does not exist. Nobody enumerates all the flights that do *not* operate — absence itself is the negative information. This convention is what makes negation practical, but it carries an obligation: the program must be a *complete* description of its world. An answer like "no version of this package satisfies the constraint" is only trustworthy if every version the question ranges over is actually in the program.

Now consider the world portage-ng reasons about: multiple repositories, each with tens of thousands of ebuilds, every ebuild in several versions, each version with its own dependencies, USE flags, and keywords — plus profiles, user configuration, and the installed system. Treating all of that as one flat closed world is semantically correct but practically untenable. Facts would have to be discovered lazily from disk in the middle of proofs, so completeness — the very thing the closed-world assumption rests on — could never be guaranteed at any given moment; and every negative answer would implicitly quantify over one enormous, undifferentiated fact base in which most facts are irrelevant to the question being asked.

portage-ng therefore **computes a closure** before reasoning begins: for each component, the complete set of facts describing it is materialised up front. Syncing a repository parses every ebuild's metadata and asserts the result into that repository's own context; the profile and the user configuration are expanded into theirs; the installed system (VDB) gets its own. Each context is now a small world that is genuinely closed *by construction* — it contains everything there is to know about its component. Within such a boundary, negation and enumeration are both sound and fast: "no version of X in this repository satisfies C" ranges over exactly one context's in-memory, indexed facts rather than over an open-ended outside world.

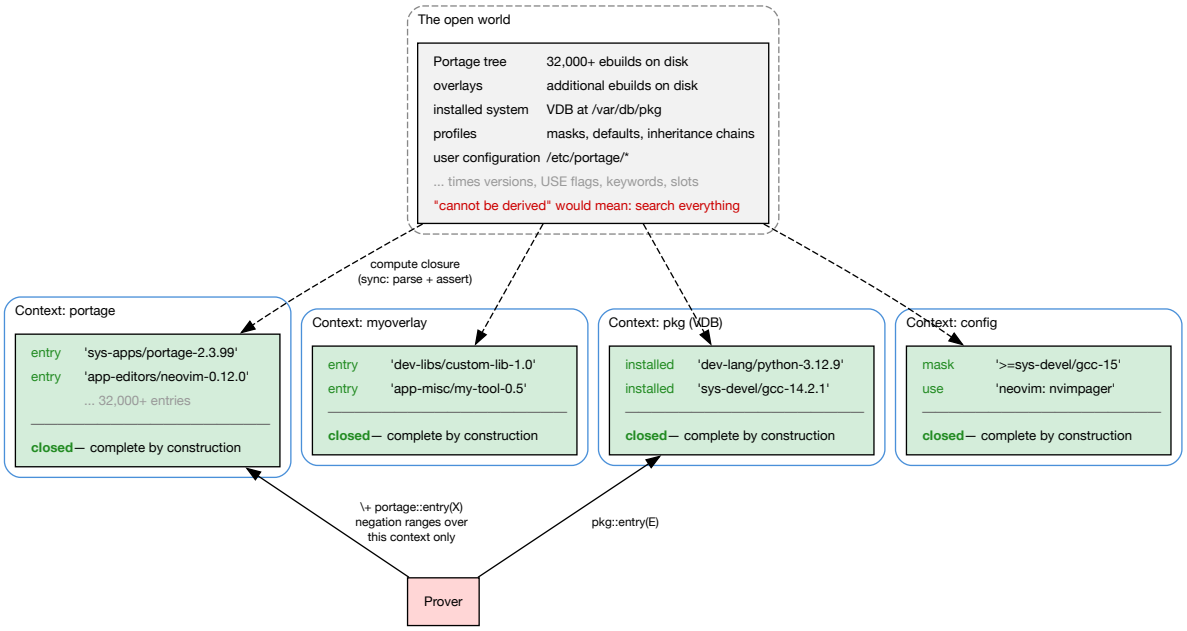


Figure 2 — Computing closures over an open world

The closed-world assumption still holds — but per context, within a well-defined boundary, which makes reasoning both tractable and modular. What remains is an engineering question: how do we organise these scoped contexts as first-class objects in Prolog?

Standard Prolog module systems offer some namespacing, but they were designed for library organisation, not for modelling independent entities that each carry their own state, rules, and encapsulation boundaries. We need a way for each component to have its own context — its own facts, its own rules — with explicit declarations of what is public interface and what is private implementation. Different repositories and different configurations must be able to coexist without interfering with each other's reasoning.

In the object-oriented world, this problem was solved long ago with encapsulation and access control. The challenge is bringing those organisational benefits to Prolog without sacrificing its declarative nature. The rules inside a context must remain ordinary logical clauses — directly translatable into traditional logic — while the context system provides the structure around them.

portage-ng needed object-oriented style programming — classes, instances, encapsulation, access control — but at **runtime**, because repositories and configurations are discovered and instantiated at startup, not known at compile time. No existing Prolog library provided this. **Logtalk**, the best-known approach to object-oriented logic programming, works by compile-time translation: source files are transformed into plain Prolog before execution. That model does not fit a system that creates and composes objects dynamically.

So portage-ng implements its own runtime object system called **context** (implemented in `context.pl`). The syntax is deliberately Logtalk-like — `::-` for method clauses, `::` for message sends, `dpublic`, `dprotected`, `dprivate` for access control — but the underlying mechanism is entirely different: contexts are created, cloned, and composed at runtime through Prolog's own assert/retract machinery. There is no compilation step and no source-to-source transformation.

To illustrate, here is a simplified example of a `person` context:

```
:- module(person, []).
:- class.

:- dpublic([person/1, '~person'/0]).
:- dpublic([get_name/1, set_name/1]).
:- dpublic([get_age/1, set_age/1]).
:- dpublic([get_title/1, add_title/1, remove_title/1]).
:- dprivate(age/1).
:- dprivate(title/1).
:- dprotected(name/1).

person(Name) ::-
  :set_name(Name).

'~person' ::-
  :this(Context),
  write('Person destructor - '), write(Context), nl.

get_name(Name) ::-
  ::name(Name).

set_name(Name) ::-
  <=name(Name).

get_age(Age) ::-
  ::age(Age).

set_age(Age) ::-
  <=age(Age).

get_title(Title) ::-
  ::title(Title).

add_title(Title) ::-
  <+title(Title).

remove_title(Title) ::-
  <-title(Title).

age(Age) ::-
  number(Age), Age > 0.
```

Several things are worth noting. The `:- class` directive declares that this module defines a context class. The `dpublic`, `dprotected`, and `dprivate` directives specify access control — just like in classical OO, public predicates can be called by anyone, protected predicates only by the class and its descendants, and private predicates only within the class itself. The constructor `person/1` initialises the instance by setting its name; the destructor `~person` is called when the instance is destroyed. The `::-` operator (instead of Prolog's standard `:-`) marks instance methods: their clauses are guarded at runtime so that each instance operates on its own state. The `::` prefix reads instance data, `<=` assigns it (replacing any previous value), `<+` adds a fact to

the instance, and `<-` removes one — as shown by `add_title` and `remove_title`, which allow a person to accumulate multiple titles. The `:` prefix calls other methods on the same instance.

Using this class is straightforward:

```
?- pieter:newinstance(person).
true.

?- pieter:person('Pieter').
true.

?- pieter:set_age(40).
true.

?- pieter:add_title('Dr.').
true.

?- pieter:add_title('Prof.').
true.

?- pieter:get_age(Age).
Age = 40.

?- pieter:get_title(Title).
Title = 'Dr.' ;
Title = 'Prof.'.
```

The `newinstance` call creates an instance named `pieter` from the `person` class, and the constructor is invoked with `pieter:person('Pieter')`. From that point on, `pieter` is a live context with its own state. Setting the age and adding titles modifies that instance's private data. Querying titles backtracks over all titles that were added — this is Prolog's backtracking working naturally within the context system.

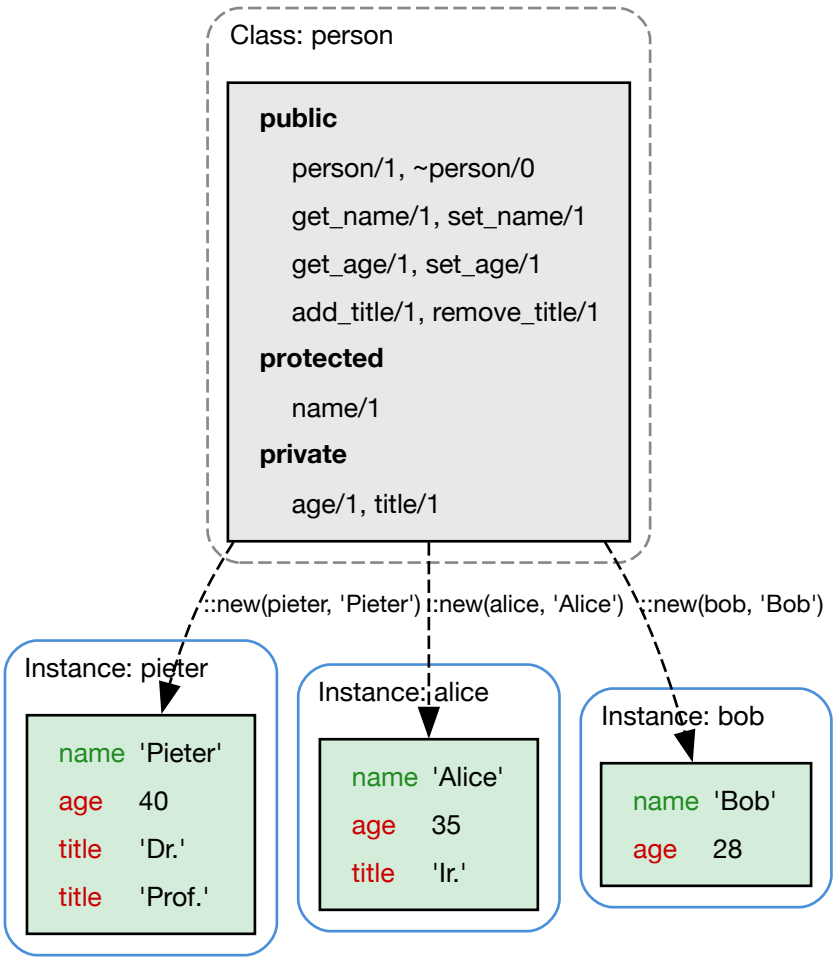


Figure 3 — Person class and instances

Each instance carries its own state: `pieter` has two titles and age 40, `alice` has one title and age 35, `bob` has no titles and age 28. The class defines the shape; each instance fills it independently.

In portage-ng, repositories are context objects: each has a name, a set of entries, and operations like `sync`, `graph`, and `query` that are dispatched through the context. The prover does not need to know which repository it is reasoning about — it receives a context and operates through it.

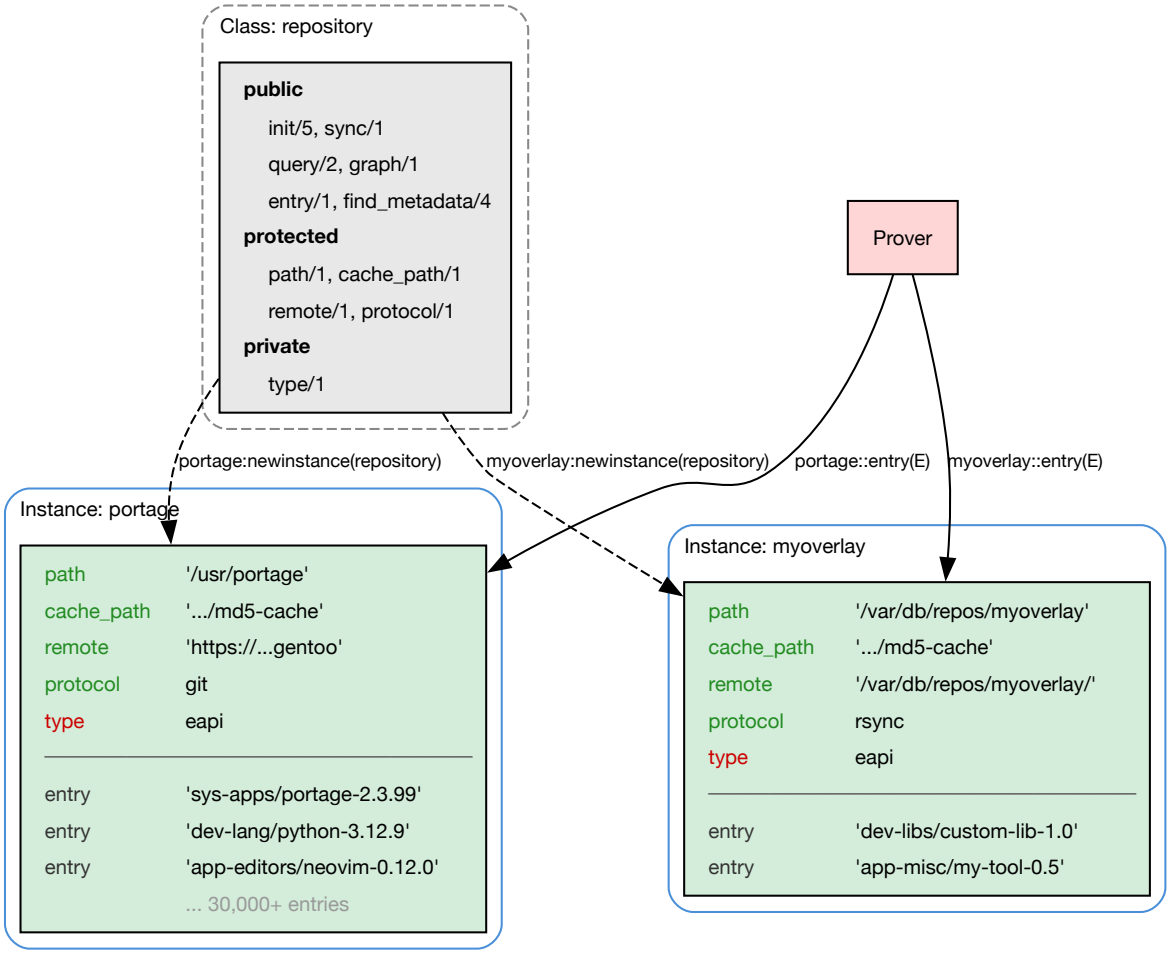


Figure 4 — Repository class and instances

The `repository` class defines the interface — `init`, `sync`, `entry`, `query` — while each instance holds its own path, protocol, and entries. The prover queries `portage:entry(E)` and `myoverlay:entry(E)` through the same module-qualified call; the context system dispatches each call to the right instance's private data.

This approach brings encapsulation, polymorphism, and modularity to Prolog while preserving its declarative core. The rules inside a context are ordinary Prolog clauses; the context system simply controls visibility and dispatches calls to the right instance. The full design is covered in chapters 21 and 22.

The `::-` operator is also portage-ng's logo — the banner printed at startup reads `::- portage-ng`. The symbolism is deliberate. Prolog writes the implication arrow of a logical rule as `:-`. The double colon `::` is the scope operator of the object-oriented tradition — and, by a happy coincidence, the repository qualifier every Gentoo user knows from atoms like `dev-lang/python::gentoo`. Superimpose the two, sharing the middle colon, and you get `::-`: a logical rule that holds within a context. Contextual logic programming, applied to Gentoo — the whole project in three characters.

1.4.7 Feature logic meets contextual logic

There is a natural synergy between feature logic and contextual logic programming that is worth noting. A context — with its named attributes, access control levels, and instance-specific state — can be viewed as a special kind of feature term: a structured record of features (name, age, title, path, protocol, …) augmented with meta-level annotations (public, protected, private) that control which features are visible to which parts of the program. Conversely, feature unification can be seen as a restricted form of context composition: merging two feature terms is analogous to combining two contexts, with consistency checking playing the role of access control.

This perspective suggests that feature logic and contextual logic programming are two views of the same underlying formalism — one emphasising data (feature terms as structured records) and the other emphasising behaviour (contexts as encapsulated reasoning units). A unified framework that captures both would be a natural extension: feature terms with access-controlled attributes, or contexts whose state is described and merged via feature unification. This remains an open area for further exploration.

1.4.8 Pengines: contexts over the network

SWI-Prolog provides **Pengines** (Prolog Engines) — lightweight, sandboxed Prolog instances that can be created, queried, and destroyed over HTTP. Each Pengine is an isolated reasoning environment with its own clause store, running inside a host server process. Clients on remote machines can create a Pengine, send it a query, and receive results — all over a standard HTTPS connection.

The connection to contextual logic programming is direct: a Pengine is, in essence, a **remote context**. Just as a local context encapsulates its own state and exposes a public interface, a Pengine encapsulates a Prolog environment and exposes it over the network. The access control that contexts provide locally (public, protected, private) is mirrored by the Pengine sandbox, which restricts which predicates the remote client may call.

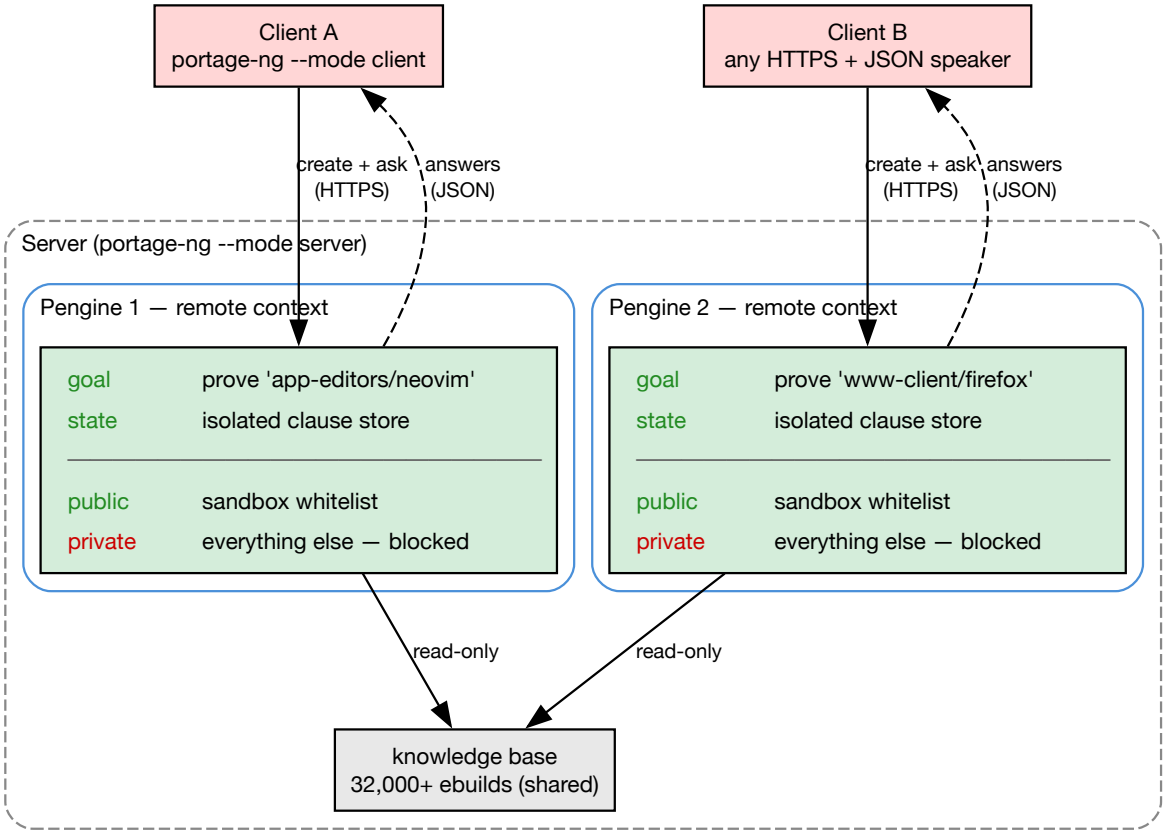


Figure 5 — Pengines: remote contexts over HTTPS

Each client interaction gets its own Pengine: an isolated goal, its own clause store, and a sandbox whitelist as its public interface. Both Pengines read the same shared knowledge base, but neither can modify the server nor observe the other — exactly the encapsulation a local context provides, transported over HTTPS.

portage-ng uses Pengines in its server mode. The server hosts the knowledge base and exposes it through a Pengine application: clients and workers create Pengines on the server, submit proving goals, and receive plan results — all without needing a local copy of the knowledge base. From the client's perspective, the interaction looks like a local Prolog query; the network transport is transparent. This is what makes client–server mode practical for embedded and resource-constrained devices: the full reasoning context lives on the server, and the client merely drives it.

1.5 The "every plan is a proof" philosophy

These ideas come together in portage-ng's central insight: a build plan should not be the output of a search algorithm that happens to terminate. It should be a **proof object** — a term that records, for every package, which rule justified its inclusion and under what constraints.

This gives three properties that traditional resolvers lack:

1. **Completeness.** If a valid plan exists, the prover finds it. If no valid plan exists, the prover completes with explicit assumptions that document exactly where the specification is unsatisfiable.
2. **Explainability.** Every package in the plan can be traced back through the proof tree to the user's original target. "Why is this package here?" is answered by inspecting the proof, not by re-running the resolver.
3. **Reproducibility.** The proof is a first-class Prolog term. Given the same Portage tree, VDB, and configuration, the same proof is produced every time.

1.6 How portage-ng relates to other resolvers

portage-ng is not a rewrite of Portage in Prolog. It is a fundamentally different approach to the same problem:

	Portage	**pkgcore**	**Paludis**	**portage-ng**
Language	Python	Python	C++	SWI-Prolog
Model	Greedy graph + retry	Frame-stack DFS + backtrack	Constraint accumulator + restart	Inductive proof search
Conflicts	Retries with mask accumulation	Backtrack to frame checkpoint; next choice	Restarts with preloads	Iterative refinement with learned domains
Completeness	Sometimes fails	Sometimes fails	May exhaust restarts	Always produces a plan
Guarantees	None	None	None	Every plan is a proof

For a detailed comparison of the reasoning models, see Chapter 23: Resolver Comparison.

1.7 A brief history

The author's involvement with Gentoo began in 2002 as the founder of the first architecture port — PowerPC. That work contributed to the keyword system described above and to expanding the range of platforms where Gentoo could run. In 2003, a formal top-level management structure was implemented for the Gentoo project (GLEP 4). Under this structure, the author served as a senior manager for Gentoo with both strategic and operational responsibility for three areas: Gentoo on alternative operating systems and LiveCD technology, developer tools, and package manager research (Portage). That experience — porting across architectures, managing the resulting configuration complexity, and researching the limits of Portage's imperative resolver — motivated the portage-ng project.

portage-ng began in 2005 as an experiment in applying logic programming to software configuration management. The initial question was simple: could Prolog's built-in search and backtracking replace the hand-written solver in Portage?

The answer turned out to be deeper than expected. Prolog did not just replace the solver — it changed what was possible. The ability to reify proofs meant build plans became inspectable objects. The ability to record assumptions meant the resolver never had to give up. The ability to parse grammars with DCGs meant the EAPI specification could be expressed directly as code.

Over two decades of development, portage-ng has evolved from a proof-of-concept into a full-featured configuration management front-end with full PMS / EAPI compliance, distributed proving, LLM-assisted plan explanation, and measured correctness against Portage across the entire Gentoo tree.

1.8 Further reading

- Chapter 2: Installation and Quick Start — getting portage-ng running on your machine
- Chapter 4: Architecture Overview — how the six pipeline stages fit together
- Chapter 8: The Prover — the inductive proof engine in detail
- Chapter 23: Resolver Comparison — deep dive into how portage-ng compares with Portage, Paludis, and pkgcore
- Chapter 25: Position in the Solver Literature — where the prover sits among feature logic, ordered logic, answer-set solvers and the package solvers that followed them

Chapter 2

Installation and Quick Start

2.1 Prerequisites

2.1.1 Required

The following tools must be present on every system that runs portage-ng. SWI-Prolog is the runtime; the others are used during repository syncing, metadata extraction, and distfile verification.

Dependency	Minimum version	Purpose
SWI-Prolog	10.0.0	Runtime interpreter. Must be built with SSL, PCRE, HTTP, crypto, and pengines support. Line editing (libedit or GNU readline) is optional; see below.
bash	5	Metadata extraction via `ebuild-depend.sh` and helper scripts.
git	any	Repository syncing (`--sync` with git protocol), version display.
curl	any	Mirror/distfile downloads, HTTP-based repository sync.
openssl CLI	any	Distfile hash verification (`openssl dgst`), TLS certificate generation for client–server encryption.
Gentoo Portage tree	—	A full Portage tree (ebuilds + md5-cache). portage-ng reads the md5-cache for dependency resolution and requires the ebuilds for building.

On most Gentoo systems these are already installed. On non-Gentoo hosts (e.g. macOS), SWI-Prolog and bash are the only items you may need to install manually.

2.1.2 Required for specific features

Some portage-ng features require additional tools. These are only needed for specific commands.

Dependency	Feature	Notes
Graphviz (>= 11)	`--graph`	The `dot` command generates interactive SVG dependency graphs.
dns-sd	Distributed mode	mDNS/Bonjour service discovery. Built-in on macOS; use `avahi-browse` on Linux.
ebuild	`--merge` / `--build`	Actual package building delegates to Portage's ebuild infrastructure. Not needed for `--pretend`.
rsync	`--sync` (rsync)	Only when using rsync-based repository sync.
tar	`--sync` (HTTP)	Only when using tarball-based repository sync.

2.1.3 Optional

The following are convenient but not required for core operation.

Dependency	Purpose
Python 3	Timeout watchdog in the dev wrapper.
make / **cmake**	Used by build helper scripts for packages that need them.
aha / **perl**	Pretty-print HTML output generation.
pv	Progress bars during batch graph generation.
Ollama	Local LLM inference and vector embeddings for `--search` / `--explain`.
MesloLGS NF (Nerd Font)	Rounded Powerline action bubbles in `--style fancy` terminal output. Download from Nerd Fonts; see Chapter 14.

2.1.4 Prolog build requirements

When compiling SWI-Prolog from source, ensure the following optional components are enabled (they are usually built by default):

- **OpenSSL** — required for `library(crypto)`, `library(ssl)`, `library(http/http_ssl_plugin)`
- **PCRE** — required for `library(pcre)` (used in EAPI parsing)
- **libedit or GNU readline** — optional. Needed only for `--shell` line editing and history. portage-ng loads `library(editline)` first and falls back to `library(readline)`. `--shell` still

starts if neither is present; up-arrow history is then unavailable. `--pretend` and `--merge` do not need either library. On Gentoo these are `dev-libs/libedit` (BSD libedit; SWI `USE=cli`) or `sys-libs/readline` (SWI `USE=readline`). Do not confuse BSD libedit with `dev-libs/editline` (Minix editline), which SWI does not use.

- **libgmp** — required for arbitrary-precision arithmetic
- **zlib** — required for qcompiled file support (`Knowledge/kb.qlf`)

2.2 Building

From the project root:

```
make check    # verify SWI-Prolog is installed
make build    # create the portage-ng binary
make install  # install to /usr/local/bin (requires sudo)
```

The `build` target uses `swipl --stand_alone=true` to produce a self-contained binary.

2.3 First run

The clone does not include a knowledge base (`Knowledge/kb.qlf` is gitignored). `--sync` is the first run after install: it materializes the Portage tree into qcompiled facts that `--pretend` and `--shell` load.

2.3.1 Sync the Portage tree

Sync the repository and regenerate the knowledge base cache:

```
portage-ng --sync
```

The sync performs three phases for each registered repository:

1. **Repository sync** — pulls the latest Portage tree (via git, rsync, or HTTP tarball depending on configuration).
2. **Metadata sync** — reads the md5-cache files and, if configured, regenerates cache entries for ebuilds that have changed.
3. **Knowledge base sync** — parses all cache entries into Prolog facts (the `cache:entry`, `cache:entry_metadata`, `cache:manifest`, etc. predicates) and saves the compiled knowledge base to disk.

```
>>> Syncing 1 registered repository

--- Syncing repository "portage" ---

 Syncing repository ... ok
 Syncing metadata   ... Ebuild: sys-apps/portage-2.3.99-r1
                        Ebuild: dev-lang/python-3.13.3
                        Ebuild: sys-libs/glibc-2.41
                        ...
```

```
                        Updated metadata.
 Syncing kb         ... Ebuild: acct-group/abrt-0
                        Ebuild: acct-group/adm-0
                        Ebuild: acct-group/audio-0
                        ...
                        Manifest: app-accessibility/at-spi2-core
                        Manifest: app-accessibility/brltty
                        ...
                        Updated prolog knowledgebase.

--- Syncing profile ---

 Saving knowledge base ... ok
```

During the knowledge base sync, every ebuild's metadata — dependencies, Use flags, keywords, slots, descriptions, manifests — is parsed and asserted as Prolog facts. The entire Gentoo repository (over 30,000 ebuilds) is held in memory as a native Prolog database, enabling lightning-fast lookups without any disk I/O during reasoning.

SWI-Prolog's just-in-time (JIT) indexing further accelerates these lookups. When a predicate like `cache:entry_metadata(portage, 'app-editors/neovim-0.12.0', description, D)` is first called, the runtime automatically builds hash indices on the arguments that are bound. Subsequent calls with the same argument pattern jump straight to matching clauses instead of scanning all 30,000+ entries linearly. This indexing is created on demand and updated transparently as facts are asserted or retracted — no manual index declarations are needed.

Once syncing completes, the knowledge base is saved to disk using SWI-Prolog's qcompile mechanism (`Knowledge/kb.qlf`). qcompile serializes Prolog clauses into a compact binary format that can be loaded back in a fraction of the time it takes to parse the original source. On subsequent runs, portage-ng loads the `.qlf` file directly, making startup near-instantaneous — even for a repository with tens of thousands of ebuilds.

2.3.2 Pretend (dry-run)

Generate a build plan without executing it:

```
portage-ng --pretend app-editors/neovim
```

portage-ng proves a dependency graph, plans it into parallel steps, and presents the result:

```
>>> Emerging : portage://app-editors/neovim-0.12.0:run?{[]}

These are the packages that would be merged, in order:

Calculating dependencies... done!

 └─ step 1 ─┤ download portage://dev-python/tree-sitter-0.25.2
            │           └─ file ─┤ 170.27 Kb  tree-sitter-0.25.2.gh.tar.gz
            │ download portage://dev-lua/mpack-1.0.13
            │           └─ file ─┤ 16.17 Kb   mpack-1.0.13.tar.gz
```

```
  └─ step 2 ─┤ install  portage://dev-lang/lua-5.1.5-r200
             │           └─ conf ─┤ USE = "readline deprecated"
             │                    │ SLOT = "5.1"
             │ install  portage://dev-libs/msgpack-6.0.0-r1
             │           └─ conf ─┤ USE = "-doc -examples -test"
             │                    │ SLOT = "0/2-c"
            ...

  └─ step 7 ─┤ install  portage://app-editors/neovim-0.12.0
             │           └─ conf ─┤ USE = "nvimpager -test"
             │                    │ LUA_SINGLE_TARGET = "luajit -lua5-1"

  └─ step 8 ─┤ run      portage://app-editors/neovim-0.12.0

Total: 59 actions (20 downloads, 19 installs, 1 update, 19 runs),
       grouped into 8 steps.
       18.82 Mb to be downloaded.
```

Actions within the same step can execute in parallel. The plan distinguishes download, install, update, and run phases. Each package shows its resolved configuration (Use flags, slot, target selection).

If portage-ng had to make assumptions during proving, they are reported at the end with suggested fixes and draft bug reports.

2.3.3 Interactive shell

Drop into a Prolog shell with the full knowledge base loaded:

```
portage-ng --shell
```

The shell provides direct access to the knowledge base. The built-in `query:search/2` predicate offers a readable way to explore it.

Search for packages by name:

```
?- query:search([name(neovim), description(D)], Repository://Entry).
D = "Vim-fork focused on extensibility and agility",
Repository = portage,
Entry = 'app-editors/neovim-9999'.
```

Press `;` to see the next result, or `.` to stop. Prolog backtracks through all matching ebuilds automatically.

Look up slot and keywords:

```
?- query:search([name(neovim), slot(S), keywords(K)], Repository://Entry).
S = '0',
K = unstable(amd64),
Repository = portage,
Entry = 'app-editors/neovim-0.12.0'.
```

Search across repositories:

```
?- query:search([name(firefox), description(D)], Repository://Entry).
D = "Firefox Web Browser",
Repository = portage,
Entry = 'www-client/firefox-149.0'.
```

Count all ebuilds:

```
?- aggregate_all(count, portage:entry(_), Total).
Total = 31535.
```

Read a single metadata field:

```
?- cache:entry_metadata(portage, 'app-editors/neovim-0.12.0', description, D).
D = "Vim-fork focused on extensibility and agility".
```

The full cache schema and query language are documented in Chapter 6: Knowledge Base.

2.4 Running tests

```
make test            # PLUnit tests
make test-overlay    # Overlay regression tests (80 scenarios)
```

See Chapter 26: Testing and Regression for details.

2.5 Further reading

- Chapter 3: Configuration — setting up Portage tree paths, `/etc/portage`, and profiles
- Chapter 15: Command-Line Interface — full CLI reference
- `portage-ng(1)` manpage — exhaustive option reference

Chapter 3

Configuration

Dependency resolution only makes sense in *your* environment: which profile you use, which USE flags you set, which packages are already on disk, and which extra trees you layered on top of Gentoo. portage-ng is designed so you do not pay a "migration tax" to express any of that. It reads the same files and databases as traditional Portage.

Configuration, in this chapter's sense, is the act of **telling portage-ng where your machine keeps that truth** (paths, repositories, profile strategy) and **which Gentoo-side files to honour** —so the prover plans against the world you actually run, not a generic default.

This chapter starts with the central configuration file (`config.pl`), then shows how to register repositories and sync them into the knowledge base, and finally covers the `/etc/portage/` files that control policy (USE flags, masks, keywords).

3.1 The configuration file

The central configuration file is `Source/config.pl`. It is a plain Prolog source file — every setting is a Prolog fact or rule that you can read, query, or override. The file is organised into logical sections; the most important ones for getting started are summarised below.

3.1.1 General

Setting	Default	Purpose
`config:name/1`	`'portage-ng-dev'`	Program name shown in banners and logs.
`config:hostname/1`	(auto-detected)	Current hostname, used to select per-machine configuration.
`config:installation_dir/1`	(from Prolog flag)	Root of the portage-ng source tree. The knowledge base, certificates, and config files are resolved relative to this path.
`config:number_of_cpus/1`	(auto-detected)	Parallelism level for parsing, proving, and building.
`config:verbosity/1`	`debug`	Verbosity level for runtime messages.

3.1.2 Repository and metadata

Setting	Default	Purpose
`config:trust_metadata/1`	`true`	When `true`, trust the repository-shipped md5-cache. When `false`, regenerate cache entries locally for every ebuild — expensive, but useful for overlay development.
`config:write_metadata/1`	`true`	Write on-disk cache entries for locally changed or new ebuilds during sync.

3.1.3 Gentoo profile

Setting	Default	Purpose
`config:gentoo_profile/1`	`'default/linux/amd64/23.0/...'`	The Gentoo profile path relative to the Portage tree's `profiles/` directory. This must match the profile symlink on your Gentoo system.

3.1.4 Profile loading strategy

Setting	Default	Purpose
`config:profile_loading/2`	`standalone → cached`	Controls whether profile data is parsed from the Portage tree on every startup (`live`) or loaded from a pre-serialized cache (`cached`). Set per mode: standalone, daemon, worker, client, server.
`config:preference_cache/2`	`standalone → cached`	Controls whether `preference:init/0` reloads materialized state from `Knowledge/preference.qlf` when the stamp matches, or rebuilds from profile + `/etc/portage` on every startup.

See Profile loading strategy for details on generating and using the profile cache. See Preference cache for the materialized preference cache.

3.1.5 Paths

Setting	Purpose
`config:portage_confdir/1`	Path to the `/etc/portage` directory (or a development copy). Determines where `make.conf`, `package.use`, `package.mask`, etc. are read from. Comment out to use built-in fallback defaults.
`config:pkg_directory/1`	Path to the VDB directory (`/var/db/pkg` on a standard Gentoo system). Defined per host in `Source/Config/<host>.local.pl`.
`config:world_file/1`	Path to the world set file (auto-resolved from hostname).
`config:set_dir/1`	Directory of named set files (one atom list per file).
`config:glsa_dir/1`	Optional override for `$PORTDIR/metadata/glsa` (GLSA XML).
`config:glsa_injected_file/1`	Applied-GLSA id file (Portage `glsa_injected` equivalent).
`config:preserved_libs_registry/1`	Path to Portage's `preserved_libs_registry` JSON used by `@preserved-rebuild`. Default maps `…/db/pkg` → `…/lib/portage/preserved_libs_registry`.
`config:preserved_libs_registry_override/1`	Host-specific override for the preserve-libs registry path (multifile / dynamic).
`config:graph_directory/1`	Output directory for generated dependency graphs and `.merge` files. Defined per host in `Source/Config/<host>.local.pl`.
`config:build_root/1`	Root directory for build work (equivalent to Portage's `PORTAGE_TMPDIR`).
`config:build_log_dir/1`	Directory for per-package build logs.

3.1.6 Machine selection

Setting	Purpose
`config:systemconfig/1`	Resolves the machine-specific configuration file. Looks for `Source/Config/<hostname>.local.pl`; falls back to `Source/Config/default.pl` if not found.

The machine config file is where repositories are created and registered — covered in the next section.

3.1.7 Proving

Setting	Default	Purpose
`config:time_limit/1`	`300` (seconds)	Maximum time for a single proof/plan computation before aborting.
`config:proving_target/1`	`run`	Proof depth: `install` for compile-time dependencies only, `run` to include runtime dependencies.
`config:reprove_max_retries/1`	`20`	Maximum iterative learn-and-restart retries when the prover encounters conflicts.
`config:avoid_reinstall/1`	`false`	When `true`, verify already-installed packages instead of re-merging them.

3.1.8 Building

Setting	Default	Purpose
`config:build_transient_retry`	`true`	Retry a phase once when it fails with bash's PID-reuse race (`wait: pid N is not a of this shell`
`config:build_serial_retry/1`	`true`	Retry a failed compile/test/install phase once with `MAKEOPTS=-j1` to rule out parallel-make races.
`config:deconflict_collisions`	`override`	Merge-time file collision handling when a blocker atom is missing from metadata: `off` (fail as usual), `report` (fail, but surface the collision as a deconfliction assumption), or `override` (re-merge with collision protection disabled so the merge succeeds).
`config:ghc_abi_repair/1`	`true`	Repair GHC ABI-hash breakage in-transaction: rebuild the packages listed by haskell-cabal's broken-package check, then re-run the failed phase (native `haskell-updater`).
`config:ocaml_abi_repair/1`	`true`	Repair OCaml/findlib ABI breakage in-transaction: map stale compiled-unit errors to their installed owners via the VDB, rebuild them, then re-run the failed phase.
`config:subslot_rebuild/1`	`true`	Plan same-version rebuilds of installed reverse dependencies when a sub-slot (`:=`) provider's ABI changes inside a transaction. Complementary to the `@preserved-rebuild` *set* (FEATURES=preserve-libs consumers); this pass is automatic during prove/plan.
`config:toolchain_reactivatic`	`true`	Re-activate the toolchain right after a toolchain package merges, before dependent builds continue.

See Chapter 16: Building and Execution for how the retry chain and the domain exception fixups work.

3.1.9 Output

Setting	Default	Purpose
`config:default_printing_styl`	`'fancy'`	Plan output style: `short`, `column`, or `fancy` (tree-structured with Unicode box drawing and Powerline bubbles; see Chapter 14 for the MesloLGS NF font on Terminal.app).
`config:color_output/0`	asserted	ANSI colour in terminal output. Retract to disable.
`config:color_palette/1`	`full`	Use flag colouring: `easy` (classic Portage red/blue) or `full` (reason-based, showing where each flag came from).

3.1.10 Network

Setting	Default	Purpose
`config:server_host/1`	`'mac-pro.local'`	Server hostname pin for client/worker mode (Bonjour must match).
`config:server_bind/1`	`localhost`	Interface the Pengine server binds (`localhost` or `*`).
`config:server_port/1`	`4000`	HTTPS port for the Pengine server.
`config:bonjour_service/1`	`'_portage-ng._tcp.'`	mDNS service name for automatic server/worker discovery.

3.1.11 LLM integration (optional)

LLM integration is entirely optional. If you do not need `--explain`, `--llm`, or semantic search, the LLM modules can be removed from the load graph without affecting core functionality (resolving, ordering, building).

Setting	Default	Purpose
`config:llm_default/1`	`claude`	Default LLM service for `--explain` and `--llm`.
`config:llm_model/2`	(per service)	Model version for each LLM provider (ChatGPT, Claude, Gemini, Ollama, etc.).
`config:llm_use_tools/1`	`true`	Whether the LLM may execute Prolog code locally during a conversation.
`config:llm_server_calls/1`	`false`	Allow Pengines clients to call `explainer:call_llm/3` on the server (uses server API keys).
`config:curl_allow_http/1`	`false`	Permit cleartext http distfile mirrors (ftp never allowed).
`config:file_integrity/1`	`prefer`	Knowledge sidecar policy: `prefer`, `require`, or `off`.

Most settings have sensible defaults. For a typical Gentoo system, the main items to configure are `config:gentoo_profile/1`, `config:portage_confdir/1`, and the repository definitions in the machine config file.

3.2 Configuring repositories

Not every literal in a proof refers to the same backing store. portage-ng models **several repository kinds** so the resolver can combine "what exists upstream", "what is already installed", and "what I added locally" without conflating them:

Name	Role
`portage`	The main Gentoo tree, backed by md5-cache — the canonical source of buildable versions.
`pkg`	Installed packages (the VDB under `/var/db/pkg/`). Ground truth for what is already on the machine.
`overlay`	Additional ebuild trees (user overlays, testing repos, local layers), each with their own cache and sync rules.

Each repository is a named **OO instance** created with a directive like:

```
:- portage:newinstance(repository).
```

The name before `:newinstance` is the name you choose for the repository — `portage` here, but it could be anything: `:- myoverlay:newinstance(repository).` would create a repository called `myoverlay`. Each instance is specialised with a location, cache path, remote URL,

protocol, and type. This uses the same **context-based OO** machinery introduced in Chapter 1 (`Source/Logic/context.pl`): a repository is an object that responds to `sync`, `find_metadata`, `find_vdb_entry`, and so on, not a loose bag of paths.

Multiple repositories coexist in one proof. If you registered both `portage` and `myoverlay`, a dependency chain can span both: a `portage://` literal may pull in a `myoverlay://` literal when only the overlay carries a needed version. Installed packages participate too — a `pkg://` literal satisfies a runtime dependency without planning a fresh merge. Keeping repositories separate avoids conflating "what is available upstream", "what I added locally", and "what is already installed", while still allowing the prover to relate them during resolution.

Machine files in `Source/Config/` decide which of these instances exist on your host; see Machine configuration.

3.3 Machine configuration

Each machine has a configuration file under `Source/Config/` that creates and registers repository instances. portage-ng looks for `Source/Config/<hostname>.local.pl` first; if not found, it falls back to `Source/Config/default.pl`.

A machine config file creates one or more repositories using `newinstance`, initialises each with paths and a sync protocol, and registers it with the knowledge base.

The five arguments to `init` are:

1. **Local path** — where the repository lives on disk.
2. **Cache path** — the md5-cache directory inside the repository.
3. **Remote URL** — the upstream to sync from.
4. **Protocol** — how to sync: `git`, `rsync`, or `http` (tarball download).
5. **Type** — the repository format: `eapi` for standard Gentoo ebuild trees.

The examples below show the supported options.

Portage tree via git (the most common setup):

```
:- portage:newinstance(repository).
:- portage:init('/usr/portage',
                '/usr/portage/metadata/md5-cache',
                'https://github.com/gentoo-mirror/gentoo',
                'git', 'eapi').
:- kb:register(portage).
```

Portage tree via rsync:

```
:- portage:newinstance(repository).
:- portage:init('/usr/portage',
                '/usr/portage/metadata/md5-cache',
                'rsync://rsync.gentoo.org/gentoo-portage',
                'rsync', 'eapi').
:- kb:register(portage).
```

Portage tree via HTTP snapshot:

```
:- portage:newinstance(repository).
:- portage:init('/usr/portage',
                '/usr/portage/metadata/md5-cache',
                'http://distfiles.gentoo.org/releases/snapshots/current/portage-
latest.tar.bz2',
                'http', 'eapi').
:- kb:register(portage).
```

User overlay (a second ebuild tree layered on top):

```
:- myoverlay:newinstance(repository).
:- myoverlay:init('/var/db/repos/myoverlay',
                  '/var/db/repos/myoverlay/metadata/md5-cache',
                  '/var/db/repos/myoverlay/',
                  'rsync', 'eapi').
:- kb:register(myoverlay).
```

Local distfiles directory:

```
:- distfiles:newinstance(repository).
:- distfiles:init('/usr/portage/distfiles',
                  '', '', 'local', 'distfiles').
:- kb:register(distfiles).
```

Multiple repositories can be registered in the same file. During proving, the resolver queries all registered repositories and distinguishes them by their `portage://`, `pkg://`, or overlay prefix.

3.4 Syncing the tree

portage-ng does not crawl ebuild directories on every pretend merge. At runtime it works from a **compiled Prolog knowledge base** built from the same **md5-cache** files Portage uses: precomputed metadata blobs (dependencies, slots, USE defaults, and so on) produced by sourcing each ebuild through bash and extracting its declared variables — a process traditionally driven by Gentoo's `egencache`, which writes the results under the repository's cache directory. Treat the Portage tree, for resolver purposes, as **a directory of cache files** plus ebuilds; the cache is what makes bulk queries feasible.

```
portage-ng --sync
```

`--sync` is the umbrella operation that brings that picture up to date: it syncs registered repositories (via git, rsync, or snapshot download), regenerates on-disk metadata where configured, **reloads** md5-cache into dynamic Prolog facts (according to the structure defined in `cache.pl`), and **persists** the result to disk so subsequent runs start near-instantaneously.

```
portage-ng --regen
```

`--regen` (alias `--metadata`) addresses a narrower problem: **refresh the on-disk md5-cache** without performing a network sync. portage-ng can generate the md5-cache entirely on

its own — it does not need traditional Portage or `egencache` for this step. Each ebuild is sourced through bash and its metadata extracted in incremental, parallel passes (see `repository:sync(metadata)` and `config:trust_metadata/1` in the source). Traditional Portage is only needed for the actual *building* of packages, since portage-ng's current focus is on the reasoning and planning side. Note that `--regen` is not a substitute for loading facts into Prolog: after regenerating the cache, run `--sync` again so `Knowledge/kb.qlf` matches the updated on-disk cache.

You can also sync a single repository by name:

```
portage-ng --sync myoverlay
```

This syncs only the `myoverlay` repository (and saves the knowledge base afterwards). Useful when you have changed an overlay but the main Gentoo tree is still up to date.

3.4.1 Repositories

In portage-ng's architecture, all repositories are registered with a central **knowledge base** (`knowledgebase.pl`). The command-line interface talks to the knowledge base, which delegates sync operations to each registered repository. After syncing the repositories, the knowledge base also triggers a **profile sync** — this reads the Gentoo profile directory (the chain of `make.defaults`, `package.mask`, `use.mask`, etc. that define your system's baseline policy) and the `/etc/portage/` user configuration files, loading them into `preference` facts that the prover consults during resolution.

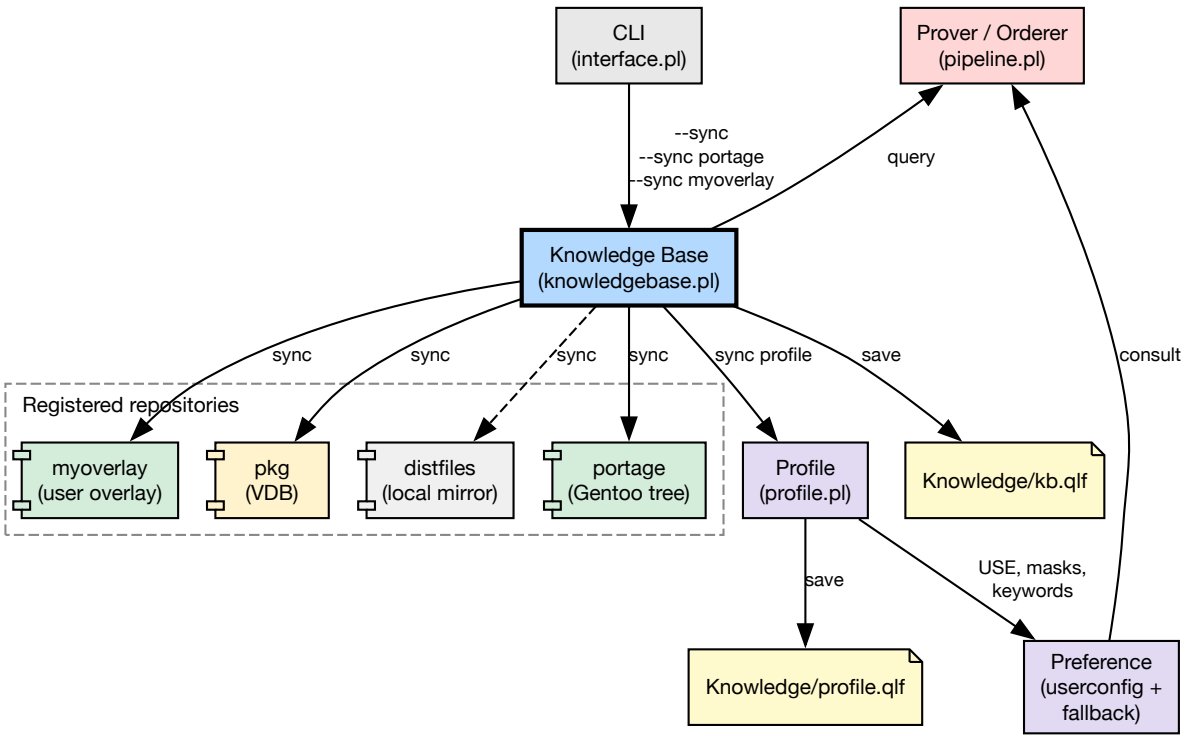


Figure 6 — Sync architecture

The result is two serialised cache files:

- **`Knowledge/kb.qlf`** — all repository and cache facts (ebuilds, metadata, manifests).
- **`Knowledge/profile.qlf`** — all profile-derived data (USE terms, masks, per-package USE, license groups).
- **`Knowledge/glsa.qlf`** — Gentoo Linux Security Advisories parsed from `metadata/glsa/` (see Chapter 20).
- **`Knowledge/preference.qlf`** — materialized preference state (built on first startup; see Preference cache).

See Profile loading strategy for details on live vs. cached profile loading.

3.4.2 Installed packages

To reason about what is already on the machine, portage-ng needs to know which packages have been installed. Portage records this in the **VDB** (Var DataBase), a directory tree at `/var/db/pkg/` with one subdirectory per installed `category/package-version`. Each subdirectory contains metadata files that capture the state at install time: dependency declarations (`DEPEND`, `RDEPEND`, `PDEPEND`), the active `USE` flags, `SLOT`, `KEYWORDS`, compiler flags, a file manifest (`CONTENTS`), and bookkeeping fields like `BUILD_TIME` and `SIZE`.

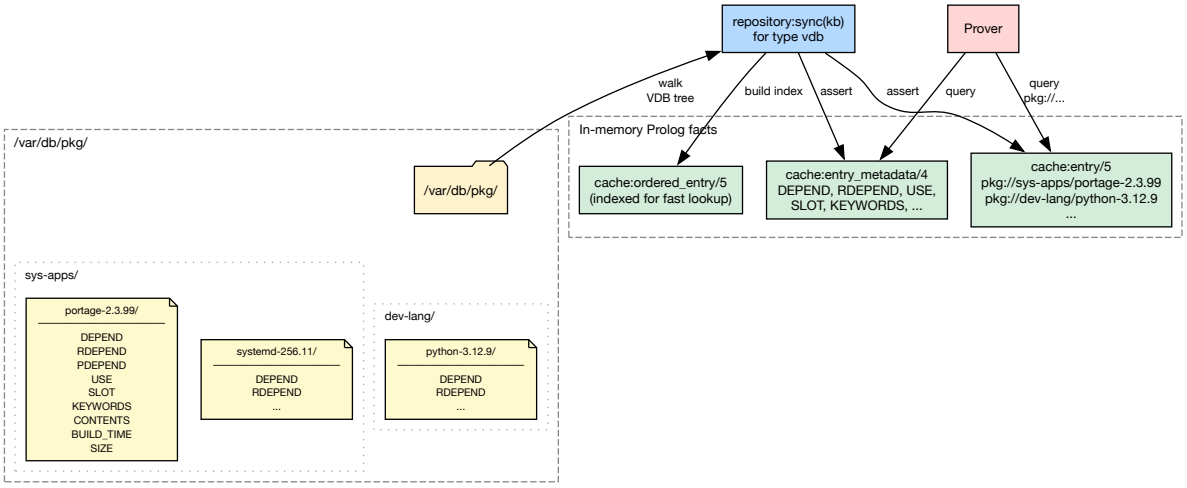


Figure 7 — VDB architecture

When `--sync` runs, the knowledge base syncs the `pkg` repository by walking the VDB tree and loading each installed package into the same in-memory fact structure used for available ebuilds. From that point on, the prover queries installed and available packages through the same interface — the only difference is the prefix: `pkg://` for installed packages, `portage://` for available ones.

This uniform representation means that during resolution, an already-installed package can satisfy a dependency directly without planning a fresh merge. In the plan output, these appear as `[nomerge]` — the prover verified the dependency is met by what is already on disk.

In client-server mode the server holds its *own* `pkg` repository, which may differ from the client's installed set. A client can upload its local VDB with `--import-vdb`; the server registers it as a per-client repository (`pkg@<clienthost>`) and uses it for that client's plans. With

`config:client_auto_import_vdb(true)` (the default) this happens automatically before each client command whenever the local VDB changed. See Chapter 18 for details.

3.5 Gentoo configuration

Gentoo users already curate policy in `/etc/portage/`: USE overrides, masks, licences, and keywords. portage-ng **reuses that investment** — it reads Gentoo's standard `/etc/portage/` configuration files, making it a drop-in replacement for dependency resolution and plan computation from a *policy* perspective.

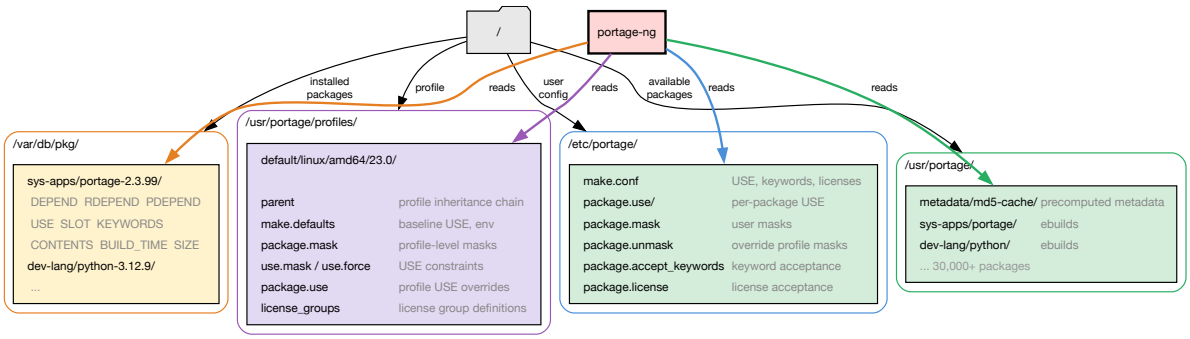


Figure 8 — Gentoo on-disk files read by portage-ng

The diagram shows the four on-disk sources portage-ng consults: user configuration under `/etc/portage/`, the profile chain under the Portage tree's `profiles/` directory, the installed-package database (VDB) under `/var/db/pkg/`, and the Portage tree itself with its ebuilds and md5-cache.

3.5.1 Supported files

portage-ng recognises the following standard Gentoo configuration files. Set `config:portage_confdir/1` in `Source/config.pl` to point at your `/etc/portage` directory (or use the bundled templates under `Source/Config/Gentoo/` during development).

File	Purpose
`make.conf`	Global environment variables (USE flags, keywords, licenses, etc.).
`package.use`	Per-package USE flag overrides.
`package.mask`	User package masks.
`package.unmask`	Overrides profile-level masks.
`package.accept_keywords`	Per-package keyword acceptance.
`package.license`	Per-package license acceptance.

All files are read from the directory set by `config:portage_confdir/1` (typically `/etc/portage/`).

These files use standard Gentoo syntax, so existing `/etc/portage/` directories work without modification.

3.5.2 File format

All files follow standard Gentoo syntax:

- Lines starting with `#` are comments
- Empty lines are ignored
- Inline `#` comments are stripped

make.conf

Bash-style `KEY="value"` assignments. Parsed by the same engine that reads profile `make.defaults` files (`profile:make_defaults_kv/2`).

```
USE="X alsa dbus -systemd"
ACCEPT_KEYWORDS="~amd64"
VIDEO_CARDS="intel"
```

package.use / package.accept_keywords / package.license

One entry per line: a package atom followed by space-separated values.

```
# package.use
app-editors/vim        -X
>=sys-libs/gdbm-1.26   berkdb

# package.accept_keywords
=sys-apps/portage-3.0  ~amd64
dev-util/pkgdev        **

# package.license
app-text/calibre       BSD
```

package.mask / package.unmask

One package atom per line (simple `cat/pkg` or versioned like `>=cat/pkg-1.0`).

```
sys-apps/systemd
>=dev-lang/python-3.13
```

3.5.3 Directory layout

All `package.*` files support both single-file and directory layouts, matching Portage's convention:

```
/etc/portage/package.use           ← single file
/etc/portage/package.use/          ← directory
/etc/portage/package.use/custom    ← files read in sorted order
/etc/portage/package.use/gaming
```

When the path is a directory, all non-hidden files in it are read in sorted (lexicographic) order.

3.5.4 Template files

For development without a real Gentoo system, portage-ng ships template configuration files in `Source/Config/Gentoo/`:

```
Source/Config/Gentoo/
  ├── make.conf
  ├── package.use
  ├── package.mask
  ├── package.unmask
  ├── package.accept_keywords
  └── package.license
```

These contain commented examples that mirror a typical Gentoo setup. On a real Gentoo system, point `config:portage_confdir/1` directly at `/etc/portage` instead.

3.6 Precedence

When the same setting appears in more than one place, portage-ng resolves it by checking sources from most specific to most general. The first match wins. For environment-like settings such as use flags, keywords, licenses, etc., the lookup order is:

1. **Command-line environment variables** — values passed on the command line override everything else.
2. **make.conf** — your `/etc/portage/make.conf` settings come next.
3. **Configuration templates** — defaults provided by the portage-ng configuration templates (under `Source/Config/Gentoo/`). These serve as a development baseline when no real `/etc/portage/` is configured.
4. **Built-in defaults** — hard-coded baseline values when nothing else is specified.

Package masks and per-package USE overrides follow a similar layering. Gentoo's profile tree is applied first (the chain of `package.mask`, `package.use`, `use.mask`, and `use.force` files that define baseline policy for your chosen profile). Your `/etc/portage/` files are applied on top, so they can override profile-level decisions. Finally, fallback defaults fill in anything left unspecified.

In practice this means your `/etc/portage/` customisations always take priority over profile defaults, and anything you pass on the command line takes priority over both.

3.7 Profile loading strategy

Profile data (USE flags, masks, per-package USE, license groups) can be loaded in two ways each time portage-ng starts:

- **Live** — the Gentoo profile tree is parsed from disk on every startup. This is the most accurate option because it always reflects the latest state of the profile, but it takes a moment longer to start.

- **Cached** — profile data is loaded from a pre-serialized cache file (`Knowledge/profile.qlf`). This makes startup near-instantaneous, but the cache must be regenerated (via `--sync`) whenever the profile changes.

The strategy is set per operating mode in `config.pl`. portage-ng supports several modes of operation (standalone, daemon, worker, client, server — see Chapter 15: Command-Line Interface for details). Each mode can use a different loading strategy:

```
config:profile_loading(standalone, cached).
config:profile_loading(daemon,     cached).
config:profile_loading(worker,     cached).
config:profile_loading(client,     live).
config:profile_loading(server,     cached).
```

If the cached strategy is set but `Knowledge/profile.qlf` does not exist yet, portage-ng falls back to live loading automatically.

3.7.1 Generating the profile cache

The `--sync` command generates `Knowledge/profile.qlf` automatically:

```
portage-ng --sync
```

After syncing all repositories, portage-ng walks the profile tree once and serializes all profile-derived data to disk. Subsequent runs that use the `cached` strategy load this file instead of re-parsing the profile tree.

3.7.2 What gets cached

The profile cache captures the following data so it does not need to be re-derived from the profile tree on each startup:

Data	Source files
USE flag defaults	`make.defaults` along the profile chain
USE masks and forced flags	`use.mask`, `use.force`
Package masks	`package.mask`, `package.unmask`
Per-package USE overrides	`package.use`
Per-package USE masks and forced flags	`package.use.mask`, `package.use.force`
License groups	`license_groups`

3.8 Preference cache

After profile and `/etc/portage` data are merged, `preference:init/0` normally materializes a large set of dynamic facts (global USE, package masks, per-package USE overrides, license

groups, world snapshots). That work can take a few seconds when profile masks must be matched against the full portage cache.

To avoid repeating that on every startup, portage-ng can persist the **materialized preference state** to disk:

File	Role
`Knowledge/preference.raw`	Textual serialization of preference facts (intermediate)
`Knowledge/preference.qlf`	QLF-compiled reload unit
`Knowledge/preference.stamp`	Fingerprint of inputs (file mtimes + env vars)

The cache is **regenerated automatically** when the stamp no longer matches — for example after `--sync` (which invalidates the cache when `kb.qlf` / `profile.qlf` change), after editing `/etc/portage`, or when `USE` / `ACCEPT_KEYWORDS` / `ACCEPT_LICENSE` change in the environment.

Control per mode in `config.pl`:

```
config:preference_cache(standalone, cached).
config:preference_cache(daemon,     cached).
config:preference_cache(worker,     cached).
config:preference_cache(client,     live).
config:preference_cache(server,     cached).
```

Client mode keeps `live` rebuilding because preference facts are injected from the server in distributed proving.

3.9 World sets

portage-ng maintains world sets — the list of packages explicitly requested by the user — under `Source/Knowledge/Sets/world/`. Each machine can have its own `.local` world set file. The `@world` target resolves to all packages in the active world set. The format is the same as Gentoo's `/var/lib/portage/world`, so you can point portage-ng at your Gentoo system's world file and use Portage and portage-ng side by side.

World set management is handled through the `set.pl` module, which supports `world(Atom):register` and `world(Atom):unregister` proof literals to add/remove packages during `--merge` operations.

After you change world membership or sync new tree data, rely on the same **sync workflow** described above: a standalone `--sync` refreshes `Knowledge/kb.qlf` and the profile cache (and invalidates the preference cache) so resolution sees an up-to-date union of tree, VDB, and world-related facts.

3.10 Further reading

- Chapter 2: Installation and Quick Start — prerequisites and first run
- Chapter 6: Knowledge Base and Cache — how the Portage tree is loaded into Prolog facts

- Chapter 15: Command-Line Interface — CLI options that interact with configuration
- Chapter 20: Gentoo Linux Security Advisories (GLSA) — `Knowledge/glsa.qlf` built during `--sync`

Chapter 4

Architecture Overview

Reasoning about software configurations is not a single algorithm you "run once." It is a chain of transformations: turn repository facts into a logical problem, search for a proof that explains *why* each step is needed, turn that proof into an ordered plan, and finally render or execute it. portage-ng is structured as a **pipeline** because that sequence is the natural shape of the work. Each stage has a clean role—parse facts, resolve a configuration, order it, print (or build) it—and can be characterized in isolation: the reader produces a fixed vocabulary of literals; the resolver produces a justified configuration with its dependencies; the orderer refines that into something a human or a build system can follow. Treating the system as a pipeline is therefore a **design decision**: it keeps stages testable, replaceable, and easier to reason about than a monolith where parsing, search, ordering, and output are tangled together.

4.1 The pipeline

portage-ng processes a user request through a linear pipeline of five stages:

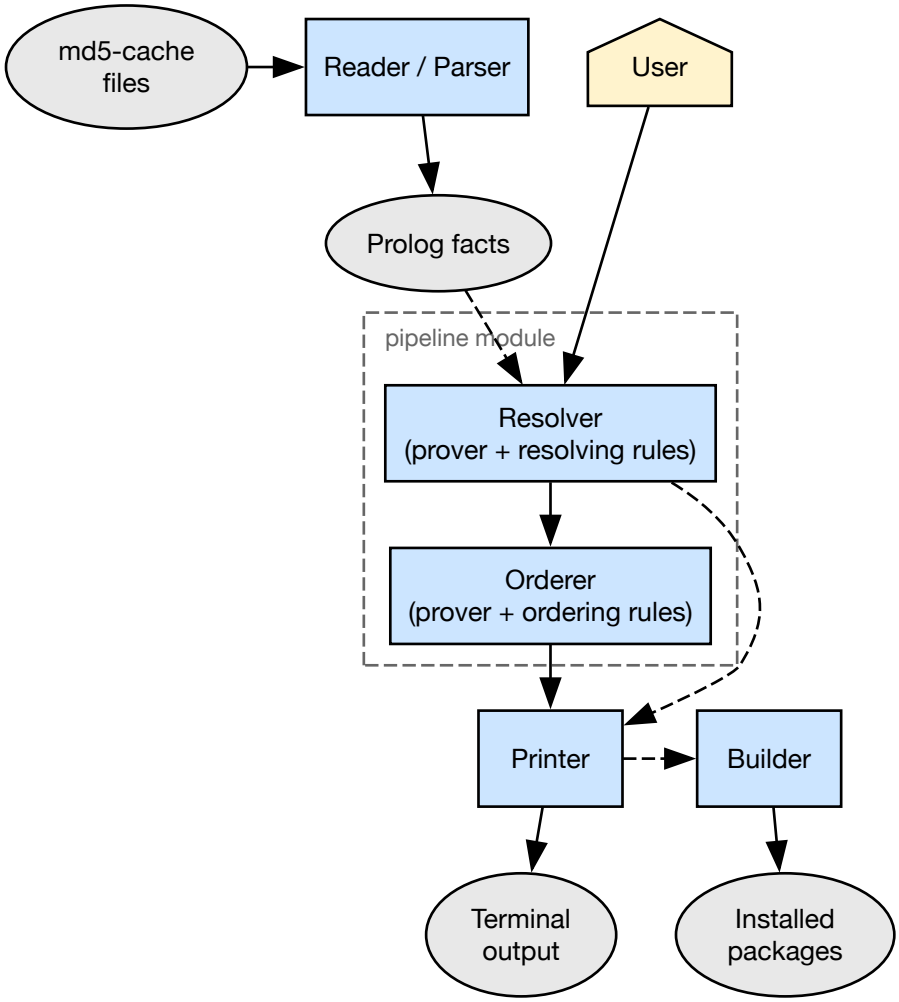


Figure 9 — Pipeline overview

```
reader/parser  →  resolver  →  orderer  →  printer  →  builder
                  └──────── pipeline ────────┘
                     (both passes run on the generic prover)
```

The resolver and orderer are thin stage wrappers around one generic proving engine, `prover.pl`. The prover knows nothing about resolving or ordering: callers hand it a rule module together with the goals (`prover:prove(Rules, ...)`). The resolver passes the `resolving` rule set (pass 1: *what* — versions, USE, slots); the orderer passes the `ordering` rule set (pass 2: *when* — waves).

The resolve pass produces four AVL trees — **Proof**, **Model**, **Constraints**, and **Triggers** — that flow through the rest of the pipeline. Together they capture *why* each literal was accepted, *what* is known, *what restrictions* must hold, and *who depends on whom*. Section Data structures describes each one in detail.

The resolver and orderer together form the `pipeline` module. Two canonical entry points share the same 5-tier committed-choice progressive relaxation (strict, keyword_acceptance, blockers, unmask, keyword_unmask):

```
pipeline:prove_plan_with_fallback(Goals, Proof, Model, Plan, Triggers)
pipeline:prove_with_fallback(Goals, Proof, Model, Triggers)
```

The first runs the full pipeline (resolve + order) and is used by all production paths. The second runs the resolve pass only and is used by layered tests and `--bugs`.

Stage	Module	Input	Output
Reader / Parser	`reader.pl`, `parser.pl`, `eapi.pl`	Ebuild md5-cache files	Prolog facts (`cache:entry/5`)
Resolver	`resolver.pl` → `prover.pl` + `resolving.pl`	Goal literals (from user)	Proof, Model, Constraints, Triggers
Orderer	`orderer.pl` → `prover.pl` + `ordering.pl`	Proof, Triggers	Ordering proof + wave-list Plan
Printer	`printer.pl`, `Printer/`	Proof, Model, Plan	Terminal output, `.merge` files
Builder	`builder.pl`, `Builder/`	Plan	Ebuild phase execution (via the ebuild contract)

4.2 Operating modes

portage-ng can run in several modes, each tailored to a different deployment scenario. The mode determines which modules are loaded, how the knowledge base is accessed, and whether proving happens locally or is distributed across machines. The mode is selected with `--mode` on the command line (e.g. `portage-ng --mode server`). When no mode is specified, standalone is used.

4.2.1 Standalone

The default and most common mode. A single process on a single machine loads the full knowledge base, runs the complete pipeline (resolver, orderer, printer, builder), and produces results locally. This is what you use for day-to-day `--pretend`, `--merge`, `--shell`, and `--sync`.

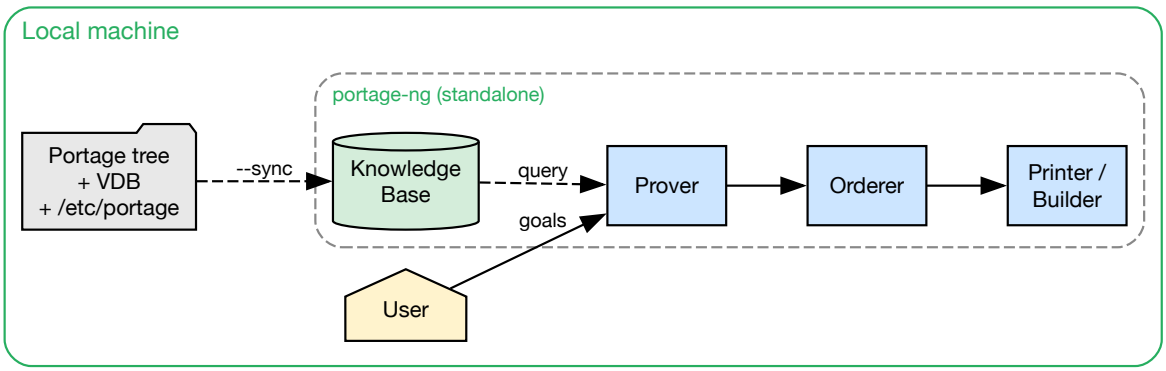


Figure 10 — Standalone mode

Everything happens in one process: the Portage tree, VDB, and `/etc/portage/` configuration are synced into the knowledge base, and the user's goal literals are proven, planned, and printed — all on the same machine.

4.2.2 Client and server

In client–server mode, the reasoning happens on a powerful server while a lightweight client submits requests and displays results. The client and server communicate over TCP/IP with SSL encryption (HTTPS), so they can run on different machines — potentially on different networks.

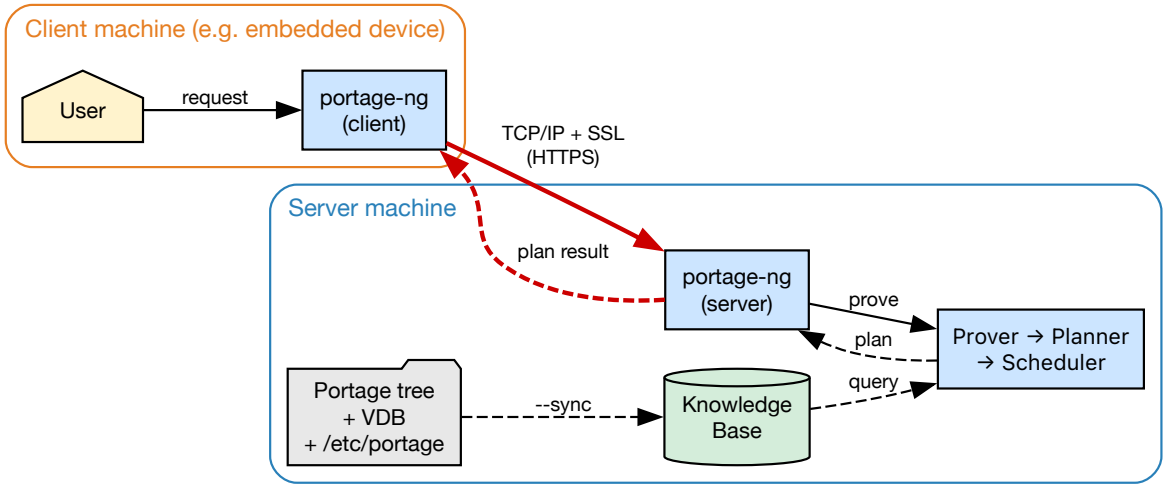


Figure 11 — Client–server mode

The server hosts the knowledge base and runs the full pipeline. The client needs only the thin slice of printing and pipeline glue required to render output. This makes client–server mode ideal for **embedded systems** and resource-constrained devices: the client binary is small, uses minimal memory, and delegates all proving to the server. Queries return in milliseconds because the knowledge base is already loaded and indexed on the server side.

4.2.3 Daemon / IPC

Daemon mode is similar to standalone, but the process stays resident and listens on a Unix socket for commands from local processes. Both the daemon and its clients run on the **same machine**.

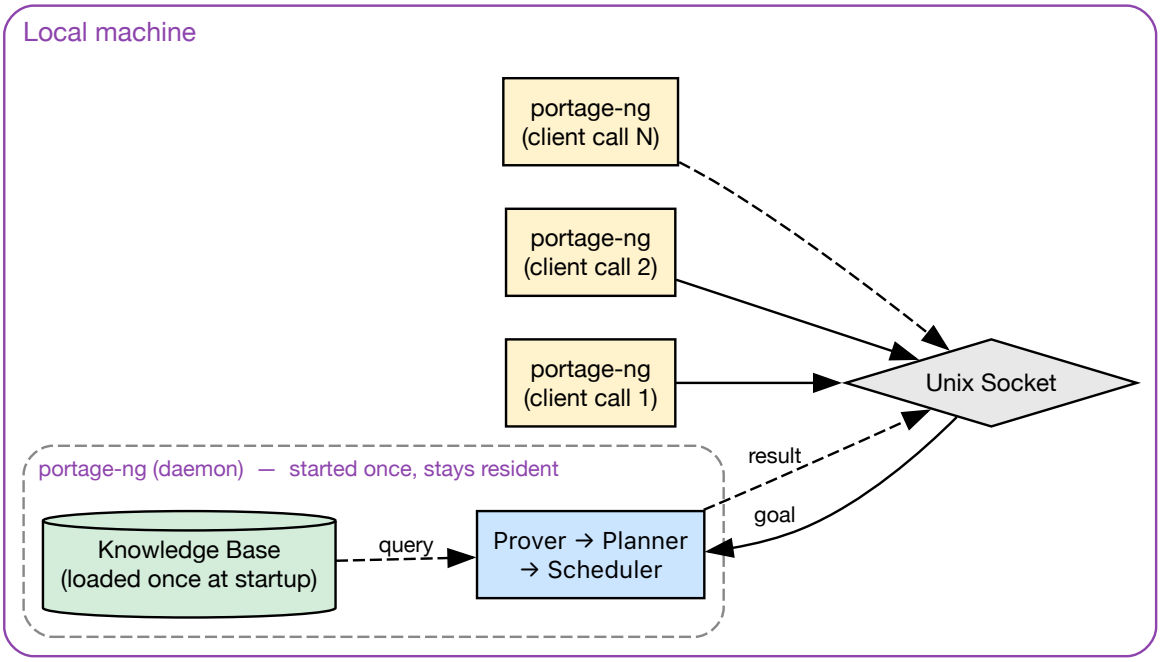


Figure 12 — Daemon / IPC mode

The key advantage is **startup performance**. In standalone mode, every invocation loads the full knowledge base from disk — tens of thousands of Prolog facts — before it can answer a single query. In daemon mode, the knowledge base is loaded **once** when the daemon starts and stays in memory. Subsequent queries arrive over the Unix socket and are answered in **milliseconds**, because there is no parsing, no qcompile loading, no JIT indexing warmup — just a direct query against the already-loaded, already-indexed knowledge base. This makes daemon mode well suited for interactive tooling, editor integrations, and scripts that issue many small queries in quick succession.

4.2.4 Workers

Worker mode enables **distributed proving** across multiple machines. A central server advertises itself via **Bonjour** (mDNS/DNS-SD), and workers on the local network automatically discover it without manual configuration.

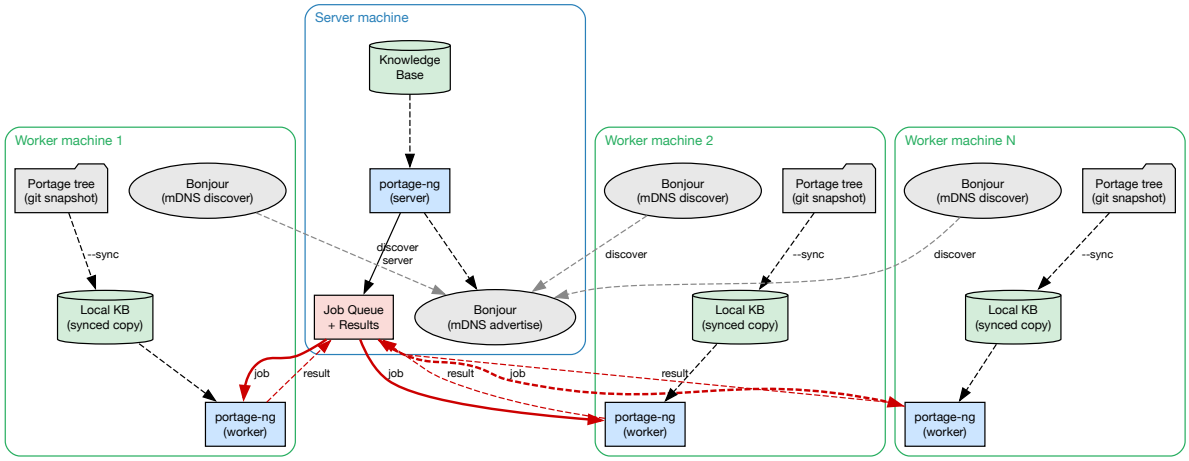


Figure 13 — Worker mode

Each worker machine maintains its own local copy of the Portage tree (typically via a **git snapshot**) and runs `--sync` locally to build its own knowledge base. This ensures all workers reason against the same set of ebuilds — tree synchronisation is a prerequisite for consistent results across the cluster.

Once a worker discovers the server, it polls the job queue for proving tasks: the server breaks a large proof (e.g. `@world`) into independent sub-goals, distributes them to available workers, and collects the results. Each worker runs the full pipeline locally (resolver, orderer), so proving scales horizontally — adding more worker machines reduces wall-clock time for large proof sets.

See Chapter 15: Command-Line Interface for the full mode reference and Chapter 18: Distributed Proving for TLS certificate setup and cluster configuration.

4.3 Module load order

Each mode loads only the modules it needs. This keeps startup time, memory footprint, and failure modes appropriate to the deployment:

The load order is defined in `Source/loader.pl`. `load_modules/1` loads a mode's groups from `loader:mode/2`, then the optional groups from `loader:optional/3` whose condition holds (the LLM groups, gated by `config:load_llm_modules/1`):

```
load_modules(common)       — SWI-Prolog libraries, OO context, config, OS,
                             interface, EAPI, reader, subprocess, bonjour,
                             feature unification
                             (loaded before mode dispatch)

load_modules(standalone)   — Full pipeline: KB (cache, repository, query),
                             Gentoo domain (version, resolving, ordering,
                             ebuild, VDB, preference, exceptions), prover,
                             resolver, orderer, printer, builder, grapher,
                             writer, test
                             (also loads LLM backends unless
```

```
                              config:load_llm_modules(false))

load_modules(server)       — Standalone groups plus HTTP server, Pengines,
                              sandbox

load_modules(client)       — HTTP/socket client, subset of printer/pipeline

load_modules(worker)       — Same pipeline as standalone + client + cluster

load_modules(daemon)       — Standalone groups plus the daemon accept loop

load_modules(ipc)          — Ultralight IPC client (no LLM)
```

4.4 Domain-agnostic core vs Gentoo-specific rules

Traditional Portage couples the resolver tightly to Gentoo semantics: USE flags, slots, profiles, and cache layout are not optional details—they are woven through the same code paths as the search strategy. That makes it hard to test "the resolver" without dragging the entire domain along, and hard to experiment with alternative rule sets or other package ecosystems.

portage-ng deliberately **separates** a domain-agnostic core from a Gentoo-specific rules layer. The prover does not know what a USE flag *is*. It sees abstract literals and Horn-style rules; expanding a goal means calling a single hook-shaped interface and continuing the search. The same engine could, in principle, reason about RPM packages, Nix derivations, or Cargo crates—you would supply different `rule/2` implementations and a different knowledge base, not a different prover. That separation is intentional: it isolates *how we search* from *what Gentoo means*, so the core can be exercised and compared without re-implementing Portage wholesale. Packages, USE flags, and slots never appear as primitives in the core; they are interpreted entirely inside the rules layer:

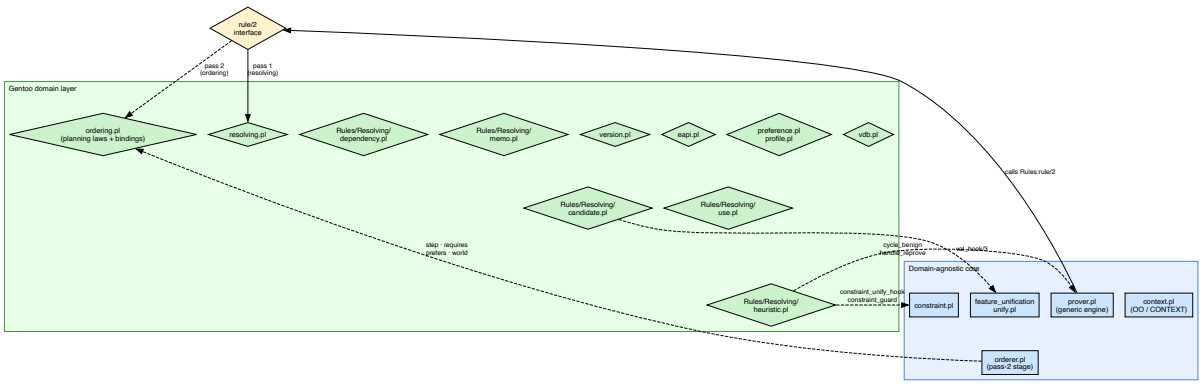


Figure 14 — Layer separation

The **`rule/2` interface** is the contract between the domain-agnostic core and the domain-specific layer. Everything Gentoo-specific—consulting the knowledge base, evaluating USE conditionals, resolving candidates, emitting constraint terms—lives on the far side of that boundary.

```
resolving:rule(Head, Body)
```

The prover calls `rule/2` to expand a literal into its dependencies. The rules module implements this by consulting the knowledge base, evaluating USE conditionals, resolving candidates, and emitting constraint terms.

This separation means the same reasoning engine could be applied to a different domain by supplying a different set of rules.

4.5 Data structures

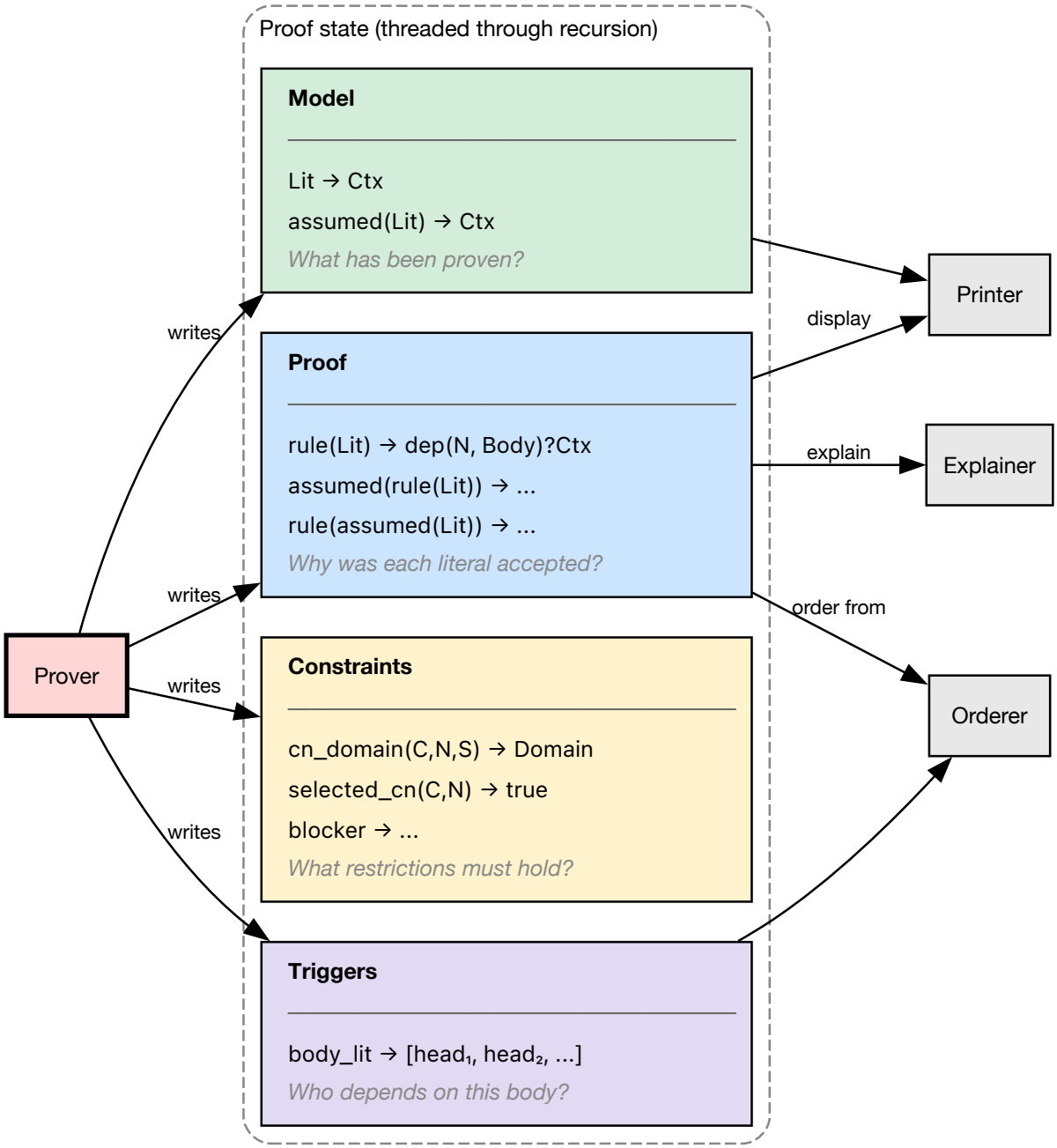


Figure 15 — Data structures

During proof search, the prover must answer four kinds of question at once: *why* was this literal accepted, *what* is already known, *what restrictions* must remain consistent across branches, and *who depends on whom* when context or assumptions change. Four balanced trees (AVL maps via `library(assoc)`) hold exactly those roles. Together they capture the **complete state** of a proof attempt: the prover threads them through recursive expansion without relying on a soup of unrelated global mutable flags for "current model" or "current explanation."

- **Proof** — Records *why* each literal was proven: the justification (which rule instance and body linked the head). Without it, you cannot explain the plan or reconstruct the dependency argument for the user.
- **Model** — Records *what* has been proven: the current state of knowledge (each literal and its proof-term context). This is the structure that memoizes success: the same literal is not re-proved from scratch along every path.
- **Constraints** — Records *restrictions* that must hold: version domains, slot locks, blockers, and similar invariants. They cross-cut the proof tree; they are not local to a single rule application.
- **Triggers** — Records *which heads depend on which bodies*—a reverse-dependency index. When a context changes or delayed work fires, the prover uses triggers to find "who cares" about that body without scanning the entire proof.

The prover maintains these four structures during proof construction:

Structure	Key	Value	Purpose
Proof	`rule(Lit)` or `assumed(rule(Lit))`	`dep(N, Body)?Ctx`	Which rule and body justified each literal; `N` is the dependency count, `Ctx` the proof context
Model	`Lit` or `assumed(Lit)`	`Ctx`	Every literal that has been established, mapped to the context under which it was proven
Constraints	e.g. `cn_domain(dev-libs, openssl, 0)`	`version_domain(...)`	Accumulated invariants: version domains, slot locks (`slot(3)`), blockers
Triggers	body literal	`[head, ...]`	Reverse-dependency index: which heads depend on this body literal

The Proof and Model structures use different key schemes to distinguish normal proofs from assumptions:

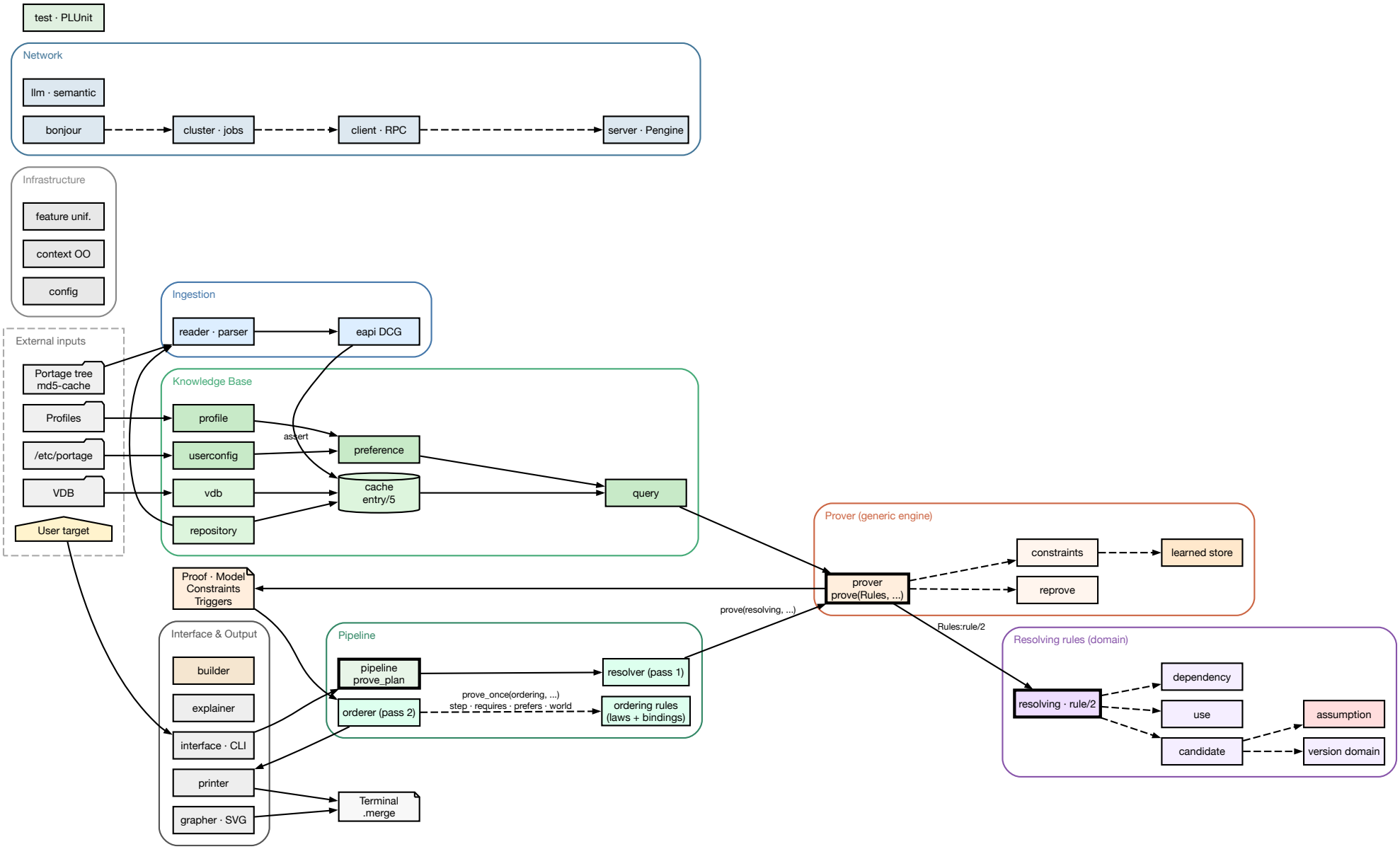
- `rule(Lit)` — normally proven literal
- `assumed(rule(Lit))` — prover cycle-break assumption
- `rule(assumed(Lit))` — domain assumption (dependency cannot be satisfied)

See Chapter 5: Proof Literals for the literal format and Chapter 9: Assumptions for the assumption taxonomy.

4.6 Architecture diagram

The following page shows the full system architecture in landscape orientation, covering all layers from external inputs through the knowledge base, prover, orderer, and output pipeline.

portage-ng: Full System Architecture


test · PLUnit
Network
llm · semantic
bonjour
cluster · jobs
client · RPC
server · Pengine
Infrastructure
feature unif.
context OO
config
Ingestion
reader · parser
eapi DCG
External inputs
Portage tree
md5-cache
Profiles
/etc/portage
VDB
User target
Knowledge Base
profile
userconfig
vdb
repository
assert
preference
cache
entry/5
query
Prover (generic engine)
constraints
learned store
prover
prove(Rules, ...)
reprove
prove(resolving, ...)
Rules:rule/2
Proof · Model
Constraints
Triggers
Interface & Output
builder
explainer
interface · CLI
printer
grapher · SVG
Terminal
.merge
Pipeline
pipeline
prove_plan
resolver (pass 1)
orderer (pass 2)
prove_once(ordering, ...)
step · requires · prefers · world
ordering rules
(laws + bindings)
Resolving rules (domain)
resolving · rule/2
dependency
use
candidate
assumption
version domain

4.7 Further reading

- Chapter 5: Proof Literals — the `Repo://Entry:Action?{Context}` term format
- Chapter 8: The Prover — inductive proof search in detail
- Chapter 11: Rules and Domain Logic — how rule modules plug into the prover
- Chapter 12: Resolution — Configuration as Proofs — pass 1: justified configuration
- Chapter 13: Ordering — Plans as Proofs — the second proving pass and wave projection

Chapter 5

Proof Literals

5.1 The universal literal format

Every term that flows through the portage-ng pipeline — from rules to prover to orderer to printer — uses the same universal format:

```
Repo://Entry:Action?{Context}
```

Each component answers a question that arises at a different stage of the pipeline:

- **`Repo`** — *where* does this fact come from? The **rules** consult different repositories (the Portage tree, the VDB, an overlay) and the repository prefix travels with the literal so the prover never confuses an available package with an installed one.
- **`Entry`** — *what* package version is meant? When the rules expand a dependency, they select a concrete cache entry (`'category/name-version'`). This identifier is the key the **prover** uses to look up and store proof work — two dependency paths that resolve to the same entry share the same proof node.
- **`Action`** — *how* should the pipeline treat this entry? The rules assign an action (`:install`, `:run`, `:download`, `:update`, …) that tells the **orderer** which phase of work this literal represents and how to order it relative to others.
- **`Context`** — *why* and *under what conditions* was this literal introduced? As the prover expands the dependency graph, each literal accumulates a feature-term context: which parent introduced it (`self`), which USE flags are required (`build_with_use`), ordering markers (`after`), slot locks, and so on. At join points where two dependency paths reach the same literal, the prover **merges** their contexts via feature unification. The **printer** reads the final context to display USE flags, slot information, and assumption reasons.

Traditional resolvers scatter this information across separate side structures. portage-ng packs it into the literal itself, making every term **self-describing**: you can inspect a single literal and know its repository, version, phase, and full provenance without consulting external tables.

5.2 Operator precedences

The literal format is defined by three infix operators declared in `Source/Logic/context.pl`. In SWI-Prolog, **higher** precedence means the operator becomes the **principal functor** at that level of the term — i.e. it sits **higher** in the parse tree. The ordering `://` (603) > `?` (602) > `:` (601) was chosen so that the structure lines up with everyday use: you scope by **repository** first,

then attach the **context list** to the **ebuild core** (`Entry:Action`), with **entry** and **action** paired at the innermost level. That makes the common cases — "everything in `portage`", or "this `category/name-version` with this phase" — parse in the way you read them. The `?{Context}` annotation is intentionally the outer wrapper around the core (after `://`) because **context is what changes most often during proof search**; the repository and entry/action spine stay stable while USE, ordering, and constraint features are merged and refined.

Operator	Precedence	Associativity	Parses as
`://`	603	xfx	`Repo :// Rest`
`?`	602	xfx	`Core ? {Context}`
`:`	601	xfx	`Entry : Action`

Because `://` has the highest precedence, a full literal parses as:

```
Repo :// ((Entry : Action) ? {Context})
```

That is: repository scopes the whole term; the ebuild core is `Entry : Action`; the context list attaches to that core.

5.3 `Repo` — the repository

The leftmost component identifies which registered repository the literal belongs to. It is an atom — the same atom used when registering the repository with the knowledge base:

```
:- portage:newinstance(repository).
:- kb:register(portage).
```

Common repository atoms:

Atom	Meaning
`portage`	The main Gentoo Portage tree
`pkg`	The VDB (installed packages database)
`overlay`	A user or test overlay

Literals from different repositories can coexist in the same proof. For example, a `portage://...` literal might depend on a `pkg://...` literal when an installed package satisfies a dependency.

5.4 `Entry` — the cache entry

The middle component is the cache entry identifier — a quoted atom in the format `'category/name-version'`:

```
'sys-apps/portage-3.0.77-r3'
'dev-lang/python-3.13.2'
```

This atom maps directly to the second argument of `cache:entry/5`:

```
cache:entry(portage, 'sys-apps/portage-3.0.77-r3', 'sys-apps', 'portage',
            version([3,0,77],'',...)).
```

The category, name, and version are also available as separate fields in the cache, but the combined atom serves as the unique key for lookup.

5.5 `Action` — the phase

The component after the entry (inside the `Entry : Action` pair) specifies what operation the literal represents. Actions fall into three categories:

5.5.1 Ebuild actions

These apply to `Repo://Entry` literals:

Action	Meaning
`run`	Full installation + runtime availability
`install`	Build and install (DEPEND + BDEPEND + RDEPEND)
`download`	Fetch source archives
`fetchonly`	Legacy fetch-only ebuild action (overlay / cycle tests). CLI `--fetchonly` does **not** prove this; it proves `:run` and filters print/execute.
`reinstall`	Reinstall an already-installed package
`update`	Update to a newer version
`downgrade`	Downgrade to an older version
`upgrade`	Upgrade (used in VDB context)
`depclean`	Remove an unneeded package
`uninstall`	Uninstall a package

5.5.2 Dependency and validation actions

Before the prover can expand a package's full dependency tree, it first needs to answer two questions: *which dependencies are actually active* given the package's USE flags, and *is the USE configuration itself consistent*? These questions are answered in two dedicated phases — `:config` and `:validate` — that run **before** the main `:install` / `:run` expansion.

The `:config` phase — computing the dependency model

When portage-ng resolves a package, it does not immediately try to prove every dependency listed in the ebuild's metadata. Instead, it first builds a **dependency model**: a stable snapshot of which dependencies are active under the current USE flag configuration.

An ebuild's metadata contains conditional dependencies guarded by USE flags. For example, `dev-lang/python` might declare:

```
RDEPEND="ssl? ( dev-libs/openssl )
         readline? ( sys-libs/readline )
         !readline? ( sys-libs/libedit )"
```

The `:config` phase evaluates each dependency term against the effective USE flags and retains only the **active** dependencies. In the example above, if `ssl` is enabled and `readline` is disabled, the model will contain `dev-libs/openssl` and `sys-libs/libedit` — the `sys-libs/readline` dependency is dropped because its USE guard is not satisfied. Same-slot self-references (a package listing itself as a build dependency in the same slot) are treated as bootstrap dependencies and checked against the VDB. Cross-slot self-references (same category and name but a different slot) are resolved normally as regular dependencies.

When a choice group or constraint forces a decision, the prover may also **assume** a flag — for instance, if an `exactly_one_of` group requires at least one member to be enabled and none currently is, the prover picks the most likely candidate and records a domain assumption so the user is informed.

The result is a **model** whose keys are the surviving dependency terms — the ones that actually need resolving.

For choice groups (OR dependencies), the `:config` phase picks one viable alternative:

```
RDEPEND="|| ( dev-db/postgresql dev-db/mariadb dev-db/sqlite )"
```

becomes a `choice_group(Deps):config?{Context}` literal. The rules try each alternative, preferring already-installed packages, and commit to one choice. The chosen dependency enters the model; the others are discarded. This means that by the time the main proof begins, every OR group has been resolved to a single concrete dependency.

This commit is always final (a cut after the first viable alternative). Variant exploration does not remove that cut: it **re-runs the prover** with thread-local branch-preference overrides (`variant:use_override`, `variant:branch_prefer`) that reorder the candidates, so the same cut selects a different branch on the re-proof — see Multiple stable models.

Action	Literal head	Meaning
`config`	grouped dependency	Resolve a dependency group under USE flags
`config`	package dependency	Check a single dependency
`config`	choice group	Pick one alternative from an OR group
`config`	USE conditional	Evaluate a USE-guarded block

The `:validate` phase — checking REQUIRED_USE consistency

Ebuilds can declare constraints on which USE flag combinations are valid. For example:

```
REQUIRED_USE="^^ ( python_targets_python3_12 python_targets_python3_13 )"
```

This says "exactly one Python target must be selected." The `^^` operator translates to an `exactly_one_of_group(...)` term. Before expanding the package's dependencies, portage-ng wraps each REQUIRED_USE constraint as a `:validate` literal:

```
exactly_one_of_group([required(python_targets_python3_12),
                      required(python_targets_python3_13)]):validate?{[
  self(portage://'dev-lang/python-3.13.2')
]}
```

The rules check whether the effective USE flags for the package (identified by the `self(...)` context tag) satisfy the constraint. For `exactly_one_of`, the check counts how many of the listed flags are enabled and verifies the count is exactly one. If the constraint is violated, the rules emit a domain assumption recording the conflict:

```
assumed(conflict(required_use, exactly_one_of_group(Deps)))
```

The full set of REQUIRED_USE operators:

Operator	Group term	Constraint
`^^`	`exactly_one_of_group(Deps)`	Exactly one flag enabled
any-of	`any_of_group(Deps)`	At least one flag enabled
`??`	`at_most_one_of_group(Deps)`	At most one flag enabled
(none)	`use_conditional_group(...)`	Conditional: if A then B

Each of these operators is wrapped as a `:validate` literal and checked against the package's effective USE flags:

Action	Literal head	Meaning
`validate`	`exactly_one_of_group(...)`	Check `^^` constraint
`validate`	`any_of_group(...)`	Check any-of constraint
`validate`	`at_most_one_of_group(...)`	Check `??` constraint

5.5.3 Non-ebuild literal heads

Some literals do not follow the `Repo://Entry` pattern:

Literal	Meaning
`world(Atom):register`	Add a package to the @world set
`world(Atom):unregister`	Remove a package from the @world set
`target(Query, Arg):run`	Top-level target resolution (`--merge`, `--fetchonly`, `--build`)
`target(Query, Arg):fetchonly`	Legacy fetch-only target (not used by CLI `--fetchonly`)
`target(Query, Arg):uninstall`	Top-level uninstall target

5.6 `Context` — the feature-term list

The context is a Prolog list wrapped in `{}` and attached via the `?` operator. It carries per-literal metadata that records provenance, ordering, constraints, and USE requirements:

```
portage://'dev-lang/python-3.13.2':install?{[
  self(portage://'sys-apps/portage-3.0.77-r3'),
  build_with_use:use_state([ssl, threads], []),
  after(portage://'sys-apps/portage-3.0.77-r3':install)
]}
```

Reading this literal: "install `dev-lang/python-3.13.2` from the portage repository, because `sys-apps/portage` needs it (`self`), with USE flags `ssl` and `threads` enabled (`build_with_use`), and schedule it after the installation of `sys-apps/portage` (`after`)."

The context list is **not** an unstructured bag of annotations. It is the proof-side counterpart of **feature terms** in the sense used by Zeller-style feature logic (see Chapter 22: Context Terms): a structured collection of features that can be **merged** when two dependency paths describe the same package under different conditions. When two paths reach the same literal with different USE requirements or other features, the prover does not arbitrarily pick one path's context — it **combines** them using feature term unification (`sampler:ctx_union/3`), which relies on the same feature machinery as the rest of the context subsystem. That is why context lives in a dedicated suffix of the literal: it is the part that must stay open to **merge** and **refine** as the proof graph grows.

5.6.1 `self` — who introduced this dependency

Every dependency in an ebuild comes from somewhere. The `self(Repo://Entry)` tag records *which package* introduced this literal as a dependency. When the rules expand a package's dependency list, they stamp every child literal with the parent's identity:

```
portage://'dev-libs/openssl-3.4.1':install?{[
  self(portage://'dev-lang/python-3.13.2')
]}
```

This says "openssl is here because python depends on it." The `self` tag serves three purposes:

1. **Provenance tracking.** The printer can show *why* a package appears in the plan — who pulled it in.
2. **USE flag resolution.** When checking whether a USE flag is enabled for a dependency, the rules look up the effective USE flags of the ebuild identified by `self`. This is how the `:validate` phase works: the `self` tag tells the REQUIRED_USE checker which package's USE configuration to consult.
3. **Self-dependency detection.** When a package lists itself as a build dependency in the same slot (which happens for bootstrap packages), the rules recognise this by comparing the dependency target's category, name, and slot to the `self` entry. Same-slot self-deps are checked against installed packages; cross-slot self-deps (e.g. `antlr-tool:4` depending on `antlr-tool:3.5`) are treated as regular dependencies and resolved normally.

At most one `self` tag is present per context. When a literal is stamped, any previous `self` is replaced — the immediate parent is what matters.

5.6.2 `build_with_use` — requirements imposed by parent

Gentoo dependency atoms can carry *bracketed USE requirements*: conditions that must hold on the dependency target. For example, in `sys-apps/portage`'s metadata:

```
RDEPEND="dev-lang/python[ssl,threads]"
```

The brackets `[ssl,threads]` mean "I need python, and it must be built with the `ssl` and `threads` USE flags enabled." The rules translate this into a `build_with_use` context tag:

```
portage://'dev-lang/python-3.13.2':install?{[
  build_with_use:use_state([ssl, threads], [])
]}
```

The `use_state(Enabled, Disabled)` term lists which flags must be on and which must be off. Negative requirements like `[-test]` appear in the disabled list:

```
build_with_use:use_state([], [test])
```

When two dependency paths reach the same package with different USE requirements, the prover merges them via feature unification. If portage requires `python[ssl]` and another package requires `python[xml]`, the merged context becomes:

```
build_with_use:use_state([ssl, xml], [])
```

If two paths disagree — one requires `[debug]` and another requires `[-debug]` — the unification detects the conflict and the constraint system handles it (potentially triggering a reprove with different candidate selection).

The `build_with_use` tag is distinct from the package's own USE flags. A package's USE flags are determined by profile, user configuration, and defaults. The `build_with_use` tag captures

what *other packages demand of this package*. The printer reads both to display the final USE flag set, marking flags that were pulled in by dependency requirements.

5.6.3 `after` — ordering markers

The ordering pass needs to know the order in which actions should be scheduled. The `after(Literal)` tag expresses a hard ordering requirement: "this literal must come after the specified literal in the final plan."

```
portage://'dev-lang/python-3.13.2':download?{[
  after(portage://'sys-apps/portage-3.0.77-r3':install)
]}
```

Ordering requirements arise naturally from the dependency structure. When package A depends on package B, the rules add `after(B:install)` to A's download and dependency contexts. This ensures that B is installed before A starts building.

The `after` tag **propagates**: when it is set on a literal, it is also injected into that literal's own children. If A must come after B, then A's dependencies also implicitly come after B. This transitive propagation ensures that entire subtrees are correctly ordered.

For cases where ordering should *not* propagate, the `after_only` variant exists. This is used primarily for PDEPEND (post-dependencies): a package's post-dependencies must come after the package itself, but the post-dependency's own children should not inherit that ordering requirement.

```
after_only(portage://'app-editors/neovim-0.12.0':run)
```

The ordering bindings (`Source/Domain/Gentoo/Rules/ordering.pl`) read both `after` and `after_only` from every literal's context to derive the hard requirements (`requires/2`) and the soft requirements — preferences (`prefers/2`) — that drive the second proving pass (Chapter 13). Neither is a constraint in the constraint-store sense (Chapters 8 and 9); the markers only steer ordering.

5.6.4 Summary of context tags

Tag	Purpose
`self(Repo://Entry)`	The parent ebuild that introduced this dependency
`build_with_use:use_state(En, Dis)`	USE flags that must be enabled/disabled on this package
`after(Literal)`	Must come after this literal; propagates to children
`after_only(Literal)`	Must come after this literal; does not propagate
`slot(C, N, Ss):{Candidate}`	Slot lock from `:=` sub-slot rebuild semantics
`replaces(pkg://Entry)`	Which installed package this action replaces
`assumption_reason(Reason)`	Why a domain assumption was made
`suggestion(Type, Detail)`	Actionable suggestion (keyword, unmask, use change)
`constraint(cn_domain(C,N):{D})`	Inline version domain constraint
`onlydeps_target`	Marks a literal as an `--onlydeps` target
`world_atom(Atom)`	Planning marker for @world set membership

Contexts are merged at join points via feature term unification, which uses Zeller-inspired feature unification. See Chapter 22: Context Terms for full details.

5.7 Canonical decomposition

The prover stores literals in assoc/AVL structures keyed by a **stable** identity. That creates a design tension: during proof search, the **context** is constantly enriched — new `build_with_use` features appear, ordering constraints are propagated, slot locks and learned domains are attached — but the **underlying package and phase** (repository, cache entry, action) are still the same logical goal. If the full term including context were used as the key, every refinement would look like a **new** node: you would get duplicate entries for "the same" install step, incoherent merging, and broken sharing of proof work.

Canonical decomposition fixes that by splitting each literal into a **core** used for identity (`R://L:A`) and a **context list** carried as associated data. Two encounters of the same core with different contexts collide on the same key; the prover then **merges** contexts (via feature term unification) instead of forking duplicate keys.

Two predicates handle decomposition:

5.7.1 `prover:canon_literal/3`

Strips the context from a literal, returning the core key and context separately:

```
canon_literal(R://(L:A),           R://L:A, {}).
canon_literal(R://(L:A?{Ctx}),     R://L:A, Ctx).
```

```
canon_literal(R://(L:A)?{Ctx},      R://L:A, Ctx).
canon_literal(R://(L:A?{C1})?{C2},  R://L:A, Merged).
```

The core `R://L:A` is used as the key in the Model AVL. The context is stored as the value.

5.7.2 `prover:canon_rule/3`

Similarly decomposes a rule head, producing a context-free key for the Proof AVL.

This decomposition ensures that when a literal is re-encountered with a different context, the prover can find the existing proof entry and merge the contexts rather than creating a duplicate.

5.8 How literals flow through the pipeline

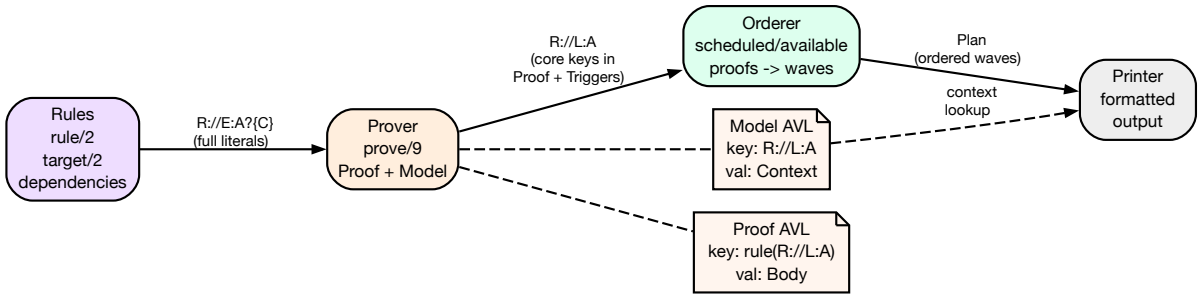


Figure 16 — Literal flow through the pipeline

1. **Rules** produce literals. The `target/2` rule resolves a user query to a `Repo://Entry:run?{Context}` literal. Dependency rules produce further literals with appropriate actions and contexts.
2. **Prover** stores the core literal (`R://L:A`) as the key in the Model AVL and the context as the value. The Proof AVL uses `rule(R://L:A)` as the key, with the rule body and context as the value.
3. **Orderer** treats each rule head in the Proof AVL as a plan step and proves `scheduled/1` for it through the planning laws; wave numbers are projected from the resulting availability proofs.
4. **Printer** reads the Plan (a list of waves), looks up each literal in the Model AVL to recover its context, and formats the output.

5.9 Worked example

Tracing `target('sys-apps/portage'):run?{[]}` through the pipeline (term shapes simplified: the real goal wraps a `qualified_target/6` term parsed from the CLI argument, and the non-oneshot run-target rule also adds a `world(...):register` condition):

```
1. User runs: portage-ng --pretend sys-apps/portage
```

```
2. Interface creates goal literal:
   [target('sys-apps'-'portage', []):run?{[]}]

3. Interface invokes the prover with this goal.

4. Prover uses rules to expand target/2:
   rule(target('sys-apps'-'portage', []):run?{[]},
        [portage://'sys-apps/portage-3.0.77-r3':run?{[]}])

5. Prover uses rules to expand :run:
   rule(portage://'sys-apps/portage-3.0.77-r3':run?{[]},
        [portage://'sys-apps/portage-3.0.77-r3':install?{[]},
         ...RDEPEND literals...])

6. Prover uses rules to expand :install:
   rule(portage://'sys-apps/portage-3.0.77-r3':install?{[]},
        [portage://'sys-apps/portage-3.0.77-r3':download?{[]},
         ...DEPEND/BDEPEND literals with self/1, build_with_use, after...])

7. Prover stores each proven literal in the Model AVL:
   Key: portage://'sys-apps/portage-3.0.77-r3':run
   Val: [] (context)

8. Orderer places :download in wave 1, :install in wave 2, :run in wave 3

9. Printer outputs:
   [1] portage://sys-apps/portage-3.0.77-r3  download
   [2] portage://sys-apps/portage-3.0.77-r3  install
   [3] portage://sys-apps/portage-3.0.77-r3  run
```

5.10 Further reading

- Chapter 8: The Prover — how the prover uses these literals
- Chapter 11: Rules and Domain Logic — how rules produce literals
- Chapter 22: Context Terms — deep dive into context semantics and feature unification

Chapter 6

Knowledge Base and Cache

6.1 From ebuild to fact: the metadata pipeline

Every package in Gentoo begins life as an **ebuild**: a bash script in the Portage tree that declares metadata (dependencies, USE flags, slot, license, and more). Portage-ng does not run those scripts. Instead, it consumes a pre-digested form of the same information.

`egencache` (or portage-ng's `--regen`) walks the tree and turns ebuilds into **md5-cache** files: flat key–value text blobs that summarize each ebuild's metadata. Portage-ng's reader and parser then loads those files, runs their contents through the **EAPI DCG grammar** (see Chapter 7), and **asserts** `cache:entry/5` (and related) facts into the in-memory knowledge base. For fast startup, those facts are **qcompiled** into `Knowledge/kb.qlf`, so the next session reloads binary bytecode instead of reparsing thousands of text files.

That end-to-end path — ebuild → cache generation → md5-cache → grammar → Prolog facts → QLF — is the **metadata pipeline**. It is deliberate: portage-ng never executes bash for package metadata; it works **entirely from metadata** that has already been extracted and normalized.

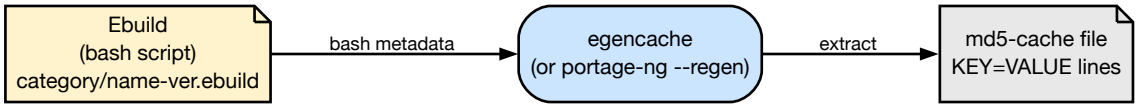


Figure 17 — Cache generation: ebuild to md5-cache

Once the md5-cache files exist on disk, portage-ng's reader parses each one through the EAPI grammar and asserts the resulting terms as Prolog facts:

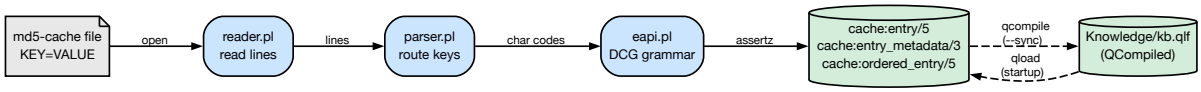


Figure 18 — Ingestion: md5-cache to Prolog facts

The knowledge base is the in-memory representation of the Gentoo Portage tree at the end of that pipeline. It stores every ebuild's metadata as Prolog facts that can be queried in sub-millisecond time.

6.2 The cache data structure

The core data structure is `cache:entry/5`, a dynamic predicate with one fact per ebuild:

```
cache:entry(Repository, Id, Category, Name, Version).
```

Argument	Example	Meaning
`Repository`	`portage`	Registered repository atom
`Id`	`'sys-apps/portage-3.0.77-r3'`	Full category/name-version string
`Category`	`'sys-apps'`	Package category
`Name`	`'portage'`	Package name
`Version`	`version([3,0,77],'',...)`	Parsed version as `version/7` term

Additional cache predicates store per-ebuild metadata:

Predicate	Content
`cache:entry_metadata/4`	Per-entry key/value metadata: EAPI, SLOT, KEYWORDS, LICENSE, etc.
`cache:ordered_entry/5`	Entries ordered by version (for candidate selection)
`cache:manifest/5`, `cache:manifest_metadata/6`	Manifest files and their checksums

6.3 Why this cache design?

The shape of `cache:entry/5` is not arbitrary; it matches how Prolog and the prover actually use the data.

Indexing and lookup. Prolog's strength is pattern matching on structured terms. `cache:entry/5` is arranged so that **first-argument indexing** on `Repository` and **third- and fourth-argument** access on `Category` and `Name` align with the most common query shapes: "in this repo, what entries exist for this category/name?"

Versions as terms. The version is stored as a **`version/7` compound term** so that **standard `compare/3`** on the term gives correct version ordering **without any runtime conversion** to another representation. The prover and rules can treat versions as ordinary Prolog data.

Splitting metadata. Rich fields (EAPI, SLOT, KEYWORDS, LICENSE, …) live in **`cache:entry_metadata/3`** rather than bloating every `entry/5` fact. That keeps the **hot path** — finding candidates by repository, category, and name — **lightweight**, while still allowing full metadata when needed.

Pre-sorted candidates and version preference. **`cache:ordered_entry/5`** holds entries **pre-sorted by version (newest first)** for candidate selection. Building that structure once at load or regen time **avoids repeated sorting during proof search**, where the same name may be considered many times under different contexts.

The ordering is more than an optimisation — it encodes **preference**. Prolog iterates over `ordered_entry/5` clauses in assertion order, so the newest version is tried first. When the prover searches for a candidate that satisfies a dependency, it encounters the highest version before any older alternative. If that candidate passes all constraint guards, it is selected without ever considering older versions. If it fails (wrong slot, masked, REQUIRED_USE violation), Prolog's backtracking moves to the next clause — the next-highest version — automatically.

This design has a formal counterpart in **ordered logic programs** as studied by Vermeir and Van Nieuwenborgh ("Preferred Answer Sets for Ordered Logic Programs," JELIA 2002). In their framework, when multiple rules can derive conflicting conclusions, a **partial order over rules** determines which one prevails. In portage-ng the "rules" are candidate versions for a given category/name, and the partial order is the version comparison: newer versions have higher priority. Prolog's clause order directly implements this priority — no separate preference layer or scoring function is needed. The result is that the prover naturally gravitates toward the newest compatible version, which matches Gentoo's standard policy, while still falling back to older versions when constraints demand it.

See Chapter 23: Resolver Comparison for more on Vermeir's ordered logic and its role alongside Zeller's feature logic and CDCL-style conflict learning.

6.4 Repositories and the knowledge base registry

Repositories are not just atoms: they are **objects** in the OO context system (`Source/Logic/context.pl`). Each repository has its own **identity**, **paths**, **sync method**, and **cache** partition. The **context** machinery provides **instance creation** (`newinstance`), **method dispatch**, and **visibility guards**, so different repository kinds can share an interface while differing in behavior — for example, `portage:sync` and `overlay:sync` can implement sync differently behind the same method.

Repositories are registered via that OO context system. Each repository is an instance of the `repository` class:

```
:- portage:newinstance(repository).
:- kb:register(portage).
```

The knowledge base module (`knowledgebase.pl`) maintains a registry of all loaded repositories. **`kb:register/1`** records which repositories are **active** so the rest of the system can iterate or dispatch over them. Multiple repositories can be registered simultaneously — for example, the main Portage tree (`portage`), the VDB of installed packages (`pkg`), and user overlays (`overlay`).

Each repository instance manages its own cache facts. The `repository` class provides methods for syncing, loading, and querying:

```
portage:sync.                  % Sync from remote + regenerate caches
portage:update_cache.          % Re-read md5-cache into Prolog facts
portage:query(Query, Result).  % Query entries (delegates to query:search/2)
```

6.5 Syncing and cache regeneration

`--sync` performs a full repository synchronization. It is the "wide" end of the metadata pipeline: it brings the tree up to date, then materializes fresh cache facts and QLF artifacts.

1. Fetches the latest Portage tree (via git, rsync, or HTTP)
2. Reads md5-cache files via the EAPI grammar into cache predicates
3. Generates `Knowledge/kb.qlf` (qcompiled facts for fast reload)
4. Generates `Knowledge/profile.qlf` (serialized profile data)
5. Invalidates `Knowledge/preference.qlf` (materialized preference cache; rebuilt on next startup)

`--regen` regenerates the md5-cache incrementally. It replaces `egencache`: only changed or new ebuilds are re-parsed, and regeneration runs in parallel across available cores.

6.6 Compiling knowledge

On subsequent startups, portage-ng loads `Knowledge/kb.qlf` instead of re-parsing the entire md5-cache directory. `qcompile` files are a SWI-Prolog binary format that loads an order of magnitude faster than parsing text files. That step closes the pipeline opened by ebuilds and md5-cache: the **authoritative** working set for proving is the compiled fact base, not the shell sources.

The raw Prolog facts are also available as `Knowledge/kb.raw` for debugging.

When `config:preference_cache/2` is set to `cached` for the active mode, the first `preference:init/0` after startup (or after invalidation) writes `Knowledge/preference.qlf` and a companion `Knowledge/preference.stamp`. Subsequent starts reload the materialized preference state in milliseconds while the stamp matches (see Chapter 3: Configuration).

6.7 Query layer

The cache facts described above — `cache:ordered_entry/5`, `cache:entry_metadata/4`, and friends — are ground relational tuples: a flat, indexed collection of facts that describes the known world. In database terminology, this is an **extensional database** (EDB). The query module (`Source/Knowledge/query.pl`) adds an **intensional** layer on top: it defines high-level query predicates that are compiled down to direct lookups over the base relations.

This architecture is closely related to **Datalog**, the declarative query language that sits at the intersection of logic programming and relational databases. In Datalog, ground facts form the base relations and rules define derived views; queries are conjunctive queries over those relations, with guaranteed termination. portage-ng's query layer follows the same pattern: list queries compile into conjunctions of cache lookups (conjunctive queries), every variable is grounded through the EDB (the Datalog safety property), and the query layer itself always terminates. Where the system goes beyond strict Datalog is in its use of compound terms (`version/7`, `slot/1`) rather than flat constants, and in the model queries that invoke the prover — at which point we leave the Datalog fragment and enter full recursive Prolog reasoning.

Rather than interpreting queries at runtime through a generic search function, portage-ng uses SWI-Prolog's `goal_expansion/2` — a compile-time macro facility that acts as a Datalog-style query compiler — to rewrite high-level query goals into **direct** calls to indexed cache predicates before the program even runs.

6.7.1 Goal expansion by example

Consider a rule that needs to find all ebuilds named `neovim`:

```
query:search(name(neovim), Repository://Entry).
```

At load time, `goal_expansion/2` rewrites this into:

```
cache:ordered_entry(Repository, Entry, _, neovim, _).
```

The high-level `search` call disappears entirely. What remains is a direct call to the indexed cache predicate, where SWI-Prolog's first-argument indexing on `Repository` and fourth-argument indexing on `Name` make the lookup near-instantaneous. No dispatching, no interpretation — just a pattern match against the fact base.

A conjunctive query expands into a conjunction of cache calls:

```
query:search([name(neovim), category('app-editors')], R://E).
```

becomes:

```
cache:ordered_entry(R, E, 'app-editors', neovim, _).
```

The payoff shows up at scale: **sub-millisecond** query behavior across **tens of thousands** of entries (on the order of 32,000+ in a typical Portage tree), because the hot queries are specialised at compile time.

6.7.2 `query:search` — the main query predicate

`query:search/2` is the primary interface for querying the knowledge base. Its first argument describes what to search for; its second argument binds the matching `Repository://Entry`:

```
query:search(name(neovim), R://E).
query:search([category('dev-libs'), name(openssl)], R://E).
query:search(description(D), portage://'app-editors/neovim-0.12.0').
```

The following search terms are supported:

Search term	Matches
`name(Name)`	Package name
`category(Cat)`	Package category
`entry(Id)`	Full entry atom (`'category/name-version'`)
`repository(Repo)`	Repository atom
`version(Ver)`	Exact version term
`slot(Slot)`	Slot value
`subslot(Sub)`	Sub-slot value
`keyword(KW)`	Architecture keyword
`description(D)`	Package description
`eapi(E)`	EAPI version
`license(L)`	License
`homepage(H)`	Homepage URL
`maintainer(M)`	Package maintainer
`eclass(E)`	Inherited eclass
`iuse(Flag)`	USE flag declared in IUSE
`masked(true/false)`	Whether the package is masked

Search terms can be combined as a list for conjunctive queries. The `all(...)` wrapper collects all matching values, and `latest(...)` returns only the first (highest-version) match.

6.7.3 `query:select` — version and metadata comparison

For queries that need comparison operators (not just equality), portage-ng uses a `select(Key, Comparator, Value)` term inside `search`:

```
query:search(select(version, greaterequal, Ver), R://E).
query:search(select(slot, equal, '3'), R://E).
query:search(select(keyword, wildcard, 'amd*'), R://E).
```

For version comparisons, the `select` clauses expand at compile time into direct `cache:ordered_entry` lookups combined with `eapi:version_compare/3`:

Comparator	Meaning
`equal`	Exact version match
`smaller`	Version strictly less than
`greater`	Version strictly greater than
`smallerequal`	Less than or equal
`greaterequal`	Greater than or equal
`notequal`	Not equal
`wildcard`	Wildcard match (e.g. `3.0*`)
`tilde`	Fuzzy matching (same base version, any revision)

For non-version keys, `select` falls through to `cache:entry_metadata/4` lookups with the appropriate comparison. This keeps the version hot path — which is exercised thousands of times during candidate selection — fully indexed and compiled.

6.7.4 `model(...)` queries — dependency model construction

Beyond simple metadata lookups, `query:search/2` also supports **model queries** that compute derived structures from the raw cache data. The most important of these constructs the **grouped dependency model** for an ebuild:

```
query:search(model(dependency(Merged, install)):config?{Ctx}, R://E).
query:search(model(dependency(Merged, run)):config?{Ctx}, R://E).
query:search(model(dependency(Merged, pdepend)):config?{Ctx}, R://E).
query:search(model(dependency(Merged, fetchonly)):config?{Ctx}, R://E).
```

Each of these is expanded at compile time into a sequence of:

1. **Self-context injection** — ensures `self(R://E)` is present in `Ctx`, so downstream rules can identify circular self-dependencies.
2. **`findall` over raw dependency metadata** — collects all `cache:entry_metadata/4` facts for the relevant dependency keys (BDEPEND, CDEPEND, DEPEND, IDEPEND, RDEPEND, and/or PDEPEND depending on the phase).
3. **`prover:prove_model/6`** — evaluates USE-conditional branches in the dependency specification, producing an AVL model of active dependency literals.
4. **`group_dependencies/2`** — groups the flat dependency list by category/name/slot/phase, producing the `grouped_package_dependency` terms that the rules layer consumes.

The result `Merged` is a list of grouped dependency terms ready for resolution by the rules layer (see Chapter 12).

How is this cached? Model construction depends on mutable proof state beyond the explicit context argument: `build_with_use` varies per dependency path, `prover:assuming` flags change between fallback attempts, and `memo:selected_cn_snap_` evolves during the proof. Two early cache attempts that tried to *clear* the cache when this state changed were abandoned as unsound. The current design instead encodes every mutable input **in the cache key**: results are memoised per proof in `memo:dep_model_cache_/5` under a hazard-encoded key (proof-

context USE state, the active assumption flags, the entry's currently selected choice-group C/N pairs), gated by `config:dep_model_cache/1`. See the "Dependency-model cache key" section in `Source/Knowledge/query.pl` for the full key design.

6.8 Further reading

- Chapter 7: The EAPI Grammar — how md5-cache files are parsed into cache predicates
- Chapter 3: Configuration — repository path setup
- Chapter 8: The Prover — how the prover queries the knowledge base
- Chapter 20: Gentoo Linux Security Advisories (GLSA) — sibling `Knowledge/glsa.qlf` store (not a package repository)

Chapter 7

The EAPI Grammar

The Gentoo **Package Manager Specification** (PMS) defines a dependency language that is easy to underestimate on first reading. A single atom can carry a version comparator, a category and package name, a Gentoo version string, slot and sub-slot operators, USE restrictions, and —wrapped around lists of atoms—USE conditionals, choice groups, and blockers. Traditional Portage implements this surface syntax with an ad hoc parser written in Python.

portage-ng uses Prolog's built-in **Definite Clause Grammar** (DCG) notation to encode the same language directly (`Source/Domain/Gentoo/eapi.pl`). The insight is simple: PMS dependency syntax *is* a grammar, so it should be expressed as one. DCG rules describe that grammar directly; the parser is the grammar, not a separate layer that tries to stay in sync with a prose spec. What PMS says and what the executable parser accepts are one artifact—the rules in `eapi.pl`—rather than a specification document drifting away from a pile of regexes and special cases.

The grammar fully implements PMS 9 / EAPI 9.

7.1 What gets parsed

The EAPI grammar is exercised whenever md5-cache metadata is loaded. Each file under the Portage tree's `metadata/md5-cache/` directory holds **one ebuild's** worth of metadata as a flat list of lines, each line a single `KEY=VALUE` pair (PMS 9, §14.3). A typical fragment looks like this:

```
BDEPEND=>=dev-build/cmake-3.16
DEFINED_PHASES=compile configure install prepare test
DEPEND=dev-libs/openssl:= dev-libs/libffi:=
EAPI=8
IUSE=debug doc test
KEYWORDS=~amd64 ~arm64
RDEPEND=dev-libs/openssl:= >=dev-lang/python-3.10[ssl,threads]
REQUIRED_USE=|| ( python_targets_python3_11 python_targets_python3_12 )
SLOT=0
```

The DCG is responsible for turning the *values* of dependency-related keys into structured terms: dependency strings (`DEPEND`, `BDEPEND`, `RDEPEND`, `PDEPEND`, `IDEPEND`), USE-conditional groups, version operators, slot operators, USE dependencies, and `REQUIRED_USE` constraints. Other keys (`EAPI`, `SLOT`, `KEYWORDS`, `DESCRIPTION`, …) use smaller, dedicated value rules in the same module —still DCG-driven, but without the full dependency expression machinery.

7.2 A worked example: one dependency atom

Consider a single atom as it might appear in `RDEPEND` or `DEPEND`:

`>=dev-libs/openssl-3.0:0/3=[ssl,-test]`

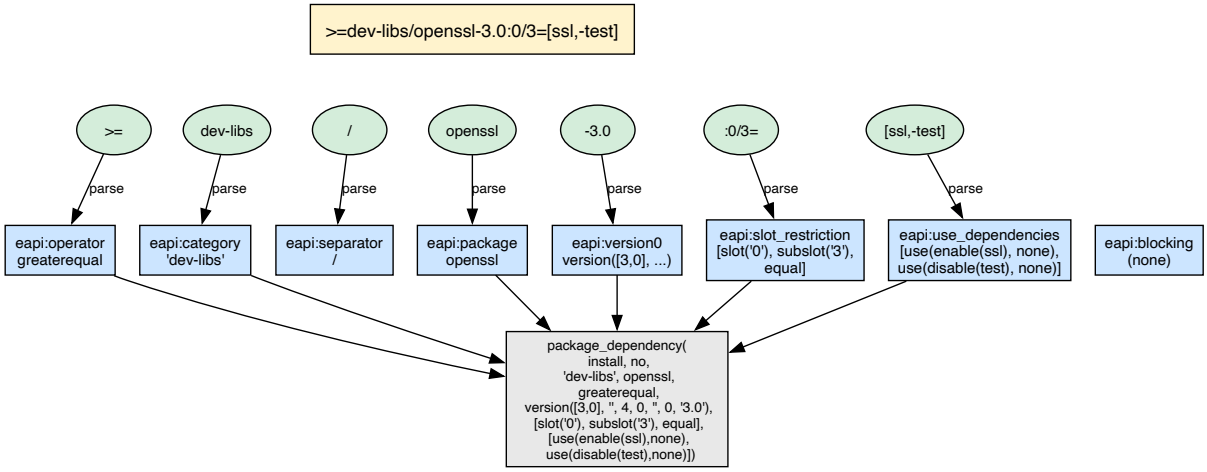


Figure 19 — Dependency atom anatomy

The core DCG rule for a package atom is `eapi:package_dependency/3` in `eapi.pl`. Conceptually it composes the way PMS §8.3 suggests reading the text: optional blocker, optional comparator, `category/package`, optional version, optional slot restriction, optional USE dependency list, with a small helper to merge "`=` + wildcard" into the dedicated `wildcard` operator:

```
eapi:package_dependency(T, _R://_E, Output) -->
  eapi:blocking(B),                                    % optional
  eapi:operator(O),                                    % optional
  eapi:category(C), eapi:separator, !, eapi:package(P),  % required
  eapi:version0(V, W),                                 % optional
  eapi:slot_restriction(S),                            % optional
  eapi:use_dependencies(U),                            % optional
  { eapi:select_operator(O, W, Op),
    Output = package_dependency(T, B, C, P, Op, V, S, U) }.
```

7.2.1 Matching each piece

1. **`blocking`** — No `!` or `!!` prefix, so this clause leaves the blocking marker as "none" (`no` in the concrete term).
2. **`operator`** — The leading `>=` matches the `greaterequal` operator.
3. **`category`**, **`/`**, **`package`** — Consumes `dev-libs`, the slash, and `openssl`.
4. **`version0`** — After the hyphen, parses `3.0` into a `version/7` term (and records that there is no `=*`-style wildcard suffix on this atom).
5. **`slot_restriction`** — The `:0/3=` fragment becomes a list describing the main slot, sub-slot, and trailing `=` (rebuild-on-slot change semantics).
6. **`use_dependencies`** — Bracket contents parse as a comma-separated list: `ssl` as an enable requirement, `-test` as a disable requirement.
7. **`select_operator`** — With no wildcard, the final operator remains `greaterequal`.

7.2.2 Intermediate state and difference lists

Operationally, each DCG goal is expanded into an ordinary Prolog predicate with two extra arguments: the **current suffix** of the input code list and the **remaining suffix** after the rule succeeds. That is the standard DCG **difference-list** threading: parsing advances by shortening the difference between "input seen so far" and "still to read."

You can read the parse as a sequence of **remaining input** snapshots (conceptual, not a separate data structure the code prints):

After this part succeeds	Remaining input (conceptually)
(start)	`>=dev-libs/openssl-3.0:0/3=[ssl,-test]`
`operator`	`dev-libs/openssl-3.0:0/3=[ssl,-test]`
`category` + `/` + `package`	`-3.0:0/3=[ssl,-test]`
`version0`	`:0/3=[ssl,-test]`
`slot_restriction`	`[ssl,-test]`
`use_dependencies`	(empty — parse succeeds)

Calling `phrase(Rule, Codes)` wraps that pattern: it requires the rule to consume all of `Codes` (or you supply an explicit remainder). There is no hand-maintained cursor variable in application code—the DCG expansion supplies it.

7.2.3 Final term

For the install-time dependency role (`package_d` in the grammar), parsing the string above yields a `package_dependency/8` term of this shape (as produced by the running grammar; minor formatting may vary):

```
package_dependency(
  install,
  no,
  'dev-libs',
  openssl,
  greaterequal,
  version([3, 0], '', 4, 0, [], 0, '3.0'),
  [slot('0'), subslot('3'), equal],
  [use(enable(ssl), none), use(disable(test), none)])
```

The first argument (`install` / `run` / `compile`) records *which* PMS dependency class is being parsed, so the same DCG surface syntax feeds slightly different typing in the abstract syntax: `install` for DEPEND/BDEPEND/IDEPEND, `run` for RDEPEND/PDEPEND (PDEPEND is re-tagged `pdepend` at the query layer), and `compile` for CDEPEND.

7.3 Why DCG instead of regex or ad hoc code?

The dependency language defined by PMS is recursive: USE conditionals contain dependency lists, which themselves contain atoms, which may carry nested USE restrictions. A regex or hand-coded string scanner can handle flat patterns, but recursive structure calls for a recursive

formalism. Prolog's DCG notation is exactly that formalism, and it brings several practical advantages.

Composition. The package-atom rule is assembled from small, self-contained nonterminals — `blocking`, `operator`, `category`, `separator`, `package`, `version0`, `slot_restriction`, and `use_dependencies` — each of which can be understood and tested in isolation:

```
dep_atom --> blocking, operator, category, '/', package, version_suffix, slot_suffix,
use_deps.
```

Local testing. Every nonterminal is an ordinary Prolog predicate, so you can call `phrase/2` on a single rule without loading the cache or the full pipeline.

Free recursion. USE conditionals and nested choice groups use the same mechanism as every other rule: `eapi:dependencies//3` recurses through lists of `eapi:dependency//3` alternatives, so nested `flag? ( … )` structures need no separate stack machine.

Failure locality. When parsing fails, the failure occurs at a named nonterminal — a clear indication of which part of the input was unexpected, rather than a terse "pattern did not match" from a regex engine.

Graceful evolution. New PMS features tend to introduce new alternatives or new rules (additional operators, value forms, EAPI-9 extensions), not a rewrite of central control flow. Adding a rule to a DCG is a one-clause change; the equivalent in a Python parser is typically a new branch in an `if eapi >= …` ladder spread across several functions.

7.4 DCG grammar design

The grammar is implemented in `Source/Domain/Gentoo/eapi.pl` as a set of DCG rules. DCGs are a natural fit for dependency specifications because:

- Dependency atoms have recursive structure (USE conditionals nest).
- The grammar is context-free at the level PMS defines.
- DCG rules compose as Prolog predicates, so the parser **constructs** Prolog terms while it reads text.

7.4.1 Dependency atoms

The table below summarizes how the surface syntax maps into fields of the abstract atom (illustrated with the same running example):

Component	Example	Parsed as
Version operator	`>=`	Comparator atom (e.g. `greaterequal`)
Category	`dev-libs`	Atom
Name	`openssl`	Atom
Version	`3.0`	`version/7` term
Slot operator	`:0/3=`	Slot + sub-slot + rebuild flag
USE deps	`[ssl,-test]`	Enable/disable (and related) wrappers

7.4.2 USE conditionals

USE-conditional dependency groups use the syntax:

```
flag? ( deps... )      — include deps if flag is enabled
!flag? ( deps... )     — include deps if flag is disabled
```

These are parsed into conditional terms that the rules layer evaluates against the USE model during proof construction.

7.4.3 Choice groups

PMS defines three choice operators for REQUIRED_USE and dependency specs:

- `||` **(any-of)** — `|| ( a b c )` — at least one of the listed items must be satisfied
- `^^` **(exactly-one-of)** — `^^ ( a b c )` — exactly one must be satisfied
- `??` **(at-most-one-of)** — `?? ( a b c )` — at most one may be satisfied

7.5 Reader/parser pipeline

Loading md5-cache is a small pipeline with clear separation of concerns.

The **repository** side (`Source/Knowledge/repository.pl`) knows where the cache lives: under each tree's `metadata/md5-cache/` directory, with one file per `category/package-version` entry (`repository:get_cache_file/2` resolves entry → path). Sync and incremental updates decide *which* entries need work; for each entry that must be read, the repository opens the flat cache file.

`Source/Pipeline/reader.pl` does one job: given a path (or stream), it reads the file line by line into a list of strings—each string is still a raw `KEY=VALUE` line, unchanged.

`Source/Pipeline/parser.pl` walks that list. For each line it converts the string to character codes and runs `phrase(eapi:keyvalue(metadata, …), Codes)`. That single DCG entry point dispatches on the key: dependency keys delegate to the full dependency grammar (`DEPEND`, `BDEPEND`, `RDEPEND`, `PDEPEND`, `IDEPEND`, `REQUIRED_USE`, …); non-dependency keys use lighter value rules (`EAPI`, `SLOT`, `KEYWORDS`, `IUSE`, …).

So the data flow is:

```
md5-cache files  →  reader.pl (lines)  →  parser.pl  →  eapi.pl (DCG)  →  cache
predicates
```

1. **`reader.pl`** reads each md5-cache file into a list of lines (one key/value pair per line).
2. **`parser.pl`** parses every line through `eapi:keyvalue/3`, which routes values to the appropriate DCG subtree.
3. **`eapi.pl`** builds structured Prolog terms (e.g. `depend(D)`, `rdepend(D)`, `slot(S)`, `eapi(E)`).
4. The results are asserted as `cache:entry/5` and related predicates, populating the knowledge base.

The reader supports incremental loading — only new or changed files need to be re-parsed when using `--regen`.

7.6 Parsed output

After parsing, each ebuild is represented by a set of cache predicates. The dependency model for an ebuild is a list of `package_dependency/8` terms:

```
package_dependency(DepType, Blocking, Category, Name, Operator, Version,
                   SlotInfo, UseInfo)
```

These terms are consumed by the rules layer during proof construction. The EAPI grammar handles all PMS 9 / EAPI 9 constructs:

- Version operators: `=`, `>=`, `<=`, `>`, `<`, `~`, `=*` (wildcard)
- Slot operators: `:SLOT`, `:SLOT/SUBSLOT`, `:*`, `:=`
- USE dependencies: `[flag]`, `[-flag]`, `[flag=]`, `[!flag=]`, `[flag(+)]`, `[flag(-)]`
- Blockers: `!cat/pkg` (weak), `!!cat/pkg` (strong)
- All-of groups (implicit conjunction)
- Any-of groups (`|| ( ... )`)
- USE conditionals (`flag? ( ... )`, `!flag? ( ... )`)

7.7 Further reading

- Chapter 6: Knowledge Base and Cache — how parsed data is stored
- Chapter 12: Resolution — Configuration as Proofs — how dependency terms are consumed during proof construction
- Chapter 24: Dependency Ordering — PMS dependency type semantics

Chapter 8

The Prover

8.1 Domain independence

The central design insight is easy to miss because portage-ng is *about* Gentoo packages: **the prover does not know what a "package" is.** It works only with abstract literals and rules (Logic). That is deliberate. It means the reasoning core can be exercised and tested without importing the whole Portage domain, and the same engine could — in principle — prove goals in any domain that encodes its constraints as Horn-style expansions behind a single hook.

The prover's only contract with the outside world is this: **given a literal, `resolving:rule/2` (or the configured `rule/2` delegate) returns a body — a list of sub-literals that must hold for the head to hold.** Everything that makes Gentoo "Gentoo" — USE flags, slots, version domains, PDEPEND side effects — lives in the rule layer and in proof-term annotations (`?{Context}`), not in the prover's control flow. The prover walks literals; the domain explains what each literal means.

That separation is what keeps the implementation in `Source/Pipeline/prover.pl` readable: backward chaining, cycle handling, context merging, and bookkeeping — not emerge policy.

The prover is the core reasoning engine of portage-ng. Given a list of target literals, it constructs a formal proof that all dependencies can be satisfied — or completes with explicit assumptions documenting exactly where the dependency specification is unsatisfiable.

8.2 Why AVL trees?

The prover maintains its main state in **four AVL trees** from `library(assoc)` (Proof, Model, Constraints, and Triggers — see the module header in `prover.pl`). Plain hash tables would win on raw point lookups, but assoc trees buy a property that matters more here: they are **persistent** (functional). Each `put_assoc/4` produces a *new* tree and leaves the previous one intact.

In Prolog, that lines up with **backtracking**. When the prover must undo a choice, variable bindings revert and the "old" assoc values bound in earlier choice points are still the right snapshots. A mutable hash map would need an explicit save/restore discipline on every failure — the sort of manual undo-stack work traditional Portage does in places, with corresponding risk of subtle inconsistency. Here, the data structures and Prolog's search rule stay aligned.

Complexity is **O(log n)** per update and lookup. For on the order of tens of thousands of literals, that is a small constant number of comparisons (roughly fifteen for 32,000 entries) — more than fast enough compared with the cost of calling into domain rules and unification.

8.3 Inductive proof search

The prover performs inductive proof search via backward chaining. For each literal in the proof queue:

1. **Check the model.** If the literal is already proven (present in the Model AVL), merge contexts via feature term unification and continue.
2. **Check the cycle stack.** If the literal is currently being proved (on the stack), handle the cycle:
   - If `heuristic:cycle_benign/2` (literal + cycle path) succeeds, treat it as already proven (benign cycle — no assumption recorded).
   - Otherwise, record a cycle-break assumption (`assumed(rule(Lit))` in Proof, `assumed(Lit)` in Model).
3. **Expand via `rule/2`.** Call `resolving:rule(Lit, Body)` to get the rule body — the list of sub-literals that must be proven to justify `Lit`.
4. **Record in Proof.** Store `rule(Lit) → dep(N, Body)?Ctx` in the Proof AVL, where `N` is the dependency count.
5. **Record in Model.** Store `Lit → Ctx` in the Model AVL.
6. **Update Triggers.** For each body literal, add `Lit` to its trigger list in the Triggers AVL.
7. **Recurse.** Add the body literals to the proof queue and continue.

Steps 1 and 6 are where **prescient proving** and the **reverse-dependency index** connect; the sections below unpack those ideas.

8.4 Proof term structure

The Proof AVL maps rule keys to structured values:

```
rule(Lit) → dep(DepCount, Body)?Ctx
```

Field	Meaning
`rule(Lit)`	The literal that was proven
`DepCount`	Number of dependencies (body length)
`Body`	List of body literals
`Ctx`	Context under which the literal was proven

The dependency count is stored alongside the body because it is used by downstream consumers without having to recompute it: the explainer reads it when reconstructing justifications, and the special value `-1` marks cycle-break assumptions. Storing the count once, at proof time, avoids repeated `length/2` calls over the same body list.

Special keys: - `assumed(rule(Lit))` with `dep(-1, Body)?Ctx` — prover cycle-break - `rule(assumed(Lit))` with `dep(0, [])?Ctx` — domain assumption

8.4.1 Concrete Proof AVL entry

The literal itself is still just a term the prover passes through; the `portage://` prefix and atom naming are domain choices. A representative Proof entry after expanding an install goal might look like this (body shortened):

```
rule(portage://'sys-apps/portage-3.0.77-r3':install)
  → dep(5,
        [ portage://'dev-lang/python-3.12.3':install,
          portage://'sys-libs/glibc-2.40-r5':install,
          … ])?{[ self(portage://'sys-apps/portage-3.0.77-r3'),
                … ]}
```

So the Proof AVL answers: **which rule instance was used**, **how many dependencies** it had, **what the body literals are**, and **under which `?{Context}` list** that expansion was valid. The exact features inside `?{…}` are documented in Chapter 5: Proof Literals; the prover treats them as data merged by feature term unification, not as special cases.

8.5 Model construction

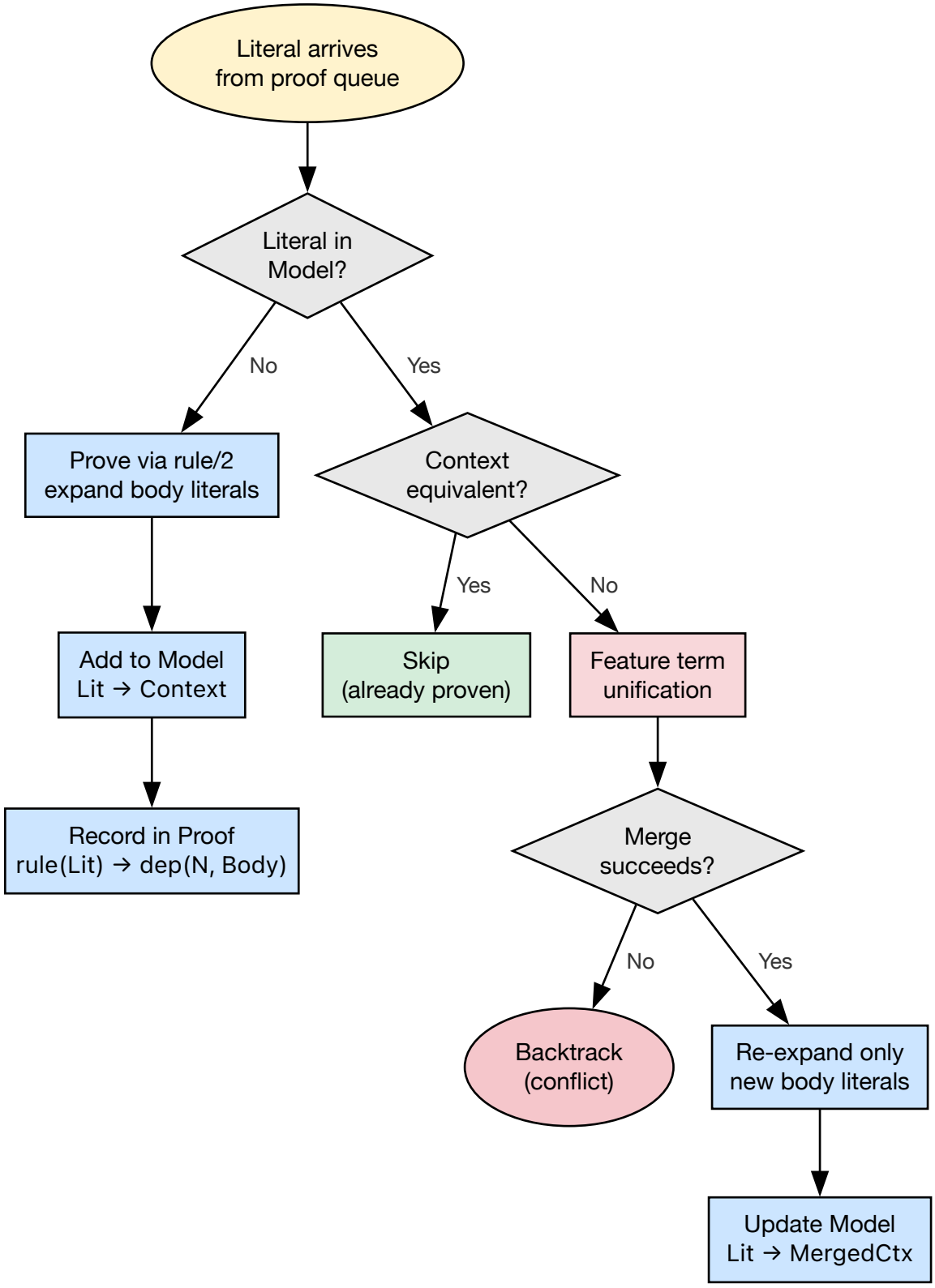


Figure 20 — Model construction flow

The Model AVL records every proven literal with its context. It serves two purposes:

1. **Memoization.** When a literal is encountered again, the prover checks the model first. If found, it merges the new context with the existing one via feature term unification rather than re-proving the literal (when the incoming context is not already equivalent to the stored one — see below).
2. **Plan generation.** The ordering pass and the printer read the model to determine which literals are in the proof and what contexts they carry.

A lightweight variant, `prove_model`, skips Proof and Triggers bookkeeping for internal query-side model construction where only the model is needed.

8.5.1 Concrete Model AVL entry

Model entries are simpler: **literal → context** under which it was last committed to the proof.

```
portage://'dev-libs/openssl-3.3.2':install
  → [ build_with_use:use_state([ssl], []),
      … ]
```

Multiple features can accumulate in that list as different dependency paths impose different requirements; the merge semantics are defined by the domain's feature term unification (`sampler:ctx_union/3`).

8.5.2 Re-encountering a literal: feature term unification

When the queue delivers the same `Lit` again with a **new** `?{Context}`:

1. The prover finds `Lit` in the Model AVL (with stored context `OldCtx`).
2. If the new context is semantically the same as the stored one (`prover:proven/3`), nothing more is done — no second expansion.
3. Otherwise it merges contexts via feature term unification (`sampler:ctx_union/3`). If the merge fails (e.g. conflicting USE enable/disable sets), the goal fails and ordinary Prolog backtracking retracts the choice that led to the clash.
4. If the merge succeeds, the prover may **re-call** `rule/2` on a canonical literal carrying `MergedCtx`, **subtract** the previously proven body from the new body, and prove only the **difference** — updating Proof, Model, and Triggers incrementally and storing `Lit → MergedCtx` in the Model.

So "seen before" does not mean "frozen forever"; it means **accumulate constraints and re-expand only what new information demands.**

8.6 Prescient proving

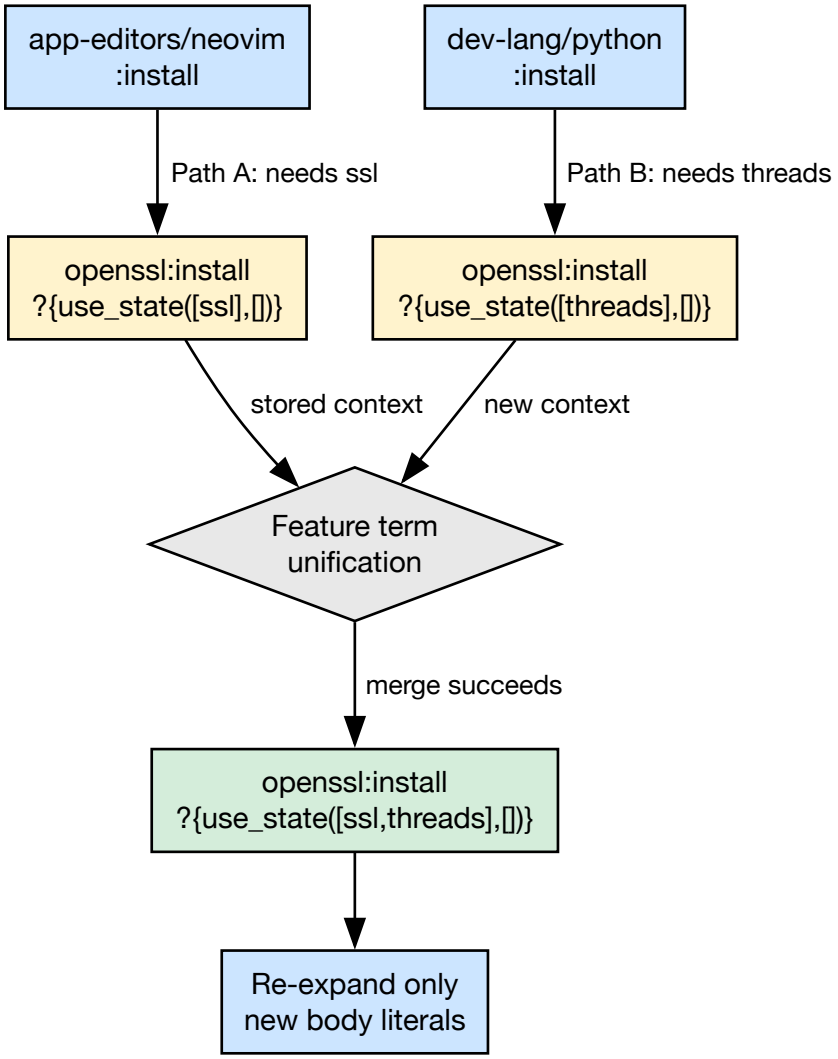


Figure 21 — Prescient proving

When a literal is re-encountered with a changed context (e.g. new USE requirements from a different dependency path), the prover merges contexts via feature-unification and re-expands only the difference. This is called **prescient proving** because knowledge about constraints imposed later in the proof is incorporated into earlier decisions **without** unwinding the whole branch and starting the literal from scratch.

8.6.1 Walkthrough: two paths into `dev-libs/openssl`

Imagine two dependency paths that both need `dev-libs/openssl:install`:

- Path A pulls it in with **USE** `ssl` required in the build set.
- Path B pulls it in with **USE** `threads` required.

Without prescient-style merging, a naive story would be: prove openssl once under path A's context; later, when path B arrives with incompatible or extra requirements, discover that the

earlier proof was too weak and **backtrack far enough to re-prove** openssl under a wider or corrected context — repeating work and thrashing the search.

With prescient proving, the second encounter does not throw away the first. The prover merges the proof-term contexts:

```
First encounter:   openssl:install ?{use_state([ssl],       [])}
Second encounter:  openssl:install ?{use_state([threads],   [])}
After unification: openssl:install ?{use_state([ssl,threads],[])}
```

The merged context commits openssl to satisfying **both** paths at once. The prover then checks whether this wider context is still consistent with every constraint the rules attach to that literal — profile masks, `REQUIRED_USE`, version domains, and so on.

- If the check **succeeds**, no full re-proof is needed. The prover re-expands the rule under the merged context and proves only the **new** body literals that the wider context introduces beyond what was already established.
- If the check **fails** (for example, contradictory flags — a USE flag required both enabled and disabled), the merge is rejected and the prover **backtracks** to try another candidate.

That is the sense in which the prover is "prescient": **later requirements are folded into the context of an earlier proof step** through merging and targeted re-expansion, rather than discovering the conflict only after committing to a too-narrow past choice.

8.7 Triggers and the reverse-dependency index

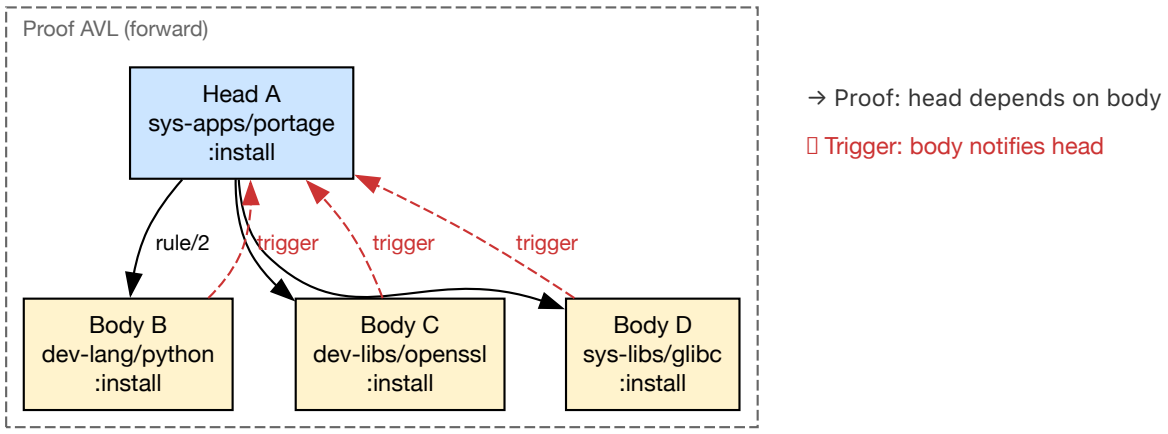


Figure 22 — Triggers reverse-dependency index

The Triggers AVL is the piece that makes prescient updates **addressable**: it records, for each body literal, **which rule heads depend on it**.

8.7.1 Construction

When the prover proves head literal **A** with body **[B, C, D]**, it extends the Triggers AVL with three reverse edges:

```
Proof (forward):     rule(A) → dep(3, [B, C, D])

Triggers (reverse):  B → [A, …]
                     C → [A, …]
                     D → [A, …]
```

In general, for a rule `rule(H, Body)` where `Body = [L₁, L₂, …, Lₙ]`, the operation `add_triggers/4` inserts:

> For each $L_i \in$ Body: Triggers[L_i] := [H | Triggers[L_i]]

Each time a rule is recorded or re-recorded (including after a prescient merge), these reverse edges are extended so the index stays consistent with the current bodies.

8.7.2 Concrete example

Suppose the prover establishes:

```
rule(sys-apps/portage:install) → dep(3, [dev-lang/python:install,
                                         dev-libs/openssl:install,
                                         sys-libs/glibc:install])
```

The Triggers AVL then contains:

```
dev-lang/python:install  → [sys-apps/portage:install, …]
dev-libs/openssl:install → [sys-apps/portage:install, …]
sys-libs/glibc:install   → [sys-apps/portage:install, …]
```

If openssl's context later changes (prescient merge adds a new USE flag), the prover can look up `Triggers[dev-libs/openssl:install]` in O(log n) time and immediately find that `sys-apps/portage:install` needs to be revisited.

8.7.3 Forward vs reverse lookup

The Proof and Triggers AVLs are **duals** of each other:

Direction	AVL	Lookup	Answers
Forward	Proof	`rule(H) → dep(N, Body)`	What does H depend on?
Reverse	Triggers	`L → [H₁, H₂, …]`	Who depends on L?

Together they form a **bidirectional dependency graph**: Proof walks forward from heads to bodies; Triggers walks backward from bodies to heads.

8.7.4 Downstream use

Later phases — the orderer (merge-order bias), diagnostics — use Triggers to answer **"if this literal moves, what else moves?"** in logarithmic time per lookup. Without these reverse edges,

nothing in the pipeline could enumerate which install heads depended on a given dependency literal, and the graph carried out of proving would be incomplete for anything that walks dependencies backwards.

8.8 Entry rules and the pipeline

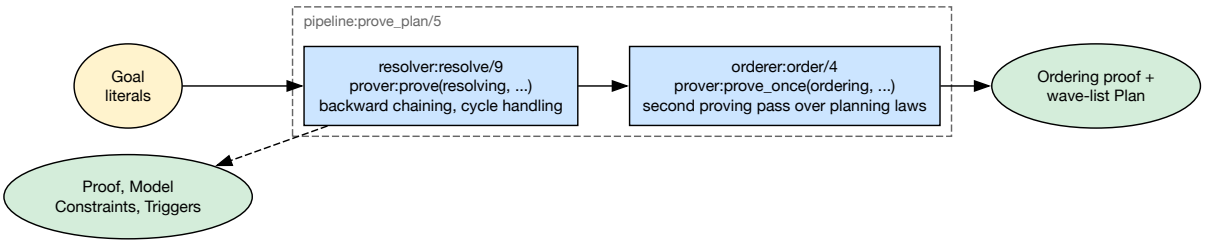


Figure 23 — prove_plan pipeline

The pipeline module provides two canonical entry points, both with the same 5-tier committed-choice progressive relaxation (strict, keyword_acceptance, blockers, unmask, keyword_unmask):

- `pipeline:prove_plan_with_fallback/5` — full pipeline (prove + order). Used by production paths (`--pretend`, `--graph`, `--build`) and `pipeline:test_stats`.
- `pipeline:prove_with_fallback/4` — resolve pass only. Used by layered tests (`resolver:test`, `orderer:test`) and `--bugs`. Each test layer adds its own stages on top.

Underneath, `prove_plan_basic/5` chains two stages:

```
pipeline:prove_plan_basic(Goals, ProofAVL, ModelAVL, Plan, TriggersAVL)
```

1. `resolver:resolve/9` — hands the `resolving` rule set to `prover:prove/10`; constructs Proof, Model, Constraints, and Triggers
2. `orderer:order/4` — hands the `ordering` rule set to the same prover for a second proving pass; projects the wave-list plan (Chapter 13)

The prover is wrapped in `with_reprove_state` which saves and restores the learned constraint store across reprove retries. Inside that, `prove_with_retries` catches `prover_reprove` exceptions and restarts up to `reprove_max_retries` times (default 3).

See Chapter 9: Assumptions and Constraint Learning for the reprove mechanism in detail.

8.9 Multiple stable models

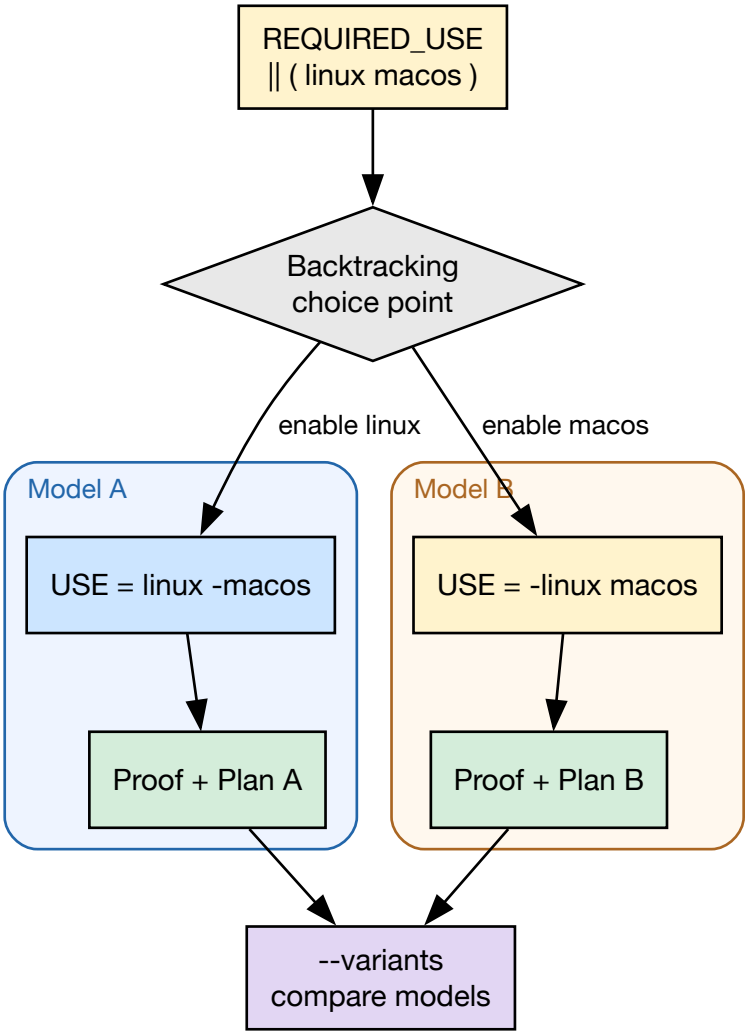


Figure 24 — Multiple stable models

The prover can produce different solutions (stable models) of the USE flag configuration space. Variants are explored by re-proving the target under thread-local branch-preference overrides (`variant:use_override`, `variant:branch_prefer`) that steer the committed choices — the choice-group cut is never removed within a single proof.

For example, a `REQUIRED_USE="|| ( linux macos )"` constraint yields two stable models:

```
Model A:  USE="linux -macos"     Model B:  USE="-linux macos"
```

The `--variants` CLI option enables this mode, running the prover with different USE flag configurations via `variant:use_override` and `variant:branch_prefer`.

8.10 Proof obligations

After a literal is proven, the prover queries the domain for additional proof obligations via `heuristic:proof_obligation/4`. This lets the domain inject derived obligations — extra literals

to be appended to the proof queue — without the prover understanding domain-specific semantics.

PDEPEND dependencies are handled this way: they are discovered only after a literal is resolved and are injected as proof obligations via `heuristic:proof_obligation/4`.

8.11 Choice-event log

For debugging which `||` arm or version candidate the resolver tried, use `--choice-log` via the `portage-ng-dev` wrapper. The wrapper passes `-Dchoice_log=true` so emit/wrap sites are compiled in; without that define, `goal_expansion` compiles them to `true` / the wrapped Goal (zero overhead, same pattern as `--profile` / `instrumentation`).

Runtime arming still requires `--choice-log` (or `choicelog:arm/0`). `choicelog` (`Source/Application/Performance/choicelog.pl`) records:

- `any_of` — trying / succeeded / failed for choice-group arms
- `version` — alternative multi-candidate binds (`index > 1` only)
- `reject` / `learn` / `reprove` / `assumption` — sparse conflict path

After prove, a human-readable dump is written to stderr. From `--shell` (with `-Dchoice_log=true`), `choicelog:events/1` returns the term list and `choicelog:dump/0` reprints it. See also Policy: Choice.

8.12 Further reading

- Chapter 9: Assumptions and Constraint Learning — the reprove mechanism and constraint learning
- Chapter 5: Proof Literals — the literal format
- Chapter 11: Rules and Domain Logic — how `rule/2` and rule modules plug in
- Chapter 12: Resolution — Configuration as Proofs — the `resolving` rule set
- Chapter 4: Architecture Overview — the full pipeline

Chapter 9

Assumptions and Constraint Learning

9.1 Always a proof, never a dead end

Most dependency resolvers stop when they cannot satisfy a constraint: no solution, no plan, and often little more than a terse error. portage-ng takes a different stance. **It does not give up.** When every above-board alternative has been exhausted, the prover records an **assumption** — effectively: "I am proceeding *as if* this dependency could be satisfied" — and continues building the proof.

The outcome is **always** a complete plan. Either the proof is **strict** (no assumptions), or it is a proof **under assumptions**, with the unresolved fragments called out explicitly. Assumptions are not treated as opaque failures; they are **proposals**. They tell you which pieces of configuration or tree state would need to change for the same reasoning chain to become a strict proof. This is the same habit of mind as in mathematics: *"Assuming the Riemann hypothesis, we can prove …"* — the argument is valid *conditional* on the assumptions; make them true, and the condition disappears.

The sections below walk through a concrete missing-dependency example, then explain how suggestion tags turn assumptions into actionable hints. After that, the chapter documents the same mechanisms in technical detail: assumption taxonomy, reprove loop, REQUIRED_USE flow, entry rules, constraint guards, progressive relaxation, assumption printing, and the analogy to conflict-driven clause learning.

9.2 Worked assumption example: missing `dev-libs/foo`

Suppose the user runs:

```
portage-ng --pretend some-package
```

and some package in the graph depends on **`dev-libs/foo`**, which has **no ebuild** in any repository the knowledge base knows about.

What the prover does

1. The **grouped package dependency** rule tries to prove the dependency by enumerating candidates for category `dev-libs` and name `foo`.
2. **Search returns no entries** — there is nothing to install, so every candidate path fails.

3. After **backtracking** exhausts those paths, the **fallback chain** runs (parent narrowing, reprove with learned domains, and so on, as documented below). None of that invents a missing package.
4. The domain layer finally takes the **assumption path**: it builds a condition whose head is `assumed(grouped_package_dependency(…))`, tags the proof-term context with a reason (and optional suggestions), and the catch-all rule `rule(assumed(_), [])` lets the prover close that branch of the proof.

What appears in the Proof AVL

The proof tree stores a **domain assumption** with a key of the form `rule(assumed(Lit))`, where `Lit` is the grouped-dependency literal. For example (category and name as atoms, dependency list abbreviated):

```
rule(assumed(grouped_package_dependency('dev-libs', 'foo', …):config?{Ctx}))
    → dep(0, [])?Ctx
```

The exact `Action` (`:config`, `:install`, `:run`, …) depends on which phase of the grouped dependency is being proved; the important invariant is the **`rule(assumed(...))`** proof key (see Assumption Taxonomy).

What the user sees in the plan output

The printer classifies this as a non-existent dependency and emits a **Domain assumptions** block, along the lines of:

```
Domain assumptions: dev-libs/foo (non-existent)
```

Exit code

When any **domain** assumption is present, the CLI exit code is **`2`** (cycle-break-only assumptions alone yield **`1`**; a fully strict proof yields **`0`**).

How to read it

The message is intentionally operational: **to resolve this assumption, ensure `dev-libs/foo` is available in your repository** (overlay, third-party tree, or corrected package name). The plan is still a single coherent merge order; the assumption marks the gap between what the prover can justify from facts and what you must supply from outside.

9.3 Assumptions as actionable proposals

Many assumptions carry **`suggestion(Type, Detail)`** (and related) tags in the literal's **`?{Context}`** list. These encode **configuration changes** that would move the proof toward strictness — often the same changes the **progressive relaxation** tiers simulate when you widen `assuming/1` flags.

Typical shapes in the codebase include:

Tag (representative)	User-facing intent
`suggestion(keyword, '~amd64')`	Accept the unstable keyword — e.g. add to **`package.accept_keywords`** (in sources: `suggestion(accept_keyword, '~amd64')`)
`suggestion(unmask, …)`	Unmask the package — e.g. **`package.unmask`** (`Repo://Entry` when known)
`suggestion(use, …)`	Adjust USE flags — e.g. **`package.use`** (in sources: `suggestion(use_change, Repo://Entry, Changes)`)

When you run in modes that **apply** suggestions (see the builder's `execute_suggestion/…` hooks), those changes are **already reflected in the plan**; your job is to **review and approve** them in your real `/etc/portage` layout, or to treat the tags as a checklist for manual edits. Progressive Relaxation ties the same ideas to the `assuming` tiers (`keyword_acceptance`, blockers, `unmask`).

9.4 Overview

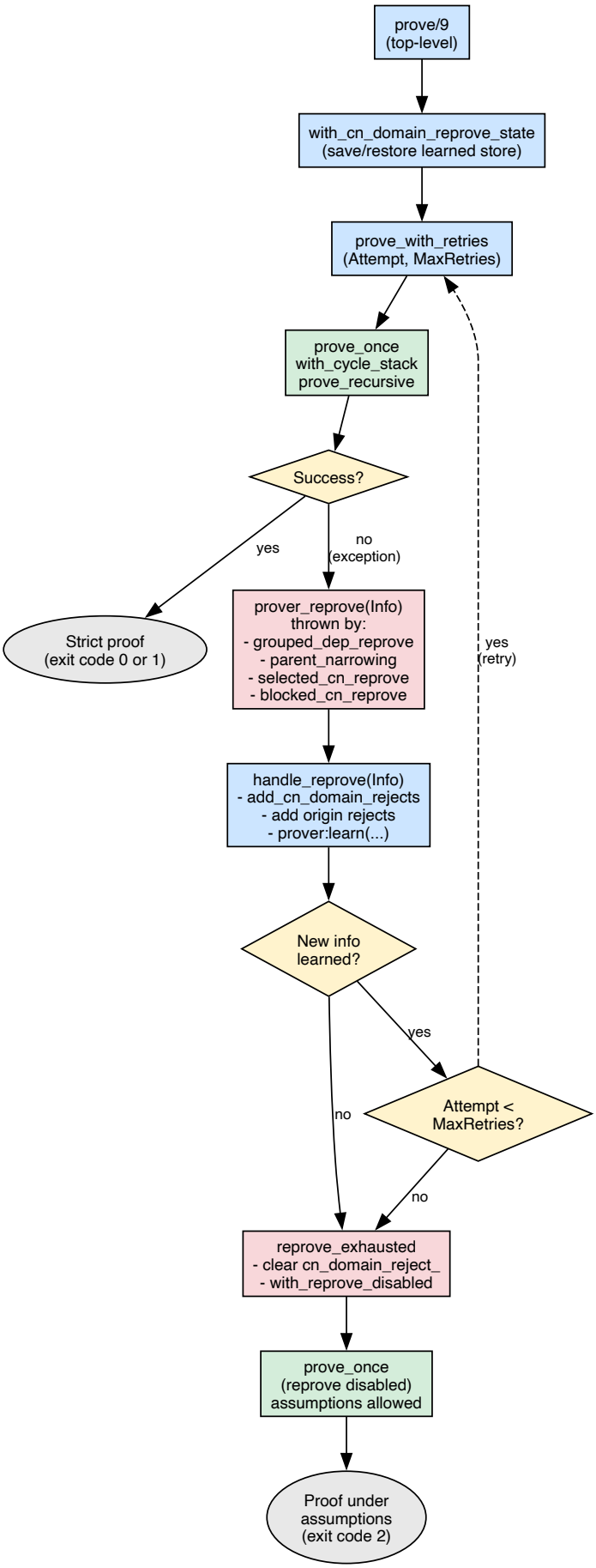


Figure 25 — Reprove mechanism flow

The portage-ng prover builds a formal proof that a set of target packages can be installed. The proof is an AVL tree mapping literals to their justifications. When part of the dependency graph cannot be satisfied, the prover records *assumptions* — lightweight markers that let the proof complete while flagging the unresolved fragment for the user.

Two fundamentally different kinds of assumptions exist, and a bounded reprove mechanism allows the prover to retry the proof with accumulated knowledge before resorting to assumptions.

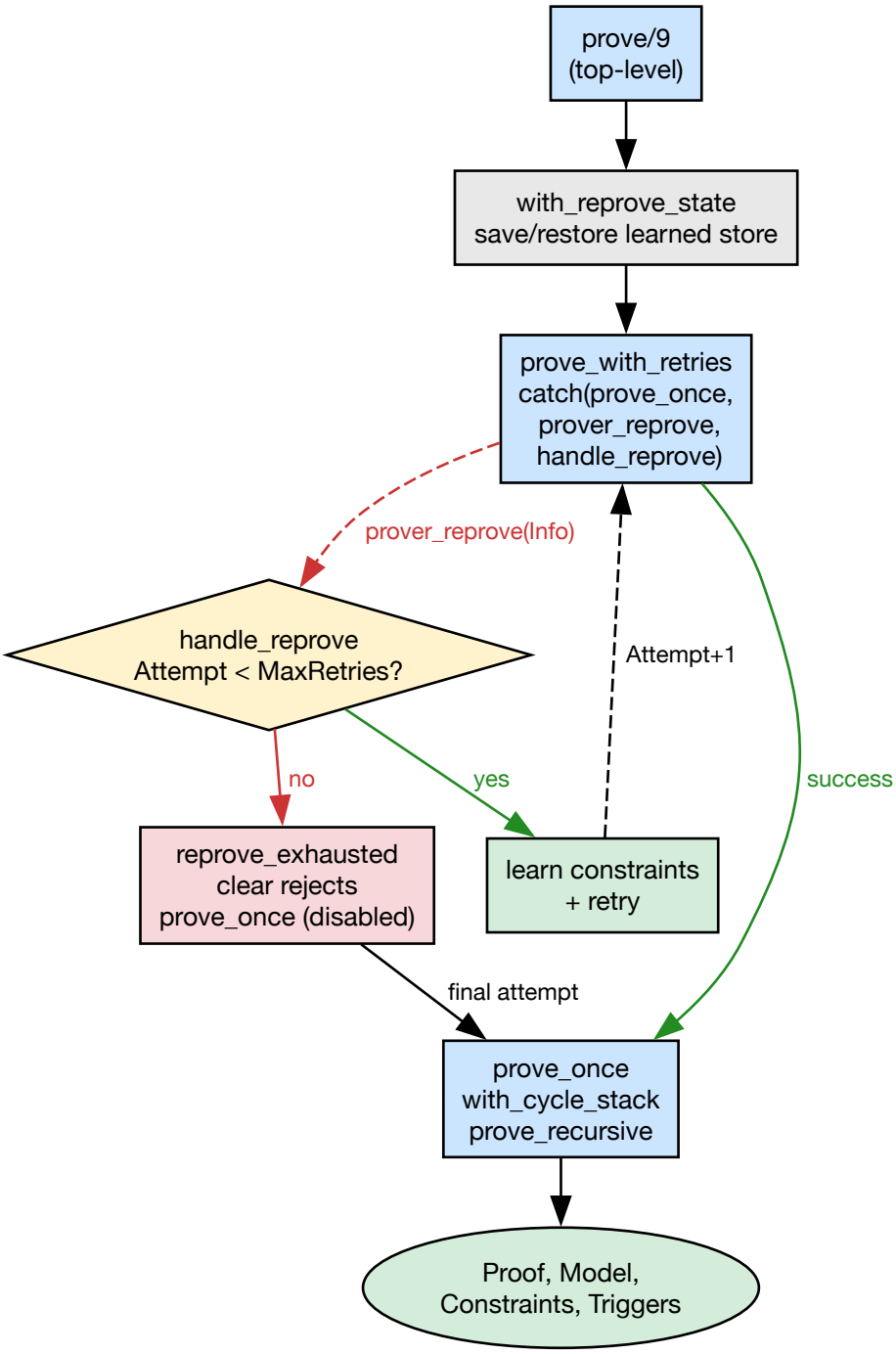


Figure 26 — Reprove loop structure

9.5 Data Structures

The prover maintains four AVL trees during proof construction:

AVL	Key → Value	Purpose
Proof	`rule(Lit)` → `dep(N, Body)?Ctx`	Which rule justified Lit
Model	`Lit` → `Ctx`	Every proven literal + context
Constraints	constraint key → value	Accumulated constraint terms
Triggers	`BodyLit` → `[HeadLit, …]`	Reverse-dependency index

9.6 Assumption Taxonomy

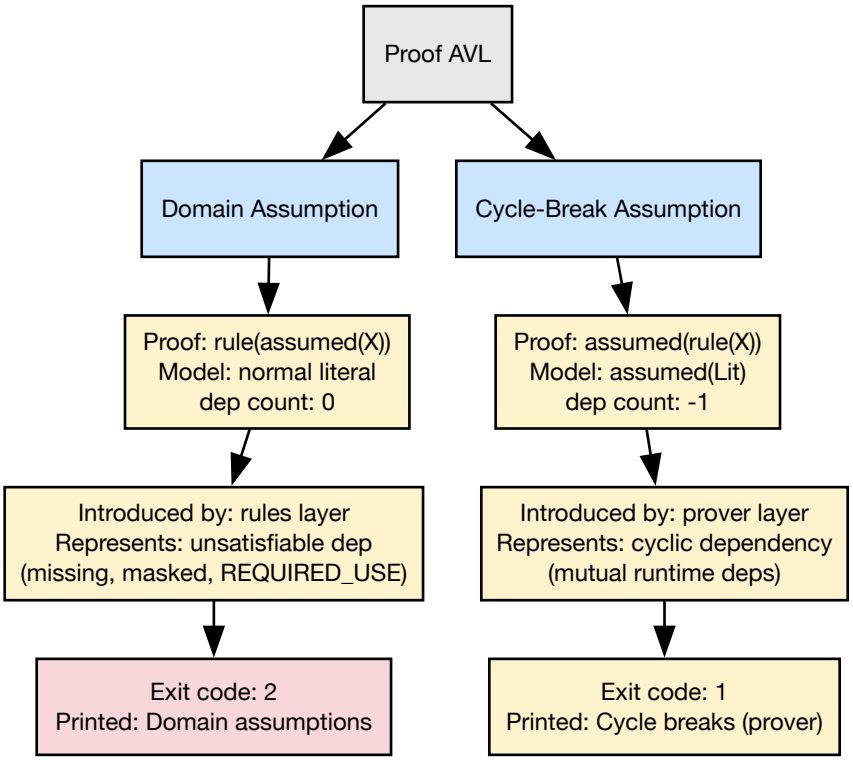


Figure 27 — Assumption taxonomy

The two kinds of assumptions are stored differently in the Proof and Model trees. Confusing them leads to wrong statistics, wrong plan output, or missed warnings.

9.6.1 Domain Assumptions (`rule(assumed(X))`)

Introduced by the **rules layer** when a dependency cannot be satisfied — for example, a package that does not exist in the tree, or a REQUIRED_USE violation that makes every candidate invalid.

How they are created:

The `grouped_package_dependency` rule exhausts all candidates (via Prolog backtracking), then the fallback chain (parent narrowing → reprove → assumption), and finally emits:

```
Conditions = [assumed(grouped_package_dependency(C,N,Deps):Action?{Ctx})]
```

The `assumed(X)` literal in the body is proved by the catch-all rule:

```
rule(assumed(_), []) :- !.
```

This stores `rule(assumed(X))` in the Proof tree.

Where they appear: - Proof: `rule(assumed(X))` → `dep(0, [])?Ctx` - Model: `assumed(X)` → `Ctx` (the `assumed(X)` body literal is proved as a regular literal; exit-code detection scans for these keys) - Plan: rendered as "verify" steps + "Domain assumptions" warning block

9.6.2 Prover Cycle-Break Assumptions (`assumed(rule(X))`)

Introduced by the **prover** when it detects a cycle during proof search. If a literal is already on the cycle stack (currently being proved), the prover cannot recurse further without diverging. Instead, it records a cycle-break:

```
put_assoc(assumed(rule(Lit)), Proof, dep(-1, OldBody)?Ctx, Proof1),
put_assoc(assumed(Lit), Model, Ctx, NewModel)
```

Where they appear: - Proof: `assumed(rule(Lit))` → `dep(-1, Body)?Ctx` - Model: `assumed(Lit)` → `Ctx` - Plan: ordered normally by the ordering pass; cycle explanation via the printer's cycle section (`Printer/Plan/cycle.pl`)

9.6.3 Summary Table

Property	Domain Assumption	Prover Cycle-Break
Proof key	`rule(assumed(X))`	`assumed(rule(X))`
Model key	`assumed(X)`	`assumed(Lit)`
dep count	0	−1
Introduced by	rules layer	prover layer
Represents	unsatisfiable dependency	cyclic dependency
Printed as	"Domain assumptions"	cycle break
Exit code contribution	2	1

9.7 Reprove Mechanism

When a conflict is detected during proof search, the domain layer does not simply fail — it records what went wrong and requests a retry with refined knowledge.

9.7.1 Triggering Reprove

Several predicates can throw `prover_reprove(Info)`:

Source	When
`maybe_learn_wildcard_domain`	Wildcard dep fails and parent already narrowed or single-version; learns upper-bound `cn_domain` from wildcard
`maybe_learn_parent_narrowing`	Parent introduced a dep that made (C,N) unsatisfiable; rejects the exact parent entry
`maybe_request_grouped_dep_reprove`	Effective domain conflicts with selected CN; domain inconsistent; version/slot constraints present
`selected_cn_unique_or_reprove`	CN-domain constraint conflicts with already-selected candidate (constraint guard)
`selected_cn_not_blocked_or_reprove`	Blocker detected via blocked source snapshot

Each throws `prover_reprove(cn_domain(C, N, RejectDomain, Candidates, Reasons))`.

9.7.2 Handling Reprove

When `prove_with_retries` catches a `prover_reprove(Info)` exception, it delegates to `heuristic:handle_reprove/2`, which proceeds in three steps:

1. **Record what went wrong.** The handler extracts the conflicting category, name, domain, and candidate list from `Info` and adds them to a reject set (`memo:cn_domain_reject_`). If the conflict was introduced by a specific parent, that origin is also recorded so the prover avoids the same path on the next attempt.
2. **Decide whether to retry.** If new information was actually learned (`Added = true`) and the attempt count is still below `reprove_max_retries`, the handler restarts the proof from scratch by calling `prove_with_retries` with an incremented attempt counter. The learned rejects carry over, so the prover will not repeat the same conflict.
3. **Give up gracefully.** If nothing new was learned, or the retry budget is exhausted, the handler calls `reprove_exhausted`, which **clears** the reject set so the final attempt runs unbiased. It then invokes `prove_once` with reprove **disabled** — no further `prover_reprove` exceptions can be thrown, so the proof completes with assumptions where necessary.

9.7.3 Learned Constraint Store

The `prover:learn/3` and `prover:learned/2` predicates maintain a key-value store that **persists across reprove retries** within the same top-level `prove/10` invocation. This is distinct from the reject set (which accumulates and is cleared on exhaustion).

The domain uses learned constraints for:

- **Candidate narrowing** — `grouped_dep_effective_domain` intersects the local+context domain with any learned domain.
- **Conflict learning** — constraint guards learn the domain when a conflict is detected.

- **Parent narrowing** — `maybe_learn_parent_narrowing` excludes the parent version when a child dep cannot be satisfied. This one does *not* go through the learned store: a version domain cannot express a hole, and a `< ParentVer` cut would also discard every newer parent version that was still eligible. The exact parent entry is placed in the reject map instead.
- **Wildcard failure learning** — `maybe_learn_wildcard_domain` derives an upper-bound domain from a wildcard constraint (e.g. `=pkg-0.6*` → `< 0.7`) when parent narrowing alone could not resolve the conflict.

9.7.4 Retry Budget

`reprove_max_retries` defaults to 20 (configurable via `config:reprove_max_retries/1`). The final attempt runs with reprove disabled so the proof can complete with assumptions if necessary. Before it runs, `heuristic:reprove_exhausted/0` clears the reject map (including the parent-narrowing rejects) so the final pass starts from a clean candidate set; only the learned constraint store survives. Keeping the parent-narrowing rejects alive across exhaustion was measured tree-wide and rejected: it fixed 3 targets but added NEGATIVE assumptions in 48 and changed them in 75 more.

9.8 Use model violation flow

When a parent package forces USE flags on a dependency via bracketed USE deps (e.g. `cat/pkg[feature]`), and the dependency's `REQUIRED_USE` forbids that flag combination, the REQUIRED_USE violation mechanism ensures the prover explores alternatives before assuming.

9.8.1 Step-by-step flow

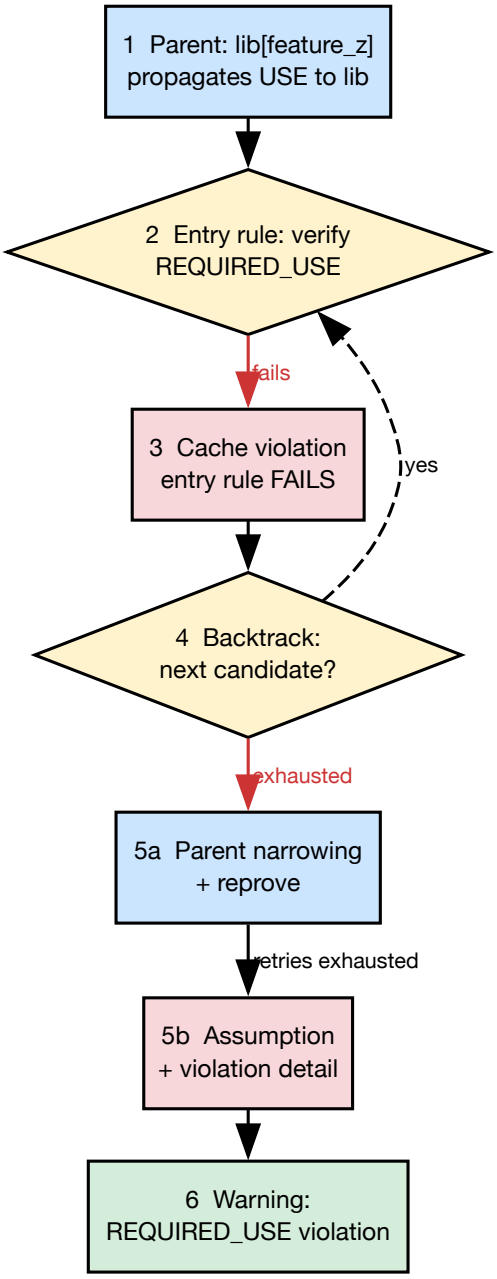


Figure 28 — Use model violation flow

The diagram above shows the six stages the prover walks through when a parent forces a USE flag that violates a dependency's REQUIRED_USE. Each step is explained below.

Step 1 — USE propagation. The parent atom `cat/app` depends on `cat/lib[feature_z]`. The bracketed flag is carried forward as a `build_with_use` context annotation, so every candidate version of `lib` will be evaluated with `feature_z` enabled.

Step 2 — Entry rule verification. When `lib`'s `:install` (or `:run`) entry rule fires, it computes the full USE model and calls `use:verify_required_use_with_bwu` to check whether the resulting flag set satisfies `lib`'s `REQUIRED_USE` expression.

Step 3 — Fail, not assume. If verification fails (e.g. `lib` declares `REQUIRED_USE=!feature_z`), the entry rule caches a structured violation description via `memo:requse_violation_/3` and then **fails**. It does *not* produce an assumption, because doing so would hide the failure from the candidate selection logic and bypass the entire reprove mechanism.

Step 4 — Candidate backtracking. The failure propagates back to `grouped_package_dependency`, which tries the next candidate version of `lib`. A different version may have a different `REQUIRED_USE` that does not conflict with `feature_z`.

Step 5a — Parent narrowing + reprove. When all candidates are exhausted, the fallback chain activates. `maybe_learn_parent_narrowing` rejects the current parent entry (`app-1.0`) and throws `prover_reprove`, giving the prover a chance to retry with a different parent that may not force `feature_z`.

Step 5b — Assumption with violation detail. After all reprove retries are exhausted, the prover falls through to the assumption path. `explanation:assumption_reason_for_grouped_dep` retrieves the cached `requse_violation_` info and enriches the assumption context with a `required_use_violation(...)` tag.

Step 6 — Warning output. The printer recognises the enriched context and emits a structured REQUIRED_USE violation warning, showing which flags were forced, what expression they violate, and which parent triggered the conflict.

9.8.2 Memo cache

The violation info is cached via `memo:requse_violation_/3` (thread-local, survives backtracking since `assertz` is side-effecting). It is: - **Asserted** in the entry rule before failing - **Consumed** in the `grouped_package_dependency` assumption path (retracted after enriching the context) - **Cleared** by `memo:clear_caches/0` at the start of each proof run

9.9 Entry rule structure

Every `:install` and `:run` entry rule follows the same layered structure. Understanding this structure is important because it explains *when* the prover fails, *when* it assumes, and *why* the distinction matters.

Gate checks. The rule begins with a deterministic cut (`!`) — there is exactly one entry-rule clause per literal form, so Prolog must not search for alternatives at this level. Candidate alternatives are provided one level up by `grouped_package_dependency`. After the cut, three quick gate checks run in order:

1. **Mask gate** — if the ebuild is masked and the `unmask` relaxation tier is not active, the rule fails immediately.
2. **Keyword gate** — if no accepted keyword exists and `keyword_acceptance` is not active, the rule fails.
3. **Already-installed short-circuit** — if the package is already installed (and `--emptytree` was not requested) with USE flags that match the request, the rule succeeds with an empty condition list. When the installed USE differs, the entry is re-emitted as a transactional `:update` action instead (portage-ng#85).

USE model verification. When none of the gates apply, the rule queries the ebuild's metadata and computes the full USE model (combining profile defaults, user overrides, and `build_with_use` annotations from the parent). It then checks the result against the ebuild's `REQUIRED_USE` expression. If the check fails, the violation is cached (`memo:requse_violation_/3`) and the rule fails — *not* assumes — so that backtracking can explore other candidate versions (see section 9.8.1).

Dependency model construction. If the USE model passes, the rule builds the dependency model: it looks up cached or freshly computed dependency lists, orders them, and returns the full condition list (selected CN, constraints, download literal, dependency literals, etc.).

If the dependency model itself cannot be built — for example because every branch of an `any_of_group` is filtered out — the rule produces a domain assumption tagged with `issue_with_model`. This is deliberately an assumption rather than a failure, because the problem is intrinsic to the ebuild metadata, not something that trying a different candidate version would resolve.

9.10 Constraint guards and reprove integration

Every time the prover unifies a new constraint term into the proof, it calls `heuristic:constraint_guard(Key, Constraints)` to verify that the constraint is consistent with what has already been proved. The guard has three possible outcomes:

- **Succeed silently** — the constraint is compatible and the proof continues normally.
- **Fail** — the constraint conflicts with the current proof state. Prolog's built-in backtracking explores an alternative within the same proof attempt (e.g. a different candidate version).
- **Throw** `prover_reprove(...)` — the conflict cannot be resolved by simple backtracking. The prover catches the exception, records a learned constraint, and restarts the proof from scratch with the new knowledge (see section 9.7).

Three specialised guards in `cnselect.pl` cover the most common conflict types:

- `selected_cn_unique_or_reprove` checks that the selected category/name pair is consistent with prior selections. If two dependency paths select different versions of the same package in the same slot, this guard detects the inconsistency and triggers a reprove with a narrowed domain.
- `selected_cn_not_blocked_or_reprove` enforces blocker constraints. When a package is blocked by another package that has already been selected, this guard triggers a reprove so the prover can learn to avoid the blocked combination.
- `maybe_request_cn_domain_reprove` handles remaining domain inconsistencies. If the selected version falls outside the intersection of all accumulated version domains, the guard learns the correct domain and triggers a reprove.

The constraint guards above operate within a single proof attempt. But what happens when the entire proof cannot succeed under strict constraints? Rather than immediately falling through to assumptions, the pipeline offers one more tool: progressive relaxation.

9.11 Progressive Relaxation

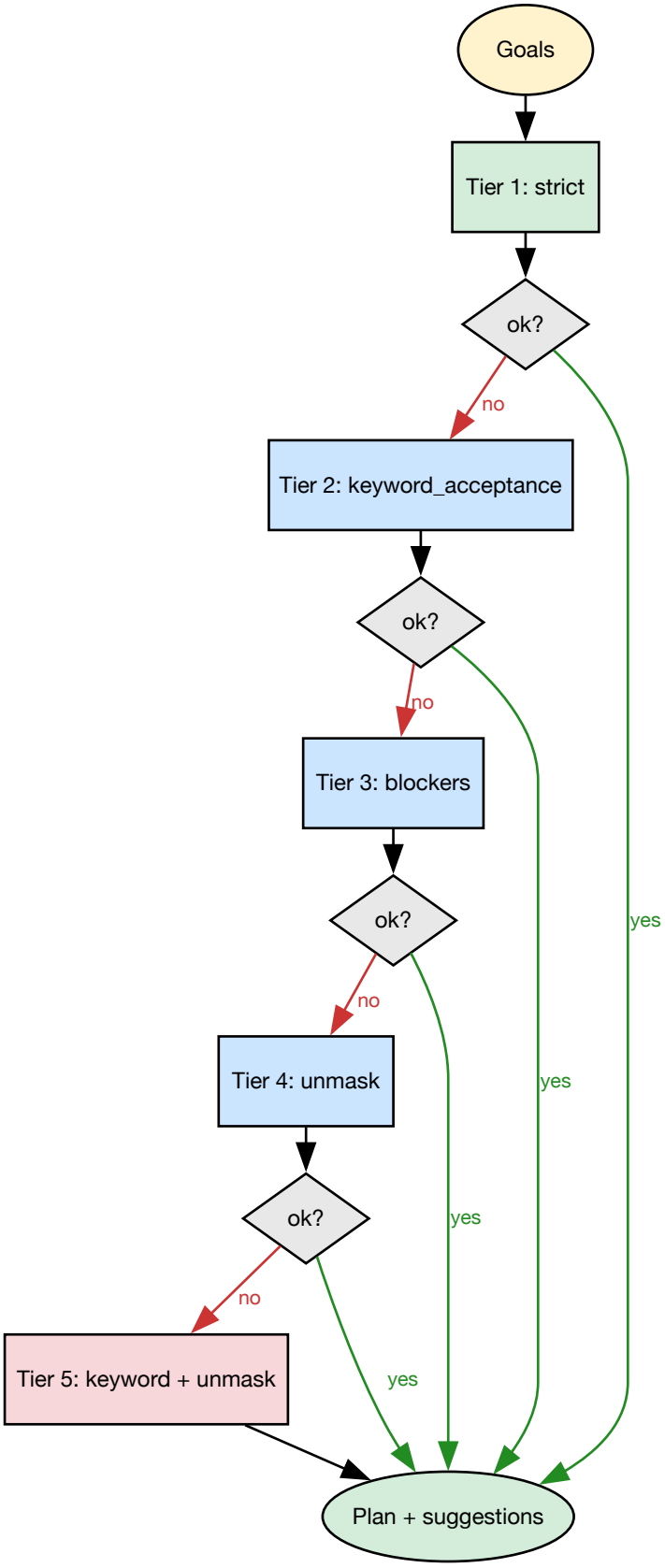


Figure 29 — Progressive relaxation tiers

Not every dependency graph can be satisfied under the strictest interpretation of the repository metadata. A package may exist only with an unstable keyword, or be masked by the profile, or conflict with an already-installed blocker. Rather than giving up at the first such obstacle, the pipeline applies **progressive relaxation**: it re-runs the entire proof under successively weaker constraints until a complete plan emerges.

The mechanism lives in the pipeline's shared fallback driver (`pipeline:fallback_tiers/1` + `pipeline:with_fallback/2`), used by both `prove_plan_with_fallback/5` and `prove_with_fallback/4`. Each tier wraps the prover call inside `prover:assuming/2`, which sets a dynamic flag that the domain rules consult at decision points.

Tier	`assuming` flag	What is relaxed
1 (strict)	none	All masks, keywords, and blockers enforced
2	`keyword_acceptance`	Unstable keywords (`~amd64`) accepted
3	`blockers`	Blocker constraints downgraded to warnings
4	`unmask`	Masked packages unmasked
5	`keyword_acceptance` + `unmask`	Both relaxations combined (last resort)

The tiers are tried in order via Prolog's committed-choice if-then-else (`->` / `;`) — the first tier that succeeds commits and returns a `FallbackUsed` tag (`false`, `keyword_acceptance`, `blockers`, `unmask`, or `keyword_unmask`).

The same 5-tier fallback chain is shared by two canonical entry points in the pipeline module:

- `prove_plan_with_fallback/5` — full pipeline (prove + order), used by production paths (`--pretend`, `--graph`, `--build`).
- `prove_with_fallback/4` — resolve pass only (no ordering pass), used by layered tests (`resolver:test`, `orderer:test`) and `--bugs`. Each test layer adds its own stages on top.

9.11.1 How `assuming/2` works

`prover:assuming(Flag, Goal)` stores a dynamic flag (`prover_assuming_<Flag>`) for the duration of `Goal`, using `setup_call_cleanup` to guarantee cleanup even on exceptions. Domain predicates test this flag with the unary `prover:assuming(Flag)`:

- **`candidate:eligible/1`** — when `keyword_acceptance` is active, candidates with any keyword are accepted; when `unmask` is active, masked candidates pass.
- **`acceptance:accepted_keyword_candidate/7`** — two fallback clauses widen the candidate pool: one for unstable keywords, one for masked packages.
- **`candidate:assume_blockers/0`** — returns `true` when blocker constraints should become warnings instead of hard failures.

9.11.2 Suggestion tags

When a relaxation flag is active and a candidate is admitted under that relaxation, the domain tags the literal's context with a `suggestion/2` term that records exactly which configuration change would eliminate the need for the relaxation:

Suggestion tag	Meaning	Target file
`suggestion(accept_keyword, '~amd64')`	Accept the unstable keyword	`package.accept_keywords`
`suggestion(unmask, R://E)`	Unmask the package	`package.unmask`
`suggestion(use_change, R://E, Changes)`	Adjust USE flags	`package.use`

These tags flow through the proof into the plan output. In builder mode, `builder:dispatch_suggestions/3` can apply the suggestions automatically (writing to `/etc/portage/package.*` files); in pretend mode, they appear as actionable hints in the plan output.

9.11.3 Formal guarantee

Each tier still produces a **complete proof** — the plan is always coherent and fully ordered. The relaxation only widens the candidate pool; it does not skip proof obligations or bypass constraint guards. The suggestion tags make it possible to **trace back** every relaxation to a concrete configuration change, so the weaker proof can be strengthened incrementally.

Once the proof is complete — whether strict or under assumptions — the results must be communicated to the user. The assumption printing pipeline inspects the Proof AVL and translates each assumption into a classified, actionable message.

9.12 Assumption printing pipeline

After the proof is complete, the printer walks the Proof AVL and collects every entry that represents an assumption. Assumptions fall into two families, and the printer handles them differently.

Domain assumptions are stored under `rule(assumed(X))` keys. These represent situations where the prover could not find a real rule and had to accept the literal on faith. When the printer encounters one, it inspects the literal and its context to classify the assumption and produce a meaningful message:

- **REQUIRED_USE violation** — the context contains a `required_use_violation(...)` tag (see section 9.8.1). The printer emits a structured block showing which USE flags were forced, which `REQUIRED_USE` expression they violate, and which parent caused the conflict.
- **Non-existent dependency** — the literal is a `grouped_package_dependency` without context. This means no candidate version exists at all (the category/name is not in the repository).
- **Grouped dependency with reason** — the literal is a `grouped_package_dependency` with an `assumption_reason` tag in its context. The printer extracts the reason label (e.g. "all candidates masked", "blocker conflict") and shows it alongside the dependency.

- **Model unavailable** — the context contains `issue_with_model`. The dependency model could not be built (e.g. all `any_of_group` branches were filtered). The printer reports this as a metadata problem.
- **Generic** — any domain assumption that does not match the above patterns is printed with the raw literal for debugging.

Cycle-break assumptions are stored under `assumed(rule(X))` keys. These mark points where the prover broke a dependency cycle by assuming a literal that was already being proved. The printer delegates to `cycle:print_cycle_explanation`, which reconstructs the cycle path and explains which packages form the loop.

9.12.1 Assumption type classification

The classification logic lives in `assumption.pl`. Given an assumption literal and its context, it returns a type tag that the warning printer uses to select the appropriate output format:

Pattern	Type tag
Context contains `required_use_violation`	`required_use_violation`
`grouped_package_dependency` (no context)	`non_existent_dependency`
`grouped_package_dependency` (with context)	Extracted from `assumption_reason`
`R://E:install`	`assumed_installed`
`R://E:run`	`assumed_running`
Blocker literal	`blocker_assumption`
Context contains `issue_with_model`	`issue_with_model`

The combination of learned constraints, constraint guards, and progressive relaxation is reminiscent of a well-known technique from the SAT solving world.

9.13 Conflict-driven clause learning connection

The learned constraint store is analogous to CDCL (Conflict-Driven Clause Learning) in SAT solvers, but expressed as version domains rather than boolean clauses:

CDCL concept	portage-ng equivalent
Conflict analysis	Constraint guard detecting domain inconsistency
Learned clause	`prover:learn(cn_domain(C,N,S), NarrowedDomain, _)`
Unit propagation	`grouped_dep_effective_domain` applying learned domains
Restart	`prover_reprove` catch-and-retry loop
Decision level	Reprove attempt number

The key difference is granularity: CDCL operates on boolean variables, while portage-ng operates on version domains — structured sets that carry more information per constraint.

With the proving and output mechanisms described, the following sections cover practical aspects: a testing checklist to catch regressions, and a source file map for navigating the codebase.

9.14 Testing learned constraints

When testing changes to the reprove or assumption mechanism, the following checklist helps catch regressions quickly:

- **Exit code** — the process exit code summarises the proof outcome:
  - `0` — no assumptions at all (clean proof)
  - `1` — only prover cycle-break assumptions
  - `2` — at least one domain assumption (e.g. missing dependency)
- **"Total: N actions"** — this line must appear in the output, confirming that the proof completed and a plan was produced.
- **"non-existent" count** — count the lines containing "non-existent" to check how many domain assumptions were made. An unexpected increase signals a regression.
- **No "Unknown message"** — the output should not contain "Unknown message" or unhandled exception traces. These indicate an assumption type that the printer does not recognise.
- **Runtime** — a single-target proof should complete within a few seconds. A significant increase compared to previous runs suggests excessive reprove retries or a learning bug.
- **Test suite** — the overlay and portage test suites (`resolver:test_stats/1`) should maintain their previous pass rate. Any drop indicates that a change has broken handling of a known edge case.

9.15 Source File Map

File	Role
`Source/Pipeline/prover.pl`	Core proof engine, reprove retry loop, cycle detection, learned store
`Source/Domain/Gentoo/Rules/resolving.pl`	Domain rules: entry rules, grouped deps, `rule(assumed(_),[])`
`Source/Domain/Gentoo/Rules/Resolving/candidate.pl`	Candidate selection and eligibility
`Source/Domain/Gentoo/Rules/Resolving/cnselect.pl`	Constraint guards, reprove triggers, parent narrowing, learned domains
`Source/Domain/Gentoo/Rules/Resolving/heuristic.pl`	Reprove state management, reject accumulation
`Source/Domain/Gentoo/Rules/Resolving/memo.pl`	Thread-local caches including `requse_violation_/3`
`Source/Domain/Gentoo/Rules/Resolving/use.pl`	`verify_required_use_with_bwu`, `describe_required_use_violation`
`Source/Pipeline/Prover/explanation.pl`	`assumption_reason_for_grouped_dep` diagnosis
`Source/Pipeline/Prover/explainer.pl`	`term_ctx/2`, "why" queries
`Source/Pipeline/Printer/Plan/assumption.pl`	Assumption type classification
`Source/Pipeline/Printer/Plan/warning.pl`	Assumption detail rendering

9.16 Further reading

- Chapter 8: The Prover — the proof search algorithm
- Chapter 10: Version Domains — domain operations used by constraint learning
- Chapter 12: Resolution — Configuration as Proofs — entry rules, fallback chains, and REQUIRED_USE handling
- Chapter 23: Resolver Comparison — Zeller, Vermeir, and CDCL foundations

Chapter 10

Version Domains

Version domains are the mechanism by which portage-ng reasons about version constraints. Every version comparison, every dependency operator (`>=`, `<=`, `~`, `=*`), and every learned constraint is expressed as an operation on version domains.

10.1 Why version domains matter

Picture two packages that both pull in `dev-libs/openssl`, but with different requirements. Package A depends on `>=dev-libs/openssl-3.0`: any OpenSSL from 3.0 upward is acceptable on that path. Package B depends on `<dev-libs/openssl-3.2`: only versions strictly below 3.2 are acceptable there. If both constraints apply to the same install, you are not looking for a single magic number first — you are asking which versions lie in the overlap of two sets. Versions that satisfy both are exactly those in **3.0 ≤ v < 3.2**: the half-open interval **[3.0, 3.2)**.

That overlap is the **intersection** (in domain terms, the **meet**) of two version domains. Version domains represent **sets** of acceptable versions; combining constraints from different dependency paths means intersecting those sets until you obtain the tightest description still compatible with everything seen so far. The rest of this chapter spells out how those sets are stored, compared, and merged in code.

10.2 Version representation

Versions are stored as `version/7` compound terms:

```
version(NumsNorm, Alpha, SuffixRank, SuffixNum, SuffixTail, Rev, Full)
```

Field	Example	Meaning
`NumsNorm`	`[3,0,77]`	Normalized numeric components
`Alpha`	`''` or `'a'`	Alpha suffix (empty atom if none)
`SuffixRank`	`4`	Numeric rank of first version suffix (`_alpha`=0, `_beta`=1, `_pre`=2, `_rc`=3, (final)=4, `_p`=5)
`SuffixNum`	`0`	First suffix number (e.g. `3` in `_rc3`)
`SuffixTail`	`[]` or `[s(5,2),s(4,0)]`	Remaining suffixes as `s(Rank, Num)` pairs, closed by the `s(4,0)` "(final)" terminator (`[]` if the version has no suffix)
`Rev`	`3`	Revision number (from `-r3`)
`Full`	`'3.0.77-r3'`	Original version string

Empty or absent versions use the atom `version_none`.

10.3 Version comparison

Versions are compared using Prolog's standard `compare/3` directly on the compound term. No runtime key conversion is needed — the `version/7` structure is designed so that standard term ordering produces correct PMS version ordering:

```
compare(Order, version([3,0,77],...), version([3,1,0],...))
% Order = (<)
```

This works because: - `NumsNorm` is a list of integers (lexicographic list comparison) - `SuffixRank` maps suffixes to integers in PMS order - `SuffixTail` lists remaining suffixes as `s(Rank, Num)` pairs ending in the `s(4,0)` "(final)" terminator, so multi-suffix versions compare pairwise per PMS (e.g. `1_rc1_p2 > 1_rc1_pre1` and `1_rc1 > 1_rc1_pre1`) - `Rev` is a plain integer

10.4 Version domain model

A version domain represents a set of acceptable versions for a package. It is stored as:

```
version_domain(Slots, Bounds)
```

Field	Type	Meaning
`Slots`	list or `any`	Acceptable slots (or `any` for unconstrained)
`Bounds`	structured term	Version bounds (upper, lower, exact, wildcard)

The `none` atom represents an unconstrained domain (all versions accepted).

10.5 Domain operations

10.5.1 Domain meet (intersection)

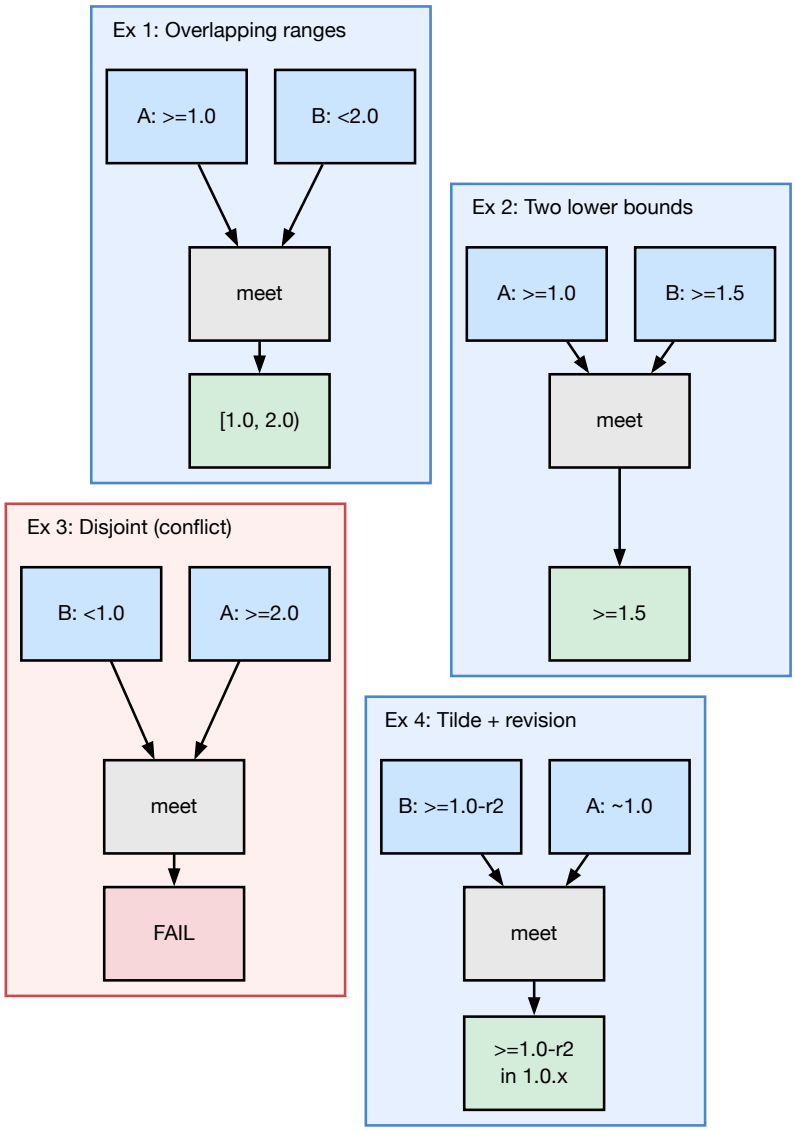


Figure 30 — Domain meet examples

When two dependency paths impose different version constraints on the same package, the domains are intersected:

```
version_domain:domain_meet(Domain1, Domain2, Intersection)
```

The intersection computes the tightest bounds that satisfy both constraints. If the intersection is empty (no version satisfies both), the goal fails: there is no `Intersection` term that stays consistent with the combined bounds (and related checks such as slot compatibility).

10.5.2 Worked examples (meet in practice)

The following sketches use dependency-style wording; internally each side becomes a `version_domain/2` (or `none`) whose bounds are merged by `domain_meet/3`. Intuitively, **meet = AND** over acceptable versions.

Example 1 — overlapping range.

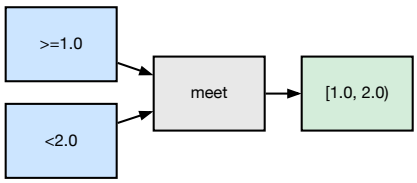


Figure 31 — Meet example 1

Lower bound `>=1.0` and upper bound `<2.0` describe the half-open interval **[1.0, 2.0)**. The meet is that interval: every version that satisfies both operators is exactly a version at least 1.0 and strictly below 2.0.

Example 2 — two lower bounds.

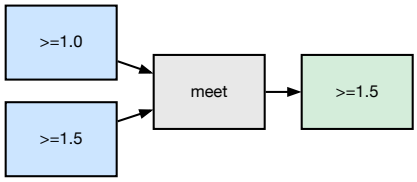


Figure 32 — Meet example 2

`>=1.0` meets `>=1.5`. The stricter requirement wins: the intersection is `>=1.5`. Anything below 1.5 was already ruled out by the second constraint.

Example 3 — disjoint constraints (conflict).

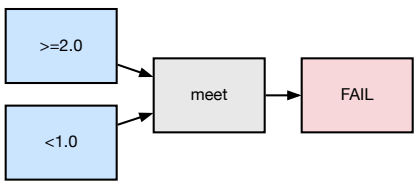


Figure 33 — Meet example 3

`>=2.0` meets `<1.0`. No version can be simultaneously at least 2.0 and below 1.0. `domain_meet/3` **fails**: the normalised domain is empty, and the prover must treat this as an unsatisfiable combined requirement.

Example 4 — tilde vs revision-aware lower bound.

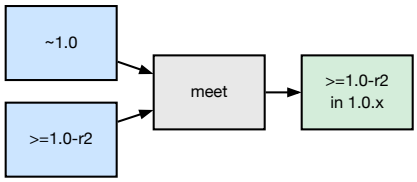


Figure 34 — Meet example 4

`~1.0` (in PMS terms: same main version as `1.0`, any revision) restricts candidates to the **1.0** line. Meeting that with `>=1.0-r2` keeps you on that line but drops revisions below `-r2`: the effective set is **`>=1.0-r2` within the 1.0.x family**, not a broader branch of the package.

In all cases, when the meet succeeds, candidate selection and consistency checks use the **narrower** domain; when it fails, there is no shared non-empty set of versions to choose from.

10.5.3 Consistency check

```
version_domain:domain_inconsistent(Domain)
```

Detects structurally empty domains — an empty slot set, conflicting exact bounds, or a lower bound above an upper bound. The meet operation rejects a result when this check succeeds (`\+ domain_inconsistent(...)`); it is purely structural and never consults the repository.

10.6 Feature logic intuition

In Zeller-style **feature logic**, a *feature* describes a set of objects that share certain properties — not necessarily a single object, but a well-bounded *set* described by those properties. A **version domain** is the same idea at the version level: it is a feature whose extension is the set of versions that satisfy given slot and bound constraints (above a threshold, below a threshold, exact match, tilde range, wildcard prefix, and so on).

Feature unification — meeting two features so that an object must satisfy both — corresponds directly to **domain intersection**. This is not merely a pedagogical analogy: portage-ng wires version domains into the generic unification hook in `feature_unification:val_hook/3`, so that merging domain values follows the same meet operation as `version_domain:domain_meet/3`.

A practical consequence is **monotonic narrowing**: along a resolution path, domains only become *tighter* (fewer acceptable versions), never wider, unless explicitly reset by a broader reprove strategy. Each successful refinement shrinks the search space; that is why successive reprove attempts can be viewed as making measurable progress toward either a concrete choice or a clear conflict.

10.7 Learned domain narrowing

The prover's learned constraint store uses version domains to carry narrowed version information across reprove retries. Each learned constraint is keyed by a category–name–slot triple:

```
cn_domain(Category, Name, Slot)
```

10.7.1 Conflict detection and learning

When two dependency edges impose incompatible version requirements on the same package, the constraint guard detects an empty intersection and records a narrowed domain for future attempts.

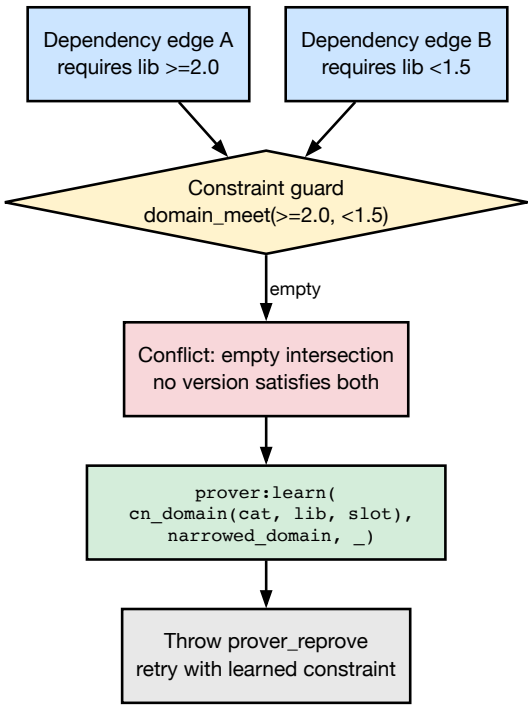


Figure 35 — Conflict detection and domain learning

The guard calls `domain_meet/3` on the two incoming domains. When the result is empty (no version satisfies both), the guard stores the narrowed domain via `prover:learn/3` and throws `prover_reprove` to restart the proof with this new knowledge.

10.7.2 Applying learned domains on reprove

On the next reprove attempt, `grouped_dep_effective_domain` intersects the learned domain with the local domain coming from the current proof context, *before* candidate selection begins.

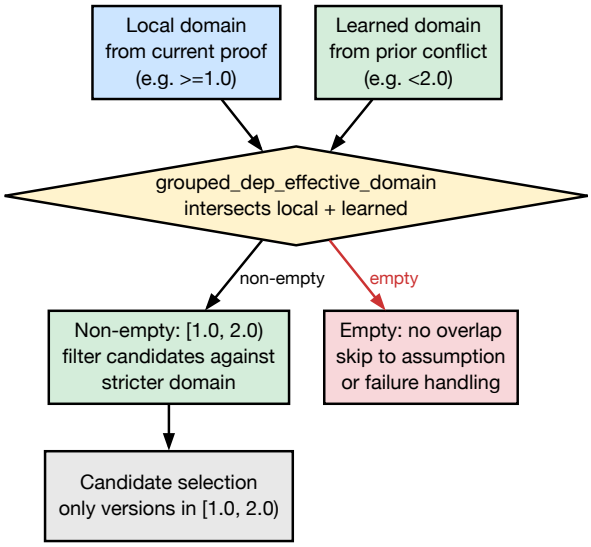


Figure 36 — Applying learned domain on reprove

If the intersection is **non-empty**, candidates are filtered against the stricter combined domain — avoiding the same dead-end choice that caused the previous conflict. If the intersection is **empty**, there is no compatible overlap left: the prover can skip directly to assumption or failure handling instead of selecting candidates from an empty domain.

Chapter 9 walks through the reprove and learning mechanics in full; Chapter 12 ties domains into resolve-time rule evaluation and candidate generation.

10.7.3 Wildcard domain learning

When a wildcard dependency like `=dev-python/gast-0.6*` cannot be satisfied, `cnselect:maybe_learn_wildcard_domain/4` derives an upper-bound domain from the wildcard: the last numeric component is incremented to produce an exclusive upper bound (e.g. `0.6*` → `< 0.7`, `1.2.3*` → `< 1.2.4`). The resulting `version_domain(any, [bound(smaller, UpperVer)])` is learned via `prover:learn/3` and triggers a reprove.

This mechanism is guarded: it only fires when the parent has already been narrowed in this proof (`cnselect:cn_previously_narrowed/2`: a learned domain exists, or a prior `maybe_learn_parent_narrowing` attempt rejected one of its entries) or when the parent is a single-version package, making parent narrowing futile. The guard ensures parent narrowing gets priority for multi-version parents, correctly handling cross-package wildcard conflicts (e.g. two packages requiring different wildcard ranges of the same dependency).

10.7.4 Connection to feature logic

This mechanism is inspired by Zeller's feature logic: version sets are identified by feature terms and refined by incrementally narrowing the set until each component resolves to a single version. Each successful `learn` call tightens the feature, and each `grouped_dep_effective_domain` call applies the accumulated tightening — a direct realisation of monotonic domain narrowing.

10.8 Version operators

The EAPI grammar supports the following version operators, each producing a different domain constraint:

Operator	Syntax	Domain meaning
`>=`	`>=cat/pkg-1.0`	Lower bound: version >= 1.0
`<=`	`<=cat/pkg-2.0`	Upper bound: version <= 2.0
`>`	`>cat/pkg-1.0`	Strict lower bound
`<`	`<cat/pkg-2.0`	Strict upper bound
`=`	`=cat/pkg-1.0`	Exact version match
`~`	`~cat/pkg-1.0`	Version match ignoring revision (any `-rN`)
`=*`	`=cat/pkg-1*`	Wildcard: any version starting with `1`

10.9 Further reading

- Chapter 9: Assumptions and Constraint Learning — how domains interact with the reprove mechanism
- Chapter 12: Resolution — Configuration as Proofs — how version domains feed into candidate selection
- Chapter 23: Resolver Comparison — Zeller's feature logic, CDCL connections, and pigeonhole-style slot allocation compared across resolvers

Chapter 11

Rules and Domain Logic

11.1 Prover and domain

The prover (Chapter 8) works with abstract literals and rules. It does not know what `:install` means for Gentoo — it only knows how to find a matching `rule/2` clause and prove its body. The **rules layer** is the bridge between that abstract proof search and a concrete domain: ebuilds, USE flags, version constraints, planning laws, or uninstall claims.

When the prover encounters a literal, it calls into whichever rule module it was given:

```
prover:prove(Rules, ...)
prover:prove_once(Rules, ...)
```

`Rules` is a module atom — typically `resolving`, `ordering`, or `unmerging`. The prover expands `Rules:rule(Head, Body)` and treats `Body` as the next obligations. It never interprets what the literals mean; all domain knowledge lives inside the rule clauses and the hooks they register.

That separation is deliberate. The same proof engine answers three different questions by loading a different rule module:

Question	Rule module	Stage wrapper
What configuration satisfies the request?	`resolving`	`resolver:resolve/9`
When can each action run?	`ordering`	`orderer:order/4`
In what order may packages be removed?	`unmerging`	depclean / unmerge pass

Callers outside a proving pass (for example query-side model construction) fall back to `config:default_rules/1`, which is `resolving`.

11.2 The `rule/2` contract

Every rule module exports the same interface:

```
Module:rule(+Head, -Body)
```

The prover passes a literal as `Head`. The module returns a list of sub-literals `Body` that must be proved to justify it. Failure means “this expansion does not apply”; success commits those

obligations to the proof search. Domain assumptions appear as `rule(assumed(X), [])` — an empty body that records a justified gap rather than aborting the proof.

Because the contract is uniform, Gentoo-specific vocabulary never enters the prover core. A new concern is almost always a new clause (or a new rule module), not engine surgery in `prover.pl`.

11.3 Rule modules in the pipeline

The pipeline chains two proving passes over one goal set (Chapter 4, Chapter 8):

1. **Pass 1 — configuration.** `resolver:resolve/9` hands `resolving` to `prover:prove/10`. Choice lives here: versions, slots, USE flags, OR-group arms. The output is a Proof / Model / Constraints / Triggers quadruple — a justified configuration.
2. **Pass 2 — plan.** `orderer:order/4` hands `ordering` to `prover:prove_once/…` over generic planning laws plus Gentoo bindings. The output is a second proof object; wave projection reads it into a parallel plan.

Depclean's uninstall order reuses the same planning laws with `unmerging` bindings: steps are `:unmerge` actions, requirements are claim releases, and the installed world provides no escape hatch.

So "rules and domain logic" is not a synonym for dependency resolution. Resolution is one rule set. Ordering and unmerging are others. The chapters that follow treat the two constructive passes as peers:

- Chapter 12: Resolution — Configuration as Proofs — pass 1: the `resolving` rule set and Gentoo policy
- Chapter 13: Ordering — Plans as Proofs — pass 2: planning laws, Gentoo bindings, wave projection

11.4 What a rule body carries

Rule bodies are more than flat dependency lists. They thread **proof-term context** `(?{…})` lists on literals — see Chapter 5 and Chapter 22): `build_with_use/1`, `constraint/1`, `after/1`, slot information, and suggestion tags for assumptions. Context is how parent requirements and local policy meet without the prover understanding Gentoo.

Heads themselves encode domain speech acts. For `resolving`, typical patterns include user `target/2` literals, action literals (`Repo://Ebuild:install`, `:run`, `:download`), grouped dependency atoms, REQUIRED_USE validation literals, and the `assumed/1` catch-all. For `ordering`, the language shrinks to `scheduled/1`, `available/2`, and `assumed(unreachable/2)`. The full head tables live in the pass-specific chapters.

11.5 Domain hooks at the prover boundary

Besides `rule/2`, the domain may answer prover callbacks through `heuristic:*` hooks (implemented for Gentoo under `Rules/Resolving/heuristic.pl` and consulted during prove):

- `proof_obligation/4` — inject derived obligations after a literal succeeds (PDEPEND is handled this way in a single resolve pass).
- `cycle_benign/2` — classify a proof-search cycle as benign before the prover records a cycle-break assumption.
- **Constraint guards / reprove helpers** — learn domains or request a reprove when selected versions conflict (see Chapter 9).

Hooks keep cross-cutting behaviour out of the generic search loop while still letting the domain steer search. Resolution and ordering each rely on them differently; the mechanism is shared.

11.6 Where the Gentoo resolve rules live

The `resolving` module is the public entry point (`Source/Domain/Gentoo/Rules/resolving.pl`). Its implementation is split across focused submodules under `Source/Domain/Gentoo/Rules/Resolving/` — candidate selection, USE evaluation, ranking, CN selection, target resolution, and so on. The inventory and the end-to-end resolve narrative belong in Chapter 12.

The `ordering` and `unmerging` modules sit beside them under `Source/Domain/Gentoo/Rules/`. They bind the generic planning laws to Gentoo dependency classes and to VDB facts; see Chapter 13.

11.7 Twin framing: configuration proofs and plan proofs

A successful run produces two proofs, not one algorithm output with a post-pass:

- **Configuration as proof** — every chosen version, USE set, and dependency edge is justified by a `resolving` rule expansion (or an explicit domain assumption / cycle break).
- **Plan as proof** — every wave placement is justified by an `ordering` (or `unmerging`) expansion: a step is scheduled because its requirements are available from earlier steps or from the installed world.

Reading the handbook in that order — rules contract, then resolution, then ordering — matches how the pipeline itself thinks.

11.8 Further reading

- Chapter 8: The Prover — proof search, models, triggers, pipeline entry points
- Chapter 9: Assumptions and Constraint Learning — domain assumptions, cycle breaks, progressive relaxation
- Chapter 10: Version Domains — version constraint representation used by resolve
- Chapter 12: Resolution — Configuration as Proofs — the `resolving` rule set in depth
- Chapter 13: Ordering — Plans as Proofs — planning laws and wave projection

Chapter 12

Resolution: Configuration as Proofs

12.1 Configuration as a first proving pass

The prover (Chapter 8) answers a question about the **final world**: which packages, which versions, which USE flags? Pass 1 gives that *what* the same status that pass 2 gives the *when* (Chapter 13): **the configuration is a proof**.

`resolver:resolve/9` hands the `resolving` rule set to `prover:prove/10`. Every chosen candidate, every USE-conditional branch taken or skipped, every OR-group arm, and every domain assumption is justified by a `resolving:rule/2` expansion (or by an explicit prover cycle break). There is no separate "resolver algorithm" whose output must later be trusted — the Proof / Model / Constraints / Triggers quadruples *are* the justified configuration.

The rules contract and the multi-module picture are in Chapter 11. This chapter is the Gentoo resolve pass in depth: how a user target becomes an installed graph, how candidates and USE are chosen, and what happens when the rules layer must propose a configuration change.

12.2 How dependency resolution works (end-to-end)

A typical run starts with a user query like `sys-apps/portage`. The rules layer turns this into a `target/2` literal and resolves it to the best eligible candidate — the newest version that is not masked, has an accepted keyword, and satisfies any slot constraints. This resolution produces sub-literals that drive the rest of the proof.

The resolution then branches depending on the action:

- `:run` resolves runtime dependencies (RDEPEND). PDEPEND is handled in the same pass through the prover's proof-obligation hook (see Hooks).
- `:install` resolves build-time dependencies (DEPEND and BDEPEND) and attaches ordering markers (`after/1`) that express which packages must be installed before others.

Each dependency atom from the metadata becomes a `grouped_package_dependency` literal. The candidate selection machinery then applies version ranges, slot operators, keyword and mask policy, and any learned constraints from prior reprove attempts (see Chapter 9 and Chapter 10).

USE-conditional dependencies are included only when the condition holds in the effective USE set for that ebuild and path. For example, `ssl? ( dev-libs/openssl )` adds `dev-libs/openssl` to the body only if `ssl` is enabled; otherwise the branch is skipped entirely. When a parent requires particular flags on a child, those requirements propagate via `build_with_use` in the proof-term context (see USE flags in depth).

The prover walks this structure depth-first: each successful rule expansion adds literals to the proof and updates the model. When a rule fails, Prolog backtracks to try an alternative candidate or, ultimately, records an assumption.

12.3 The `resolving:rule/2` head patterns

The resolve pass uses the shared `rule/2` contract (Chapter 11) with Gentoo heads:

```
resolving:rule(+Head, -Body)
```

Target rules translate a user query into a concrete ebuild. Action rules (`:install`, `:run`, `:download`) expand an ebuild into its dependency obligations. Dependency rules resolve individual atoms to candidates. Validation rules enforce REQUIRED_USE constraints. The catch-all `assumed(X)` clause handles domain assumptions when no real rule applies.

Head pattern	Purpose
`target(Q, Arg):run`	Resolve a user target to a candidate ebuild (`--merge`, `--fetchonly`)
`target(Q, Arg):fetchonly`	Legacy fetch-only target (CLI `--fetchonly` proves `:run` instead)
`target(Q, Arg):uninstall`	Uninstall target resolution
`Repo://Ebuild:install`	Build and install an ebuild (DEPEND + BDEPEND)
`Repo://Ebuild:run`	Runtime availability (RDEPEND)
`Repo://Ebuild:download`	Fetch source archives
`Repo://Ebuild:fetchonly`	Legacy fetch-only ebuild action
`Repo://Ebuild:depclean`	Remove unneeded package
`grouped_package_dependency(...):Action`	Resolve a grouped dependency
`package_dependency(...):config`	Configure a single dependency
`exactly_one_of_group(...):validate`	Validate REQUIRED_USE `^^`
`any_of_group(...):validate`	Validate REQUIRED_USE any-of
`at_most_one_of_group(...):validate`	Validate REQUIRED_USE `??`
`assumed(X)`	Catch-all for domain assumptions

12.4 Candidate resolution

When the rules layer encounters a dependency, it must choose a concrete version of the target package. This process has three stages: eligibility filtering, version-ordered selection, and a fallback chain for when no candidate works.

12.4.1 Eligibility filtering

Before a candidate version is considered, `candidate:eligible/1` checks two things:

- **Masking** — is the ebuild masked by the profile or user configuration?
- **Keyword acceptance** — does the ebuild have an accepted keyword for the current architecture?

(Installed status is a separate check, `candidate:installed/1`, used by the entry rules' already-installed short-circuit.)

If a candidate fails these checks and no relaxation tier is active (see Chapter 9, Progressive Relaxation), the entry rule fails and Prolog backtracks to try the next candidate.

12.4.2 Version-ordered selection

`target:resolve_candidate/2` resolves a query to a specific `Repository://Ebuild` pair. Candidates are tried newest-first via `cache:ordered_entry/5`, so the prover naturally prefers the latest eligible version.

12.4.3 Dependency ordering within a group

Before proving the dependencies of a package, `ranking:dep_priority/2` sorts them so that tightly constrained siblings are proved first. This reduces greedy conflicts where an unconstrained sibling selects a version that later clashes with a tighter constraint:

BaseK	Constraint type	Example
1	Tight upper bound (range)	`>=1.0 <2.0`
4	Tilde constraint	`~dev-ruby/railties-8.1.1`
8	Wildcard constraint	`=dev-python/gast-0.6*`
999	Unconstrained	`dev-libs/openssl`

Lower keys are proved first. Slot specificity is folded into the base key via `min` — a fully slot-qualified dependency (slot + subslot) gets key 0 and outranks every tier above. The effect is that slotted, tilde and wildcard dependencies lock their `selected_cn` before unconstrained siblings pick a potentially conflicting version.

12.4.4 Self-dependencies and cross-slot handling

When a package lists itself as a build dependency (e.g. `antlr-tool:4` needing `antlr-tool:3.5` to bootstrap), the rules layer distinguishes **same-slot self-deps** from **cross-slot self-deps**.

Same-slot self-deps (same category, name, and slot as the parent) are treated as bootstrap dependencies: if the package is already installed, the dependency is satisfied; otherwise the rule fails so that backtracking can reach a bootstrap alternative.

Cross-slot self-deps (same category and name but a *different* slot) are treated as **regular dependencies** and resolved normally. This prevents model build failures when the cross-slot version is not yet installed.

12.4.5 Fallback chain

When every candidate for a grouped dependency has been tried and none succeeded, the rules layer activates a fallback chain before giving up:

- **Wildcard domain learning** — `maybe_learn_wildcard_domain` fires when a wildcard dependency (e.g. `=dev-python/gast-0.6*`) fails resolution and the parent has already been narrowed by a prior parent-narrowing attempt, or the parent is a single-version package (where parent narrowing would be futile). It derives an upper-bound `cn_domain` from the wildcard constraint (e.g. `< 0.7`) and learns it via `prover:learn/3`, then throws `prover_reprove`.
- **Parent narrowing** — `maybe_learn_parent_narrowing` records that the current parent version led to a dead end and throws `prover_reprove`, so the prover can retry with a different parent.
- **Domain reprove** — `maybe_request_grouped_dep_reprove` checks whether domain or constraint conflicts exist and, if so, triggers a reprove with learned constraints.
- **Domain assumption** — as a last resort, the rules layer emits `assumed(grouped_package_dependency(...))`. This records the failure as a domain assumption so the proof can still complete.

12.5 Cycles and how portage-ng handles them

Circular dependencies are a fact of life in the Portage tree. A language runtime may be packaged with tooling that itself depends on that runtime, creating a loop. The prover detects these cycles during its depth-first proof search: it keeps track of which literals are currently being proved, and if the same literal appears again while it is still on the stack, a cycle has been found.

Before breaking a cycle with an assumption, the prover asks the domain whether the cycle is **benign**. The hook `heuristic:cycle_benign/2` inspects the repeating literal and the cycle path. If the hook succeeds, the literal is treated as already justified and added to the model without a cycle-break assumption. If the hook fails, the prover records a cycle-break assumption (`assumed(rule(Lit))` in the proof, `assumed(Lit)` in the model). This is separate from domain assumptions introduced by `rule(assumed(X), [])`.

The benign classification is conservative and pattern-based. For example, cycles that pass through `:run` (RDEPEND paths) are often treated as ordering-style cycles rather than hard failures — mirroring how traditional resolvers tolerate certain cyclic patterns.

After the proof is complete, the ordering pass (Chapter 13) resolves cyclic portions of the graph by citing the installed world (VDB) where possible, so that the merge ordering respects the cycle structure. For more on proof search and assumptions, see Chapter 8 and Chapter 9.

12.6 USE flags in depth

USE flags play a central role in dependency resolution. They determine which dependency branches exist, which packages are eligible, and whether REQUIRED_USE constraints are satisfied.

12.6.1 Effective USE and conditionals

For each ebuild the rules layer computes an **effective USE set** — the final set of flags that are active for this particular proof path. USE-conditional dependencies like `ssl? ( dev-libs/openssl )` are evaluated against this set: if the flag is active, the dependency is included; otherwise it is skipped.

The key predicate is `use:effective_use_for_entry/3` (with the context wrapper `use:effective_use_in_context/3`), which computes the full effective USE set for an ebuild. Whether a USE-conditional group is active is decided by the `candidate:eligible(use_conditional(...))` clauses together with the `use_conditional_group` rules in `resolving.pl`.

12.6.2 `build_with_use`

When a parent dependency requires specific USE flags on a child (e.g. `dev-libs/openssl[threads]`), those requirements travel through the proof as `build_with_use` context annotations. They influence how the child's effective USE set is computed, ensuring that parent requirements are not silently ignored.

12.6.3 `REQUIRED_USE`

Gentoo's REQUIRED_USE expressions (e.g. `^^ ( gtk qt5 )` meaning "exactly one of gtk or qt5") are enforced through dedicated validation literals. If the active USE set violates a REQUIRED_USE expression, the rule fails and the prover backtracks to try another candidate or records an assumption (see Chapter 9, section 9.8).

12.6.4 Priority order

USE flags are resolved in priority order, highest priority first:

1. `build_with_use` from the parent's dependency context
2. **User configuration** (`/etc/portage/package.use`)
3. **Profile defaults**
4. **Ebuild IUSE defaults**

The most important consequence is that context wins over profile defaults: a `build_with_use` requirement from the parent can force or forbid a flag regardless of what the profile would normally choose. This is why two proofs for the same package can produce different USE sets — they arrive through different dependency paths with different context annotations.

12.6.5 Conflicts and backtracking

When USE-derived constraints conflict — for example, REQUIRED_USE fails, a conditional branch does not apply as expected, or an eligibility check fails — the relevant rule fails. The prover then backtracks: it tries another candidate version, another slot, or another branch of the search tree. If no alternative succeeds, the candidate layer records a domain assumption, often tagged with a suggestion for which `package.use` change would resolve the conflict (see Assumptions as proposals).

12.7 Choice groups

Gentoo's PMS defines three choice-group operators that constrain how many members of a set may be active at the same time. The rules layer maps each operator to a dedicated validation literal that the prover must satisfy as part of the proof:

Operator	Rule clause	Semantics
any-of (a b c)	`any_of_group(Deps):validate`	At least one must be satisfied
`^^ ( a b c )`	`exactly_one_of_group(Deps):validate`	Exactly one must be satisfied
`?? ( a b c )`	`at_most_one_of_group(Deps):validate`	At most one may be satisfied

If the validation literal fails (e.g. two members of an `exactly_one_of` group are both active), the prover backtracks to try a different USE configuration or candidate version.

When a disjunctive dependency group (`||`, `^^`, `exactly_one_of`) must *select* an alternative, the candidate layer ranks the members with `ranking:prioritize_deps_keep_all/3` and commits to the first arm that passes config checks (see Any-of (`||`) arm selection below). Profile-forced and installed preferences still dominate via `is_preferred_dep/2`, the leading signal of the `preference` criterion; USE_EXPAND target digits (`ranking:use_expand_target_key/2`) are its last signal and also feed `use_expand` — `llvm_slot_20` → `[20]`, `python_single_target_python3_13` → `[3,13]`, and so on.

12.8 Any-of (||) arm selection

Gentoo ebuilds often write `|| ( arm1 arm2 … )`. The PMS only requires that *at least one* arm is satisfiable; it does not say which arm to pick. Wrong order locks the prover onto a suboptimal (or incompatible) branch — for example cabal's `|| ( ( >=text-1.2.3 <text-1.3 ) ( >=text-2 <text-2.2 ) )` must prefer the text-2.x arm when that is the newest tree candidate (portage-ng#112).

12.8.1 Portage vs portage-ng entry points

	Portage	portage-ng
Mechanism	`dep_zapdeps` ordered `choice_bins` + intra-bin upgrade promotion (`lib/portage/dep/dep_check.py`)	`candidate:resolve(choice_group…)` → `ranking:prioritize_deps_keep_all/3` then first `any_of_config_dep_ok`
Structure	Nine preference bins (lists)	One declared criterion list (`ranking:choice_criter` compared lexicographically by standard order of terms, original index last
Graph reuse	Digraph `all_in_graph`	`selected_cn` snapshot (`snap_all`)

Ranking *is* the preference policy: after the cut commits, later arms are not tried unless the proof backtracks for another reason.

12.8.2 Preference criteria (most significant first)

Declared as `ranking:choice_criteria/1`; each criterion's value comes from `ranking:criterion_value/5` and `prioritize_deps_keep_all/3` compares arms criterion by criterion (`ranking:compare_preference/3`) using the standard order of terms — a *larger* value is preferred, and the original ebuild index breaks remaining ties (left-to-right). There is no weighted sum and no integer packing: booleans are `no @< yes`, counts are integers, versions are `version/7` terms (which compare as PMS versions), slots and USE_EXPAND targets are digit-group lists (`'3.10'` → `[3,10] @> [3,3]`), and `none` sorts below everything else. Adding a preference means adding a criterion to the list, not choosing a magnitude.

Criterion	Value	Intent	Emerge analogue
`license_ok`	`yes`/`no`	Prefer license-acceptable arms	Availability / license gate
`use_sat`	`yes`/`no`	Prefer arms needing no USE flip on the arm's best candidate	`preferred_*` vs `unsat_use_*`
`use_unmasked`	`yes`/`no`	Among USE-unsat arms, prefer flips that do not fight use.mask / use.force	`all_use_unmasked` (masked → `other`)
`preference`	`pref/6`	Installed / profile-preferred, `--favour` / `--avoid`, not self-CN, no forced upgrade, `-bootstrap`, USE_EXPAND target — in that order, as the six arguments of one compound	`preferred_installed` + favour
`snap_all`	`yes`/`no`	Prefer arms whose non-`virtual/` CNs are already in the proof snapshot	`all_in_graph`
`snap_admits`	`yes`/`no`	Prefer an arm whose version bounds admit the already-selected candidate. Vacuous when that CN is not selected yet (so `version` still picks the newest arm) and, for a `pkg:N` atom, when nothing is selected in slot `N`	Selected atom vs arm bounds
`slot`	digit list / `none`	Prefer higher explicit package slot (`pkg:N`) — only active when *all* arms target the same (C,N)	`want_update` / higher-slot promotion
`no_downgrade`	`yes`/`no`	Demote arms whose newest admitted version is below installed or snap-selected	`downgrade_probe` → `other`
`installed`	count	Prefer arms that reuse more installed CNs	`other_installed` / `_some` / `_any_slot`
`overlap`	count	Prefer arms that appear in several sibling [illegible] groups	Soft stand-in for minimize-slots pressure
`use_expand`	[illegible]	[illegible]	[illegible]
`build_index`	integer, ascending / `version_none`	[illegible]	[illegible]
`version`	`version/7`	[illegible] the same (C,N)	`has_upgrade and not has_downgrade`

Worked examples:

- `|| ( foo[a] foo[b] )` with `a` already effective → `use_sat` picks `foo[a]`.
- Cabal text ranges (above) → `version` picks the text-2.x arm when neither is selected. Once text-1.2.5.0 is selected, `snap_admits` keeps the 1.2 arm (portage-ng#123).
- `|| ( sys-devel/llvm:18 sys-devel/llvm:20 )` → `slot` prefers `:20`.
- An arm whose packages are already in `selected_cn` beats a fresh CN → `snap_all`.

The same pattern orders the *proof* of a package's dependencies: `ranking:dep_priority/2` sorts sibling deps by their position in the declared `ranking:tightness_classes/1` list (sub-slot pin, tight upper bound, tilde, slot pin, wildcard, any-same-slot, any-different-slot, unconstrained, …), taking the first class that applies, so the tightest constraint locks `selected_cn` before a looser sibling can pick a conflicting version.

`version` and `slot` are gated to same-CN groups because comparing newest tree versions — or highest slots — of *different* packages is meaningless and overrides ebuild order: `virtual/mta` would pick `notqmail` (a `-9999` live ebuild inflates its score) over the intended `nullmailer`, and `virtual/jdk` would pick source `dev-java/openjdk` over `openjdk-bin` (portage-ng#115, portage-ng#116). Ungated slot-ranking likewise flipped ruby-single choices: `( ruby:3.3 rubygems[ruby33] )` — the profile default, listed first — lost to `( ruby:4.0 rubygems[ruby40] )` because 4.0 > 3.3, scheduling the `~arch` ruby:4.0 slot plus a `ruby_targets_ruby40` USE toggle that emerge never takes (webkit-gtk build cluster, Aug-2026 tinderbox run). Emerge never version- or slot-ranks across CPs inside a choice; it falls back to ebuild order there.

`virtual/` atoms are skipped for `snap_all`, `installed`, and `no_downgrade` (Portage's zero-cost treatment of virtuals in those checks). Scores are computed once per arm per `prioritize_deps_keep_all/3` call, with a short-lived per-call cache for installed / reference-version lookups. Ranking must not walk the ProofAVL; it only sees the proof-context list and memo snapshots (see also Chapter 27).

12.8.3 What we deliberately do not implement

These Portage mechanisms are **design omissions**, not open bugs. The inductive prover and ordering pass already provide the effects they target.

Overlapping `||` DNF (`_overlap_dnf`) and `minimize_slots` / `new_slot_count`. Portage merges overlapping `||` groups that share a CP into DNF, then sorts bins by ascending new-slot count. portage-ng does not rewrite the dep tree into DNF: expansion is exponential in overlapping width and does not belong on the per-`choice_group` hot path. The prover already commits `selected_cn` / learned `cn_domain` across the proof; later `||` sites reuse those choices via `snap_all` and constraint guards — the same "prefer packages already chosen" effect without a cross-product. `overlap`, `installed`, and `snap_all` cover the common "don't pull a second redundant package" pressure. Remaining edge cases (two overlapping `||`s neither yet selected) are rare relative to cost; if tinderbox surfaces one, prefer a targeted heuristic over full DNF.

Virtual expand (`_expand_new_virtuals`). Portage expands new-style virtuals into a newest-first `||` of providers before zapdeps. portage-ng already resolves virtuals through the virtual-provider path and `candidates_prefer_proven_providers/5`, and skips `virtual/` in the scores

above. A second expand-into-`||` pass would duplicate that work and fight proven-provider reuse.

Circular-dep demotion inside `||`. Portage demotes arms that close a known cycle with the parent (or `--onlydeps` parent CP) into `other`. portage-ng handles cycles in the prover and the ordering pass (cycle-break assumptions, world citations, `unreachable` assumptions) — see Chapter 8, Chapter 9, and Chapter 13. Demoting at ranking time would second-guess cycle-break polarity and needs a parent circular map that the proof-context list does not carry.

Intra-choice `cp_map` slot consistency (Portage bug 600346). Portage keeps a per-choice CP→slot map so several atoms in one choice stay slot-consistent. portage-ng arms are usually a single atom or a same-CN `all_of_group` of version bounds; cross-CP multi-atom arms that need `cp_map` are uncommon. Slot consistency is enforced later by `selected_cn`, slot constraints, and constraint guards.

12.8.4 Validation

- PLUnit: `ranking_any_of_version_branch`, `ranking_any_of_preference_keys` in `Source/Test/Unit/rankingtest.pl`.
- Overlay suite (`||` / USE / slot cases) and tinderbox-ng compare on USE-dep `||` and llvm/gcc/python slot packages.

12.9 Slot operators

Dependency atoms can carry a slot operator that tells the rules layer how to handle multi-slot packages. A package like `dev-lang/python` may offer several slots (e.g. `3.11`, `3.12`), and the slot operator determines which slots are acceptable and whether a sub-slot change should trigger a rebuild of the dependent package.

Operator	Meaning	Context effect
`:SLOT`	Depend on a specific slot	Filters candidates to that slot
`:*`	Any slot is acceptable	No slot constraint applied
`:=`	Sub-slot rebuild trigger	Records the selected sub-slot; a change triggers rebuild
`:SLOT=`	Specific slot + rebuild	Combines slot filter with rebuild tracking

12.10 Blockers

A blocker dependency says that two packages cannot coexist. Gentoo distinguishes two strengths:

Type	Syntax	Behaviour
Soft blocker (PMS: *weak*)	`!cat/pkg`	The blocked package should not be present; never fails the proof — walked past, recorded, and reported as an actionable assumption when effective
Hard blocker (PMS: *strong*)	`!!cat/pkg`	The blocked package must not be present; the constraint guard fires immediately

Plan output says *soft* / *hard*; the rules name the atoms `weak` / `strong` after the PMS syntax. A soft blocker is the negative counterpart of an ordering preference — see the terminology box in Chapter 13 ("Dependency types and ordering strength").

Internally, blockers produce `blocked_cn` constraint terms. These are checked against `selected_cn` constraints by `selected_cn_not_blocked_or_reprove`: if the blocked package has already been selected elsewhere in the proof, the guard triggers a reprove so the prover can learn to avoid the conflicting combination (see Chapter 9, section 9.10).

12.11 Hooks

PDEPEND (post-dependencies) represent packages that should be present at runtime but are not required at build time. Unlike DEPEND and RDEPEND, they do not block the build — they are installed afterwards.

In portage-ng, PDEPEND is handled in a single pass inside the prover via the `heuristic:proof_obligation/4` hook. Whenever a literal is successfully proved, the hook checks whether the corresponding ebuild has PDEPEND entries. If it does, those entries are injected as additional proof obligations on the spot. This avoids a separate PDEPEND resolution pass and ensures that post-dependencies are part of the same proof and plan.

12.12 Assumptions as proposals

When strict resolution cannot satisfy every dependency, the rules layer records a **domain assumption** rather than giving up. From a user perspective, an assumption is not a dead end — it is a **proposal** for a configuration change.

The literal's proof-term context is annotated with **suggestion** tags that spell out exactly what to change. Common suggestions include:

- `suggestion(unmask, ...)` — unmask a package
- `suggestion(accept_keyword, ...)` — accept an unstable keyword
- `suggestion(use_change, ..., Changes)` — adjust USE flags

The printer collects these tags and shows them next to the assumption, so you can see which `/etc/portage` file to edit and what to put in it. The plan is still constructed as if the change had already been applied: the merge list is coherent under the stated proposal, and the output tells you which configuration changes would make it real.

For the full story on assumptions and constraint learning, see Chapter 9.

12.13 Rules submodules

The `resolving` entry point is not a single monolithic file. It is split across focused submodules under `Source/Domain/Gentoo/Rules/Resolving/`, each handling a distinct concern:

Module	File	Purpose
`acceptance`	`acceptance.pl`	Keyword, mask, and license acceptance; keyword-aware candidate enumeration
`candidate`	`candidate.pl`	Grouped-dep resolution pipeline, blocker matching, eligibility protocol
`cnselect`	`cnselect.pl`	CN-consistency: selected_cn reuse, CN-domain reject map, learned-domain narrowing
`dependency`	`dependency.pl`	Self-entry injection, USE-requirement collection, slot/BWU proof-context propagation
`featureterm`	`featureterm.pl`	Proof-context list helpers (`after/1`, strip `build_with_use`, etc.)
`heuristic`	`heuristic.pl`	Prover hooks: constraint guard, cycle classification, PDEPEND obligations, reprove state
`memo`	`memo.pl`	Thread-local caching declarations, `clear_caches/0`
`ranking`	`ranking.pl`	Dependency ordering (`dep_priority/2`), choice-group ranking, BWU memo seeding
`slotmeta`	`slotmeta.pl`	Slot canonicalization, restriction merging, constraint queries
`target`	`target.pl`	Target resolution, update/downgrade transactions, depclean, `--exclude` helpers
`use`	`use.pl`	USE evaluation, conditionals, `build_with_use`, newuse, REQUIRED_USE, BWU conflicts

12.14 Policy cards (declarative view)

This chapter walks **how** resolution proceeds. For a newbie-oriented view of **what** Gentoo policy requires — PMS meaning, literals, owning modules, and short invariants — start here:

- Policy cards hub
- Policy by example — curated overlay curriculum
- One-page map — `rule/2` head → schema → test → card

Prefer those cards when onboarding or reviewing a rules change; keep this chapter for the end-to-end narrative and `||` ranking detail.

12.15 Further reading

- Chapter 11: Rules and Domain Logic — the `rule/2` contract and how rule modules plug into the prover
- Chapter 8: The Prover — how the prover calls `rule/2` and builds the proof
- Chapter 9: Assumptions — the fallback chain, reprove mechanism, and progressive relaxation
- Chapter 10: Version Domains — how version constraints feed into candidate selection
- Chapter 13: Ordering — Plans as Proofs — pass 2
- Policy cards — declarative Gentoo policy surface

Chapter 13

Ordering: Plans as Proofs

13.1 Why parallel planning?

Traditional package managers (Portage, apt, and similar) typically expose a **sequential** plan to the user: install *A*, then *B*, then *C*. Even when the underlying resolver knows that *B* does not depend on *A*, the presented order is often a single linear timeline.

portage-ng takes a different stance: it produces **parallel** plans from the start. Wave 1 might download *A*, *B*, and *C* concurrently; wave 2 might install *A* while *D* is still downloading; wave 3 might install *B* and *C* together, and so on. This is **not** a post-processing optimization layered on top of a linear schedule. Parallelism falls out of the ordering proofs themselves: two actions may share a wave exactly when neither one's availability proof depends on the other.

On a multi-core machine with fast I/O, overlapping work this way can dramatically reduce wall-clock time compared to a strictly sequential narrative.

Planning is also at the **action** level, not the package level. The same logical package may appear as separate literals for download, install, run, and so on. Those actions can therefore land in different waves: one package can still be downloading while another is already installing, whenever the dependency graph allows it.

13.2 From proof to plan: a second proving pass

The prover (Chapter 8) answers a question about the **final world**: does a consistent solution exist — which packages, which versions, which USE flags? Its output is a proof, and every fact in that proof carries a justification.

But a proof is not a plan. To execute anything, the actions must be ordered in time. Earlier versions of portage-ng produced that ordering with procedural graph algorithms (Kahn's topological sort for the acyclic portion, Kosaraju SCC decomposition for cycles). The machinery worked, but its answers came with no justification: a package landed in wave 7 because "the algorithm said so", and diagnosing a mis-ordered plan meant archaeology across several procedural passes.

The ordering engine (`Source/Pipeline/orderer.pl`) gives the *when* the same treatment as the *what*: **it runs the same prover core a second time**.

- **Pass 1** proves a solution exists. It is the existing prover with the existing domain rules, completely unchanged. All *choice* lives here — versions, USE flags, OR-group selection.

- **Pass 2** constructs an ordering of that solution. The prover core is re-entered over a small set of generic **planning laws**, with the pass-1 proof and the installed system (VDB) as its facts. Its output is a second proof object in which every placement is justified.

The pass-2 proof reads the way a Linux From Scratch book reads: *fontconfig can be built at step 8 because python-3.14.6 is already installed; the new python is built at step 12*. A plan is no longer certified by empirical testing of an algorithm's output — the plan **is** a proof.

13.3 The planning laws

Pass 2 needs only a handful of generic laws. They are the `rule/2` clauses of the `ordering` module (`ordering.pl`, alongside the Gentoo bindings) and own no Gentoo vocabulary at all:

```
% A step can be placed once everything it requires is available:
rule(scheduled(H), Conds) :-
  ordering:step(H),
  findall(available(H, D), ordering:requires(H, D), Conds).

% A requirement is available when an earlier plan step provides it, or —
% failing that — when the world as it stands already provides it, or —
% failing that too — by recording the bootstrap failure as a negative
% domain assumption instead of failing the pass:
rule(available(H, D), Body) :-
  (   ordering:step(D),
      \+ prover:currently_proving(scheduled(D))
  ->  Body = [scheduled(D)]
  ;   ordering:world(H, D)
  ->  Body = []
  ;   Body = [assumed(unreachable(H, D))]
  ).

rule(assumed(unreachable(_, _)), []).
```

Three literals make up the entire pass-2 language:

- **`scheduled(H)`** — step *H* can be placed; its proof is the placement justification.
- **`available(H, D)`** — hard requirement *D* of step *H* is satisfiable in time: by an earlier plan step, or by the installed world.
- **`assumed(unreachable(H, D))`** — a **negative domain assumption**: no plan step and no installed package can provide *D* for *H*. This is the genuine bootstrap boundary, reported honestly instead of papered over.

The consumer *H* appears in the availability literal on purpose: whether a requirement can be bridged by the installed world depends on the consumer's position in the derivation (cycle membership), so availability proofs are never shared across consumers. `scheduled/1` proofs are position-independent and memoize globally through the prover's proven fast path — each step is scheduled once, no matter how many consumers cite it.

13.4 The Gentoo bindings

The laws ask three questions they cannot answer themselves: what is a step, what does a step require, and what does the world already provide. One domain file — `Source/Domain/Gentoo/Rules/ordering.pl` — answers them by reading the pass-1 proof and the VDB:

Binding	Question answered	Source
`step/1`	What are the plan's steps?	Pass-1 proof rule heads
`requires/2`	What must exist before a step?	Build-time deps (DEPEND/BDEPEND) in the step's pass-1 rule body, including a planned same-CN merge when the grouped `:install` is keep-installed, and the runtime providers of a `virtual/*` package being merged
`prefers/2`	What would we like earlier, without insisting?	Runtime deps (RDEPEND), PDEPEND completion, ordering hints
`world/2`	What does the system already provide?	VDB (installed packages)

This is the same split that keeps Gentoo semantics out of the prover core: the laws are the engine, the bindings are microcode loaded from disk. An ordering quirk in the tree becomes a rule edit in `ordering.pl`, never engine surgery in `orderer.pl`.

The orderer hands the `ordering` rule module directly to the generic prover (`prover:prove_once(ordering, ...)`), so the prover core itself needs no knowledge of which pass it is running — it just expands whatever rule set it was given.

13.5 Dependency types and ordering strength

Gentoo's dependency classes do not all impose the same ordering strength. The bindings translate each class into one of two kinds of edge, a hard requirement or a soft requirement (preference). Since the same two words recur elsewhere in portage-ng, here is what they mean everywhere:

> **Hard and soft: terminology**
>
> Throughout portage-ng *hard* means "enforced by the proof — the proof fails without it" and *soft* means "can never make a proof fail". The pair is used for three things:

	Hard	Soft
Ordering edge (pass 2)	**hard requirement**, `requires/2`: the plan is invalid unless the provider comes first — DEPEND/BDEPEND	**soft requirement** or **preference**, `prefers/2`: the plan is better if the provider comes first, valid if not — RDEPEND, PDEPEND completion, `order_after` / `schedule_after`, configure closure, assumed-dep alias
Blocker (pass 1)	**hard blocker**, `!!atom`: the constraint guard refuses any model in which the blocked package is selected	**soft blocker**, `!atom`: pass 1 walks past it and records `assumed(blocker(...))`; the printer reports the *effective* ones as an actionable domain assumption
USE dependency	unconditional `[flag]`: the provider must be built with the flag (a "HARD usedep" in the use-enable feedback code)	conditional `[flag?]`, `[flag=]`: follows the consumer's own USE state

A soft blocker is the negative counterpart of a preference — "I would rather this package were not here" against "I would rather this one came first" — and both are soft in the same sense. They differ in what happens when they are not satisfied: a preference on a loop is void and silent, because nothing is at stake; an effective soft blocker is reported, because something is — the blocked package is on the system, and Portage would unmerge it once the blocking package is merged.

PMS calls the two blocker kinds *weak* (`!`) and *strong* (`!!`), and the rules mirror that in their atom names (`grouped_package_dependency(weak|strong, ...)`; `--info` prints `[weak block]` / `[strong block]` when describing an ebuild's syntax). Plan output and this handbook say *soft* and *hard*, the words emerge users know from `[blocks b]` / `[blocks B]`. Strictly, PMS's distinction is one of timing — a weak blocker's target may coexist until the end of the batch, a strong blocker's never — and emerge's "soft" means "auto-resolvable by a later unmerge"; in practice the two coincide with `!`.

Constraint is a synonym for neither: it is reserved for the constraint store (Chapters 8–10), where `constraint/1` literals narrow the model. `constraint(order_after(...))` travels through that store but is a preference. Ordinary English "hard failure" or "hard timeout" carries none of these meanings.

The two ordering edge kinds are:

- a **hard requirement** (`requires/2`): the plan is invalid unless the provider comes first;
- a **soft requirement**, or **preference** (`prefers/2`): the plan is better if the provider comes first, but still valid if it does not.

- **DEPEND** and **BDEPEND** — build-time dependencies. They must be satisfied before the build can start, so they become `requires/2` edges: the consumer's `scheduled/1` proof waits on them. A keep-installed grouped `:install` is a wave-1 leaf (empty body: the installed copy still matches the version constraint). When a merge of the same category-name is *also* planned — typically a subslot rebuild — the binding aliases the group onto that planned action so the consumer cannot share a wave with the rebuild and configure against the stale install (perl 5.42→5.44 / `Syntax-Keyword-Try` before `XS-Parse-Keyword`). Assumed grouped deps stay a preference (portage-ng#95); this alias is the hard edge for the keep-installed path. Blocker groups never alias: `!dev-ml/findlib` in ocaml's BDEPEND names a package that must be *absent*, so it must not make ocaml wait for the planned findlib merge — that closed a hard cycle with findlib's own DEPEND on ocaml and left findlib configuring before ocaml was on `PATH` (portage-ng#119).
- **RDEPEND** — runtime dependencies. They must be satisfied before the package is *used*, not before it is built. They become `prefers/2` edges — soft requirements, honored when they lie on no loop (see "Preferences: soft requirements" below for the two `--optimize` strategies), never allowed to force a world bridge or an unreachable assumption. One exception, the **virtual collapse** (portage-ng#119): a `virtual/*` package has no build and installs no files, so a DEPEND/BDEPEND on it means the consumer needs the virtual's RDEPEND providers at build time — Portage collapses a GLEP 37 virtual's runtime deps onto the consumer with the consumer's dep priority. The pass-1 `:install` body of the virtual already names those providers as grouped `:run` heads; for a virtual being merged they are `requires/2` edges, so `dev-ruby/typeprof` waits for `dev-ruby/rubygems` through `virtual/rubygems` instead of co-waving with the empty virtual (the orderer-era recurrence of portage-ng#18). A preference would not do: it competes on equal footing with the PDEPEND completion pull of ruby's post-install group (which contains the virtual itself) and loses by processing order.
- **PDEPEND** — post-install dependencies. They are resolved inside the pass-1 proof (via `heuristic:proof_obligation/4`, see Chapter 8) and create no proof edge. The bindings add a **completion preference**: a consumer of a PDEPEND provider prefers to wait for the provider's post-install group, matching emerge's behaviour (portage-ng#18). The preference is dropped for consumers inside the provider's own PDEPEND cycle (portage-ng#19).
- **IDEPEND** — install-time dependencies (EAPI 8+). They constrain ordering around the install phase and flow through the same context machinery as DEPEND.

For the exact mapping from PMS ordering semantics to internal edges, see Chapter 24: Dependency Ordering.

13.6 Cycles: citing the installed world

Dependency cycles are where the rule-based engine differs most visibly from its predecessor. Consider the classic loop: python depends on tk at build time when built with `tk` support, tk depends on fontconfig, and fontconfig needs python to build.

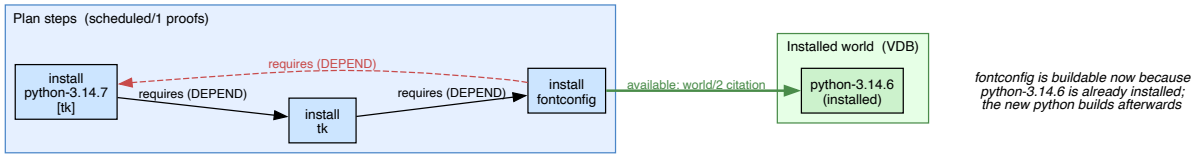


Figure 37 — Ordering a cycle through the installed world

When pass 2 proves `scheduled(fontconfig:install)` and reaches the requirement on python, the first clause of the availability law — "an earlier plan step provides it" — is refused: the guard `\+ prover:currently_proving(scheduled(D))` detects that python's own scheduling proof is still open on the derivation stack, i.e. citing it would close a loop. The law falls through to the next question: does the *world as it stands* provide python?

- **If an older python is installed**, `world/2` answers yes, and the proof records a **citation of the VDB entry**: *fontconfig is buildable now because python-3.14.6 is already installed*. This is exactly how Linux From Scratch reasons about its temporary toolchain — a fact about the present system, not a heuristic about graphs.
- **If nothing bridges the loop** (a bare system bootstrapping from nothing), the plan reports an honest `unreachable` assumption — the genuine bootstrap boundary — instead of an arbitrary cut.

Note what disappeared: there is no SCC decomposition, no merge-set post-pass, no progressive edge relaxation. A cycle is not a special case to be repaired after the fact; it is simply the situation in which the first clause of a law fails and the next one is consulted.

The pass-1 prover still records its own **cycle-break assumptions** (Chapter 9) — those concern the existence proof. Pass-2 world citations and `unreachable` assumptions concern the *ordering* and appear in the plan's assumption report separately.

13.7 Preferences: soft requirements

A preference is not a promise. Runtime-ish edges are collected separately from hard requirements and are folded into the plan **after** the hard structure is fixed; they can delay a step, never pull it earlier than its hard requirements allow. Whatever happens to the preferences, the hard structure — every build-time dependency merged before the package that needs it — is satisfied by every plan portage-ng prints.

The bindings currently derive preferences from six sources:

1. **RDEPEND groups** — a package prefers its runtime providers earlier.
2. **`order_after` hints** — ordering-only constraints recorded in proof context by the rules layer (see Chapter 5).
3. **`schedule_after` hints** (portage-ng#89) — plain anchoring for sub-slot ABI rebuilds: the rebuild alone goes after its changed provider. Unlike `order_after`, these are *not* indexed as a PDEPEND completion group, so the provider's other consumers do not wait for the rebuilds (which would serialize the plan).

4. **PDEPEND completion** (portage-ng#18/#19) — consumers of a PDEPEND provider prefer the provider's post-install group first.
5. **Configure closure** (portage-ng#21) — an `:install` action prefers the runtime providers of its `:run` sibling, so packages whose configure phase probes runtime tools are ordered correctly.
6. **Assumed-dep aliases** (portage-ng#95) — when a grouped dependency degraded to a domain assumption in pass 1 but a concrete action for the same package *is* planned, the consumer prefers that action.

Preferences can contradict each other. Two packages that each name the other as a runtime dependency both prefer to go second; a package whose PDEPEND group in turn depends on it prefers to be "complete" only after something that prefers to come after it. No order honors every preference in such a loop, so *some* preference has to go — and there are two reasonable policies for choosing which. They are selected on the command line with `--optimize`.

13.7.1 A loop of soft requirements: Nautilus and Sushi

In this worked example a soft requirement is called what it is in plain English: a *wish*. GNOME's file manager `gnome-base/nautilus`, built with `USE=previewer`, declares a PDEPEND on `gnome-extra/sushi`, the quick-look previewer: the file manager is only *complete* once Sushi is in place. Sushi in turn declares an RDEPEND on Nautilus: it wants a running file manager before it is merged. Neither package needs the other to *build*; both edges are soft requirements — wishes.

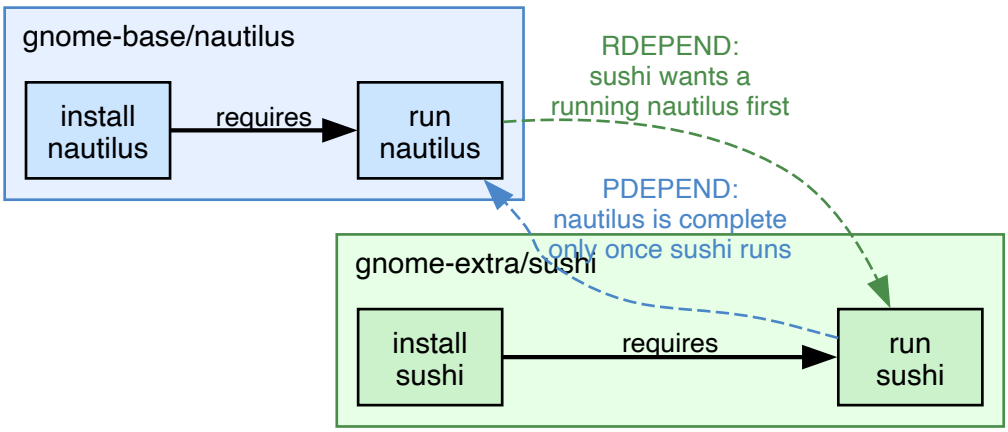


Figure 38 — Two wishes that cannot both be granted

Solid arrows are hard requirements (a package runs only once it is installed); dashed arrows are the two wishes. Follow the dashed arrows and you walk in a circle: `run nautilus` wants to come after `run sushi`, which wants to come after `run nautilus`.

13.7.2 `--optimize soft-requirements`: honor as many preferences as possible

Under this strategy every preference is handed to the wave projection, which accepts them one by one and keeps each exactly when it closes no cycle against the hard requirements and the preferences already accepted. In the loop above the first wish it meets is granted; the

second would close the loop and is dropped — silently and safely, matching how Portage treats runtime cycles as freely orderable.

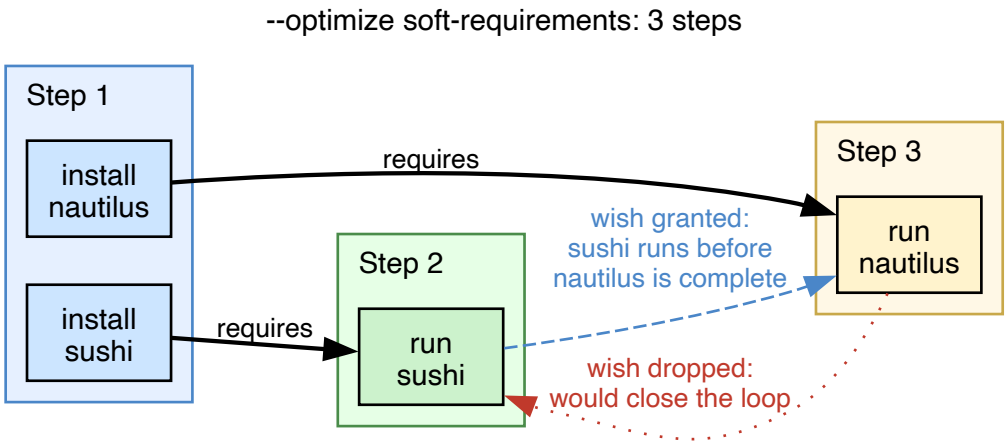


Figure 39 — soft-requirements: one wish granted, one dropped

The plan is three steps long: both installs, then Sushi's run, then Nautilus' run. One of the two wishes has been honored; which one depends on the order in which the projection happened to visit them. Larger loops pay more: with n packages preferring each other in a circle, the projection keeps $n - 1$ preferences and lines the members up one per step — a staircase of single-package steps in the Gantt chart. The listing is close to the linear order traditional `emerge` prints, which is what this strategy is for: a familiar-looking plan, and the most individual runtime orderings honored.

13.7.3 `--optimize parallelism` (default): a preference on a loop is void

Under the default strategy the decision is made in the rules, before the projection sees anything. A preference from H for D is void when H and D are *mutually reachable* — each depends on the other, directly or through other steps, counting hard requirements and preferences alike. Inside such a loop no order is better than another (there is nothing for the preferences to agree on), so all of them are void and only the hard requirements shape the plan; preferences between packages that are *not* on a common loop are kept, every one of them.

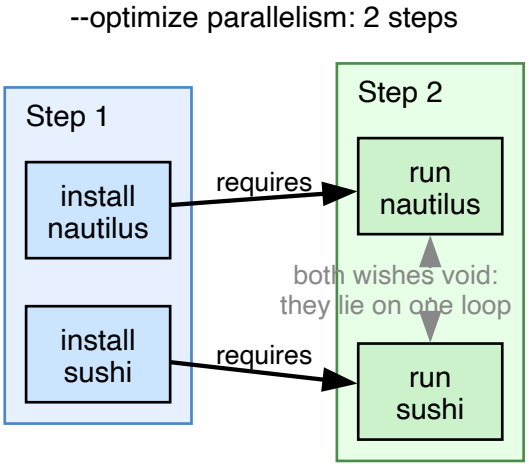


Figure 40 — parallelism: both wishes void

The same two packages now take two steps: both installs side by side, then both runs side by side. For a loop of n packages the saving is proportionally larger — the 22-package runtime loop under `app-xemacs/xemacs-devel` planned as 47 steps under `soft-requirements` and as 19 under `parallelism`; `dev-ruby/dalli`'s gem loop went from 85 to 21. Because a preference is void or kept on the basis of the graph alone, the outcome does not depend on visiting order, and the set of preferences that reaches the projection is acyclic by construction: the projection becomes a pure evaluator and its cycle branch is never taken.

The cost is the soft requirement itself. On the Nautilus/Sushi loop the `soft-requirements` plan did get Sushi in before Nautilus was declared complete; the `parallelism` plan makes no such attempt. For runtime dependencies this is harmless — the files are on disk either way, and nothing in the loop is *built* against anything else in it — but it is a real difference, which is why both strategies exist.

13.7.4 Hard requirements are never traded away

Not every loop is made of wishes. `media-sound/audacious` PDEPENDs on `media-plugins/audacious-plugins` (a wish: the player is complete once its plugins are in), while the plugins DEPEND on the player (a hard requirement: they are built against it).

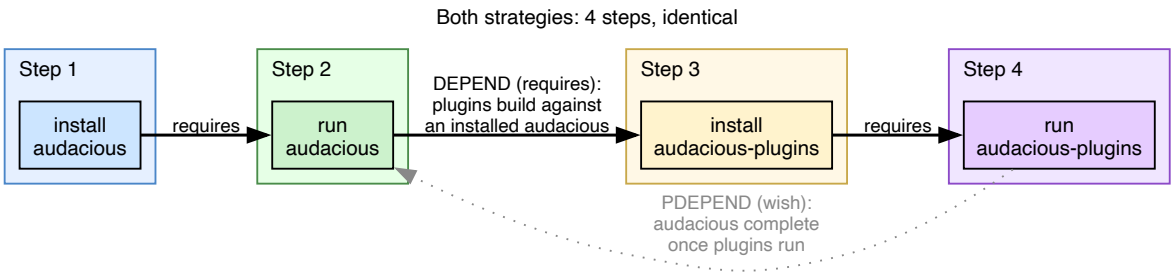


Figure 41 — A loop with a hard edge: both strategies agree

Here there is nothing to choose. The hard edge forces the player before the plugins under both strategies; the PDEPEND wish is the only edge that could give way, and it does — void under `parallelism`, dropped under `soft-requirements` — for the same reason: it points back along a path the hard edge already fixed. The plans are identical. The strategies differ only where a loop consists of soft requirements alone.

Two consequences follow for anyone comparing plans across strategies:

- **Plans never break.** The hard requirements are the same edges under both strategies; `parallelism` only ever withholds preferences. Every step in a `parallelism` plan is buildable when its wave is reached.
- **Loops are read off the declared edges.** Mutual reachability is computed over `requires` ∪ raw preferences *before* pass 2 bridges any hard cycle through the installed world (previous section). A preference that sits on a hard cycle the world later bridges is therefore void too; the hard edges of that cycle are still enforced.

Where the two strategies live in the code: the raw preferences are `ordering:prefers0/2`; `ordering:prefers/2` hands them on unchanged under `soft-requirements` and filters them through `ordering:same_component/2` under `parallelism`. The mutual-reachability classes come from a generic, domain-free library predicate (`components:classes/3` in `Source/Logic/components.pl`), built once per ordering pass over all plan steps and looked up by the rule — the ordering rules still contain no graph algorithm, and the cost is linear in the size of the plan.

Within a wave, actions are finally reordered by **merge-order bias**: the actions other packages wait on most (highest reference count in the Triggers AVL) are listed first, so the builder starts the most-blocking work as early as possible.

13.8 Wave projection and plan output

The wave-list plan is a **projection** over the pass-2 proofs — an evaluator, not a decider. Every ordering decision was already made (and justified) during the proving pass; the projection merely assigns wave numbers by reading availability proofs:

- a step whose requirements are all world-bridged or assumption-bridged can start in wave 1;
- a step that cites earlier plan steps lands one wave after the last of them;
- accepted preferences raise a step's wave further, never lower it.

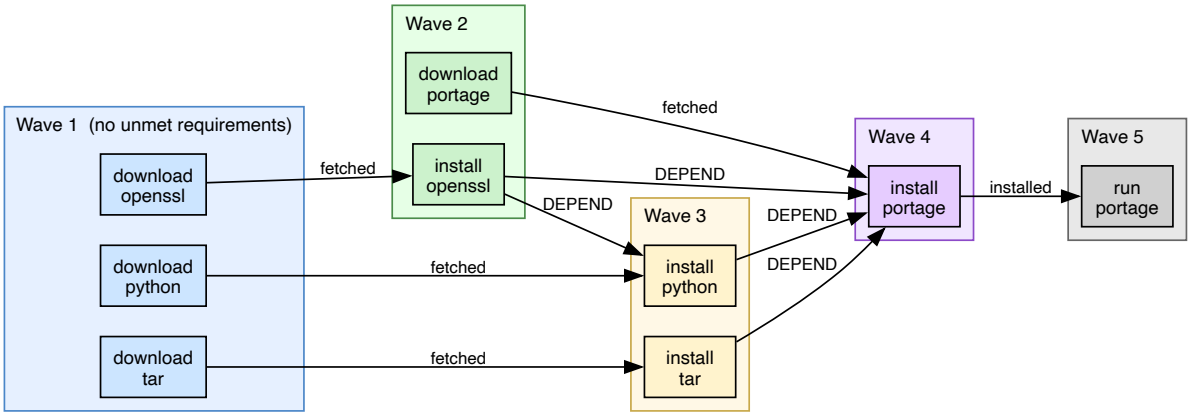


Figure 42 — Wave plan produced by the ordering pass

The output contract is unchanged from earlier releases: a list of waves, each containing full-format pass-1 rule terms. All actions within a wave are independent and can run concurrently. The printer renders the waves as numbered steps (Chapter 14); the builder executes them with real parallelism (Chapter 16). Neither consumer knows or cares that the waves are now backed by proofs.

The plan is annotated per entry with:

- **Wave number** — which parallel wave it belongs to
- **Action** — download, install, run, etc.
- **Literal** — the full `Repo://Entry:Action?{Context}` term

13.9 The same laws order uninstalls

Depclean's uninstall order is the same three laws proved over a different set of bindings (`Source/Domain/Gentoo/Rules/unmerging.pl`). A step is the `:unmerge` of a removable package; what a step *requires* is the release of every claim on it — each removable consumer must be unmerged first; and the *world* provides nothing, because an installed consumer's claim is a present fact in the VDB — there is no "already provided" escape like merge ordering has.

Cyclic claim chains fall through the same `currently_proving` guard and surface as **retained-claim assumptions**: the report names exactly which package still depends on which at its unmerge point, instead of a bare "cycle detected" flag. The wave projection is reused unchanged, and the flattened waves are the uninstall order (consumers first, dependencies last). Kahn's topological sort — the last procedural survivor of the pre-proof planner — was retired with this pass.

One binding detail is load-bearing: the claim index reads the VDB dependency models through the query layer, whose inlined model construction dispatches through the *active* rule module. The index is therefore prepared eagerly, before the unmerge prove scopes the rule module to `unmerging` (see `unmerging:with_unmerge_pass/2`).

13.10 Further reading

- Chapter 8: The Prover — how the Proof AVL is constructed
- Chapter 9: Prover Assumptions — pass-1 cycle breaking
- Chapter 11: Rules and Domain Logic — how rule modules plug into the prover
- Chapter 12: Resolution — Configuration as Proofs — pass 1: the configuration being ordered
- Chapter 14: Output and Visualization — how the plan is rendered
- Chapter 16: Building and Execution — how the plan is executed
- Chapter 24: Dependency Ordering — PMS ordering semantics

Chapter 14

Output and Visualization

After the prover completes and the ordering pass produces a parallel plan, the next step is to present the result to the user. This happens before any building — even a `--pretend` run produces the full plan output. portage-ng offers several output formats: a colour-coded terminal plan, `.merge` files for regression testing, interactive SVG dependency graphs, Gantt charts, and structured reports.

14.1 Terminal plan display

The most common output is the terminal plan, which resembles `emerge -vp` but adds parallel wave information and richer detail. The printer (`Source/Pipeline/printer.pl`) orchestrates the display, delegating to submodules under `Source/Pipeline/Printer/`.

14.1.1 Merge list

The plan is rendered as a merge list. Each line represents one action, printed as a numbered step with a coloured **action bubble** and the full package atom.

A typical plan fragment looks like this:

```
└─step 01─┤ install  portage://dev-libs/expat-2.6.4
           │ install  portage://dev-libs/libxml2-2.13.5

└─step 02─┤ install  portage://dev-lang/python-3.12.3
           │ run      portage://dev-lang/python-3.12.3
           │ confirm  portage://dev-lang/python-3.12.3

└─step 03─┤ update   portage://sys-apps/portage-3.0.77-r3
           │          (replaces portage-3.0.65)
           │ download https://.../portage-3.0.77-r3.tar.xz

└─step 04─┤ verify   dev-python/tree-sitter
           │          (non-existent, assumed installed)
```

Actions within the same step can run concurrently — the step number is the wave. Each line shows:

- **Step number** — which parallel wave this action belongs to.
- **Action bubble** — a full word indicating the operation:
  - `install` — new install
  - `update` — version upgrade (shows the replaced version)

  - `downgrade` — version downgrade (shows the replaced version)
  - `reinstall` — reinstall of the same version
  - `run` — runtime dependency check
  - `confirm` — verify that a running dependency is available
  - `download` — fetch source from a mirror
  - `verify` — assumed dependency that needs manual verification
- **Package atom** — repository, category, name, and version (e.g. `portage://dev-libs/openssl-3.1.4`).
- **Annotations** — contextual notes such as `(replaces ...)` for upgrades/downgrades, `(~amd64)` for keyword-accepted packages, `(USE modified)` for USE flag changes, or `(non-existent, assumed installed)` for unresolvable dependencies.

`--fetchonly` / `-F` does not introduce a separate plan action. It proves the same `:run` plan as `--merge` and hides merge-phase bubbles (install / run / update / …), leaving downloads and configuration pre-actions. See Chapter 15.

Target packages — the ones you explicitly asked to prove — appear in **bold green** with a green action bubble. Non-target dependencies use cyan text. Assumed or unresolvable dependencies use yellow or red bubbles to draw attention.

14.1.2 Printing styles

portage-ng supports three printing styles, selectable via `config:printing_style/1` (default: `fancy`):

- **`fancy`** (default) — the bubble style: a compact visual layout where each action line includes colour-coded indicators and right-edge annotations.
- **`column`** — a tabular layout that aligns version, slot, USE, and repository information in fixed columns for easy scanning.
- **`short`** — a minimal one-line-per-action format.

14.1.3 Powerline bubbles (MesloLGS NF)

In `fancy` mode, action labels are drawn as **Powerline bubbles**: a coloured pill with rounded caps (`U+E0B6` / `U+E0B4`). Those code points live in the Private Use Area, so a stock monospace font such as Menlo or SF Mono shows them as tofu boxes or blank glyphs. Install a Nerd Font that includes the Powerline set — recommended: MesloLGS NF (Meslo LG S) — then select **MesloLGS NF** as the profile font in Terminal.app (**Settings → Profiles → Text → Font**). Open a new window afterwards.

On the Linux console (`TERM=linux`), portage-ng falls back to angle-bracket labels (`<install>`) via `config:powerline_bubbles/0`; no special font is required there.

14.1.4 Pre-action steps

Before the merge list, the printer can show **pre-action steps** — configuration changes that the plan assumes have been applied. These correspond to the `suggestion` tags from the prover (see Chapter 9):

- **Accept keyword** — packages that need `~arch` keyword acceptance.
- **Unmask** — packages that need unmasking.
- **USE changes** — flag changes needed in `package.use`.

Each pre-action step shows the exact line you would add to the corresponding `/etc/portage/package.*` file.

14.1.5 Summary line

At the bottom, a summary line shows the total number of actions (new installs, upgrades, reinstalls, etc.), the step count, and the predicted download sizes (to be downloaded vs. already downloaded).

14.2 Assumption and warning output

After the merge list, the printer shows any assumptions the prover had to make. These are grouped into two categories:

Domain assumptions are situations where the prover could not find a real solution and had to accept a literal on faith. Each assumption is printed with:

- The package or dependency that could not be satisfied.
- A classification label (e.g. "non-existent dependency", "REQUIRED_USE violation", "model unavailable").
- An actionable suggestion showing how to resolve the issue.

Cycle breaks are points where the prover broke a dependency cycle. Each cycle break shows the cycle path (which packages form the loop) and an explanation of why the cycle was broken rather than treated as benign.

The assumption type classification is handled by `Printer/Plan/assumption.pl`, and the detailed rendering by `Printer/Plan/warning.pl`. For a full description of the assumption taxonomy, see Chapter 9.

14.3 Writing module

The writer (`Source/Application/Output/writer.pl`) generates `.merge` files — one per target package — containing the portage-ng plan output in a format comparable to `emerge -vp` output. These files are stored in the graph directory configured by `config:graph_directory/1` in `Source/Config/<host>.pl`.

`.merge` files serve two purposes:

- **Regression testing** — by comparing `.merge` files against `.emerge` files (the corresponding `emerge -vp` output for the same target), the compare tooling can detect regressions in dependency resolution accuracy. See Chapter 26 for the comparison workflow.
- **Offline review** — the files provide a persistent record of what portage-ng would do for each target, without needing to rerun the resolver.

14.4 Dependency graph generation

The grapher (`Source/Application/Output/grapher.pl`) produces interactive SVG dependency graphs that let you visually explore the dependency tree for a target.

The generation process has three stages:

1. **Edge extraction** — the proof is traversed to collect all dependency edges (which package depends on which, and through what action).
2. **DOT generation** — a `.dot` file is written with nodes representing packages and edges representing dependencies. Nodes are colour-coded by action type and annotated with version and slot information.
3. **SVG rendering** — the `dot` command (Graphviz) renders the DOT file into an SVG. For large graphs, platform-specific scripts under `Source/Application/System/Scripts/` can run multiple renderings in parallel.

Graph generation is triggered by the `--graph` CLI flag. The resulting SVGs include interactive navigation (zoom, pan, node highlighting) via a built-in JavaScript theme (`navtheme` module).

The screenshot below shows a dependency tree for `app-editors/vim`. The root package appears at the top with its dependencies fanning out below. Nodes are colour-coded: the red-bordered root, grey nodes for already-installed packages, and white nodes for packages that need to be merged. The toolbar at the top allows filtering by dependency type (BDEPEND, DEPEND, RDEPEND, etc.) and switching between graph views.

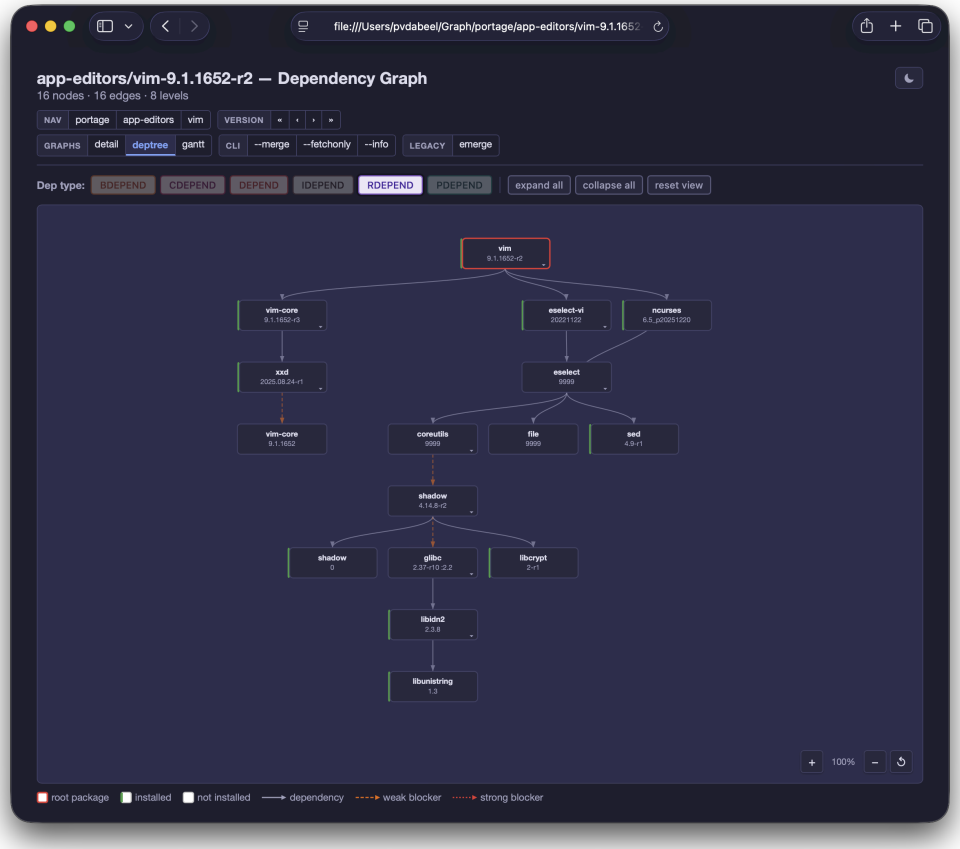


Figure 43 — Dependency graph for app-editors/vim

The **detail view** shows a single package with its full metadata — USE flags (with conditionals), candidate versions, installed status, and dependency atoms. This view is useful for understanding exactly which candidates are available and how USE conditionals affect the dependency tree.

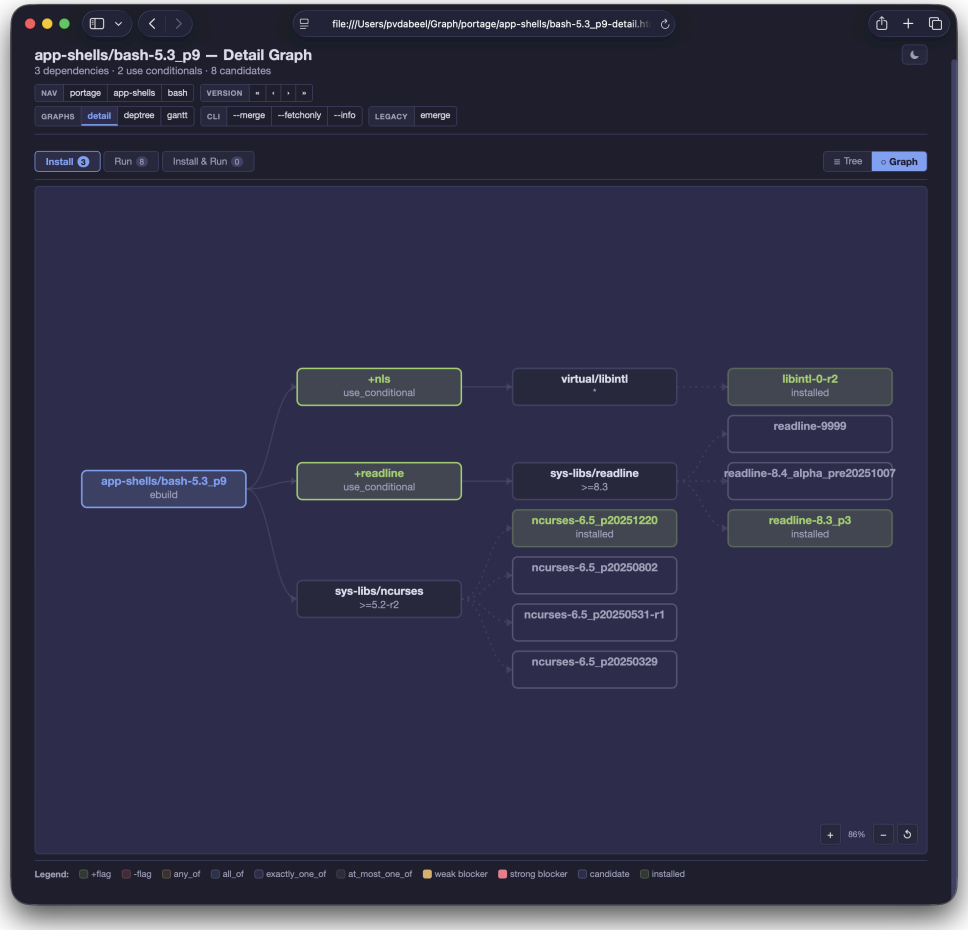


Figure 44 — Detail view for app-shells/bash

14.4.1 Graph submodules

Module	Purpose
`dot`	Graphviz DOT file generation with colour and layout
`deptree`	Hierarchical dependency tree visualisation
`detail`	Detailed single-package view with all metadata
`gantt`	Gantt chart rendering (see below)
`terminal`	Terminal-based ASCII graph rendering
`security`	GLSA page: advisories referencing the package, fold-open full text (see Chapter 20)
`tracker`	Bugs page: Bugzilla bugs naming the package from the synced bug store, open bugs and exact-version matches highlighted; facet chips (status, resolution, version, component, severity, keyword, age) and free-text filtering, state kept in the URL hash (see Chapter 19)
`navtheme`	JavaScript navigation theme for interactive SVGs

14.5 Gantt charts

The `gantt` module produces Gantt charts that visualise the parallel build schedule computed by the ordering pass. The default view is wave-accurate: each column is one planning step, and packages in the same wave sit side by side. **Critical path** highlights the longest *duration-weighted* chain of dependent actions (forward edges only). Durations come from `Knowledge/phase_stats.pl` (`phase_seconds/3`, exact CPV or the average of other versions of the same C/N). Downloads use bytes at `config:download_mbit/1` (default 100 Mbit/s): a live git3-src cache, else this CPV's Manifest, else the average Manifest size of other same-C/N versions (so a `9999` ebuild is sized from release tarballs when the git cache is cold). Actions with no measurement keep a unit weight so the path degrades to hop count. **Time** keeps that wave order and sizes each wave by its longest recorded action — a wave starts only when the previous wave finishes — so a 20-minute rust compile is a wide band and later steps sit to its right.

The screenshot below shows the execution plan for `app-editors/neovim`. Each row is a package; colour-coded blocks show download (blue), install (green), and run (light green) phases across ten steps. Dependency edges are drawn as coloured curves linking the phases — red for DEPEND, blue for BDEPEND, green for RDEPEND — making it easy to see which packages gate others and where parallelism is exploited.

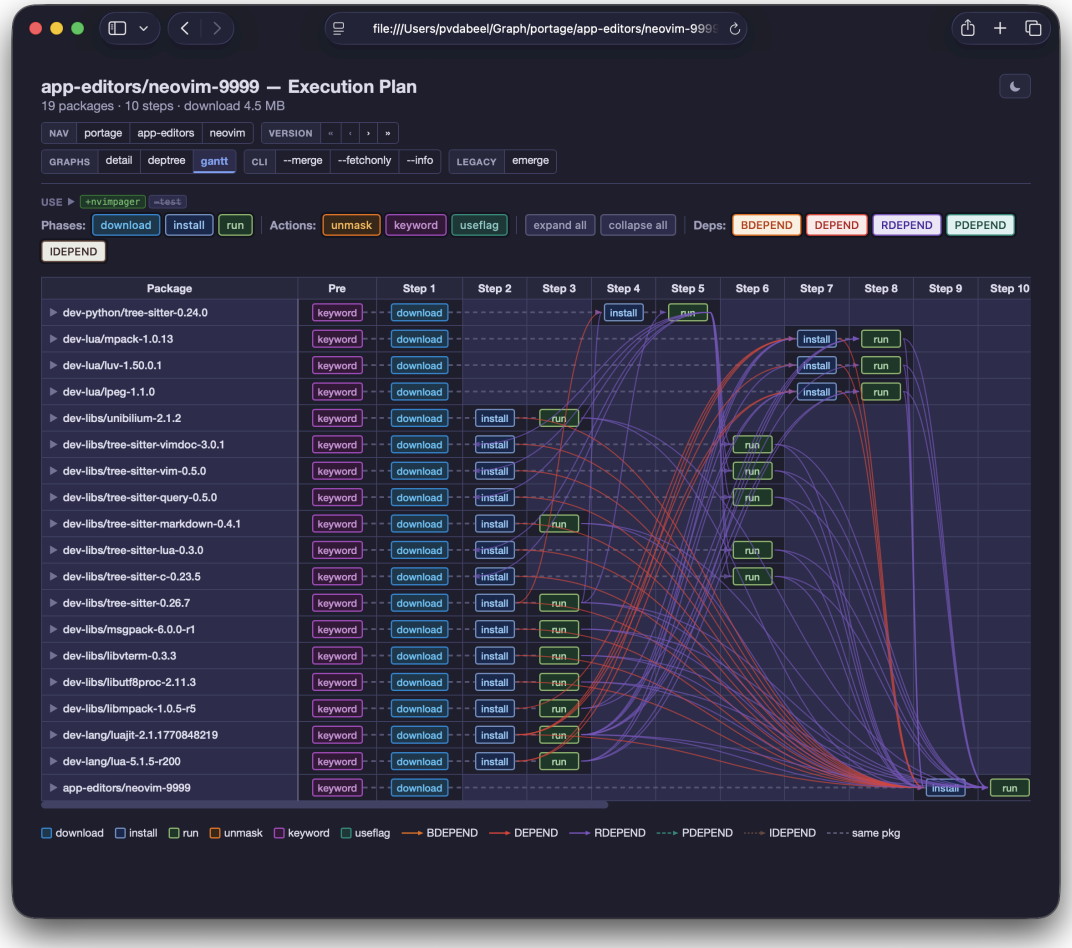


Figure 45 — Gantt chart for app-editors/neovim

14.6 Report generation

The report module (`Source/Application/Output/Report/report.pl`) generates structured reports for analysis, typically as JSON files. Reports can include:

- **Plan summaries** — total actions, waves, parallelism metrics.
- **Assumption breakdowns** — counts and details of domain assumptions, cycle breaks, and blocker conflicts.
- **Performance statistics** — proof time, plan time, cache hit rates, reprove retry counts.
- **Comparison data** — structured output consumed by the tinderbox-ng compare harness (`tinderbox-ng compare` / `tinderbox-ng analyze`), which diffs portage-ng plans against emerge output.

14.7 Printer submodules

The printer pipeline is split across focused submodules:

Module	File	Responsibility
`plan`	`Printer/Plan/plan.pl`	Plan rendering (waves, actions, USE flags)
`assumption`	`Printer/Plan/assumption.pl`	Assumption classification and display
`cycle`	`Printer/Plan/cycle.pl`	Cycle explanation rendering
`warning`	`Printer/Plan/warning.pl`	Assumption detail and warning blocks
`timing`	`Printer/Plan/timing.pl`	Build time display
`index`	`Printer/index.pl`	Package index display
`info`	`Printer/info.pl`	Package info display
`stats`	`Printer/stats.pl`	Statistics display
`state`	`Printer/state.pl`	State tracking during printing
`news`	`Printer/News/news.pl`	Gentoo news item display

14.8 Further reading

- Chapter 13: Ordering — Plans as Proofs — how waves and parallelism are computed
- Chapter 15: Command-Line Interface — `--graph`, `--verbose`, `--quiet`, and other output flags
- Chapter 16: Building and Execution — how the plan is executed
- Chapter 26: Testing and Regression — how `.merge` files are used for regression testing

Chapter 15

Command-Line Interface

portage-ng is meant to sit beside Portage, not replace it in name or habit. Many flags will feel immediately familiar: `--pretend`, `--verbose`, `--emptytree`, and the usual resolution switches mirror what you already use with emerge-style workflows. On top of that, a proof-based resolver can expose tools that a traditional dependency solver does not: `--explain` and `--llm` for plan dialogue, `--diagnose` / `--log` for metacircular build-failure repair, `--variants` for USE-sensitive alternatives, and `--search` that can treat a phrase as a natural-language query when structured parsing does not apply.

The CLI is organized around one idea: **every invocation either reasons about packages or acts on them.** Reasoning covers dry-runs, search, similarity, estimates, upstream checks, Bugzilla lookup, and anything that inspects the knowledge base without changing the system. Acting covers merge, unmerge, depclean, fetch-only, and sync-style maintenance. Keeping that distinction in mind makes it easier to choose flags and to script portage-ng safely (often pairing `--pretend` with exploratory options before any real merge).

15.1 Modes

portage-ng operates in one of six modes, selected with `--mode`:

Mode	Description
`standalone`	Full local operation — the default and most common mode
`daemon`	Persistent daemon serving IPC clients via Unix socket
`ipc`	Thin IPC client forwarding requests to a running daemon
`client`	Remote RPC client connecting to a server over HTTPS
`worker`	Compute node for distributed proving (polls server for jobs)
`server`	HTTP + Pengines server with job/result queues

15.1.1 Standalone

The default mode. Loads the full pipeline, knowledge base, LLM modules, and domain logic into a single process. All resolution, planning, graph generation, and building happens locally. Every CLI action (`--pretend`, `--sync`, `--graph`, `--shell`, etc.) is available.

15.1.2 Daemon

Keeps the same in-memory footprint as standalone — full knowledge base, resolver, orderer — but listens on a **Unix domain socket** for incoming requests. Use `--background` to fork the daemon into a detached process. The daemon avoids the startup cost of reloading the knowledge base on every invocation, making repeated queries fast.

15.1.3 IPC

A thin front-end that does **not** load the full resolver stack. It connects to a running daemon over the Unix socket, forwards the command-line arguments and environment, streams output back, and exits with the daemon's exit code. If `--background` auto-start is configured and no daemon is listening, the IPC client can launch one automatically. Note that `--shell` is not supported in IPC mode.

15.1.4 Client

A lightweight process that treats a remote **server** as the source of truth for the knowledge base. Local queries are proxied over HTTPS using Pengine RPC (with TLS certificates and digest authentication). The client loads enough of the pipeline to drive the CLI, but proving and KB access happen on the server side. Use `--host` and `--port` to specify the server.

15.1.5 Server

Runs the full standalone pipeline first (local KB, resolver, orderer), then adds an HTTPS Pengine server, TLS, and Bonjour service advertisement. The server exposes job and result message queues so that workers can poll for proving tasks. Use `--background` to fork the server process. See Chapter 18: Distributed Proving.

15.1.6 Worker

A compute node that loads the full proving pipeline locally (like standalone) plus an RPC client for server communication. On startup, the worker discovers the server via Bonjour or explicit `--host`/`--port`, syncs its local portage tree to the server's snapshot, registers its CPU count, and spawns one thread per core. Each thread polls the server for jobs, proves them locally, and posts results back. See Chapter 18: Distributed Proving.

15.2 Actions

Actions are grouped by area. Use the tables below as a quick map from flags to behaviour; the sections that follow add context on targets, search, and everyday workflows.

15.2.1 Merge and resolution

Flag	Action
`--pretend`	Generate and display a build plan (dry-run)
`--merge`	Execute the build plan
`--unmerge <target>`	Remove a package
`--depclean`	Remove unneeded packages
`--fetchonly` / `-f`	Same `:run` proof as `--merge`; print and execute downloads (and unmask / keyword / USE / license pre-actions) only
`--fetch-all-uri` / `-F`	Same as `--fetchonly`, plus every `SRC_URI` regardless of USE

`--fetchonly` is a **restriction on the merge plan**, not a second resolve. The prover still proves `target:run` (same 5-tier fallback as `--merge`). The `fetchonly` preference flag then:

- hides install / run / update / downgrade / reinstall from the printer
- lets the builder execute only the remaining download (and pre-action) steps
- skips `@world` registration, even on a real (non-pretend) run

`-F` uses that same filter and additionally asks `ebuild:distfile_scope/1` for every file listed in `SRC_URI`. Combine with `--build` to actually fetch; without `--build` the filtered plan is printed only (same split as `--merge` vs `--build`).

15.2.2 Information

Flag	Action
`--search <query>`	Search packages (supports natural-language via embeddings)
`--similar <target>`	Find packages similar to target (vector similarity)
`--info`	System overview (version, hostname, repositories, world set) without arguments; per-package details with a target
`--pretend @installed`	List installed packages (via the computed `@installed` set)

15.2.3 Repository management

Flag	Action
`--sync`	Sync the Portage tree and regenerate caches
`--regen`	Regenerate md5-cache incrementally
`--import-vdb`	Client mode: ship the local VDB to the server so remote plans reflect the client's installed packages (see Chapter 18)

15.2.4 Visualization

Flag	Action
`--graph`	Generate interactive SVG dependency graphs
`--estimate`	Show build time estimates

15.2.5 Diagnostics

Flag	Action
`--bugs <target>`	Prove the target (resolve-only) and print Gentoo Bugzilla bug-report drafts for its domain assumptions
`--search-bugs <term>`	Search for known issues: local bug store first, then Gentoo Bugzilla (policy `config:bugzilla_search/1`, see Chapter 19)
`--upstream <target>`	Check upstream versions via Repology
`--explain` / `--llm`	Get AI-assisted plan explanation
`--diagnose` / `--log`	Metacircular LLM diagnose of a failed build
`--variants`	Show plan variants with different USE configurations
`--shell`	Drop into an interactive Prolog shell

15.3 Options

15.3.1 Resolution options

Flag	Effect
`--emptytree`	Prove all dependencies from scratch (ignore VDB)
`--onlydeps`	Prove only dependencies, not the target itself
`--deep`	Deep dependency resolution
`--newuse`	Detect USE flag changes requiring rebuilds
`--update`	Update to newest version
`--optimize parallelism`	Plan ordering strategy (default): a soft requirement (preference, e.g. RDEPEND-before-consumer) whose two ends lie on a common loop is void, so the loop's members merge side by side; preferences between loops are all kept. Shallower plans, order-independent
`--optimize soft-requirements`	Hand every preference to the planner, which honors as many as it can and drops only those that would close a loop; loop members merge one per step, resembling `emerge`'s linear listing

Both `--optimize` strategies satisfy every hard requirement; they differ only in what happens to soft requirements (preferences) that contradict each other. See Chapter 13, "Preferences: soft requirements", for a worked example with diagrams.

15.3.2 Output options

Flag	Effect
`--verbose`	Verbose output (show USE flags, slot info)
`--quiet`	Minimal output
`--ci`	Non-interactive CI mode (exit codes 0/1/2)
`--jobs N`	Number of parallel jobs
`--timeout N`	Abort proving/planning after N seconds (0 = no limit)

15.4 Target syntax

Targets can be specified in several formats:

Format	Example	Meaning
`cat/pkg`	`sys-apps/portage`	Resolve latest version
`=cat/pkg-ver`	`=sys-apps/portage-3.0.77`	Exact version
`>=cat/pkg-ver`	`>=dev-lang/python-3.10`	Version constraint
`@set`	`@world`, `@security`, `@changed-deps`	Package set (file-backed, profile, or computed)
`pkg`	`portage`	Ambiguous name (searched across categories)

15.5 Package sets

`@name` targets expand to concrete atoms via `eapi:substitute_sets/2` before proving. File-backed sets (`@world`, `@system`, user sets under `config:set_dir/1`) come from preference configuration. A set file (or the world file) may contain another `@name`; those nested references are expanded in place. A cycle, or an `@name` that is not a configured set, is a hard error when reached from inside another set. **Computed sets** are registered in `Source/Domain/Gentoo/Preference/sets.pl` and resolved on demand by `sets:expand/2`.

```
portage-ng --mode standalone --list-sets
portage-ng --mode standalone --ci --pretend @security
portage-ng --mode standalone --ci --pretend @preserved-rebuild
portage-ng --mode standalone --ci --pretend @changed-deps
```

An empty computed set prints an informational line and exits 0 under `--ci` (nothing to do), not a hard failure.

Set	Atoms	Meaning
`@world` / `@system`	as configured	Preference / profile sets
`@installed`	`cat/name:slot`	Everything installed
`@live-rebuild`	`cat/name:slot`	Installed `PROPERTIES=live` packages
`@changed-subslot`	`cat/name:slot`	Subslot differs from highest visible ebuild
`@downgrade`	`cat/name:slot`	Highest visible ebuild is older than installed
`@unavailable`	`cat/name:slot`	No visible ebuild in the same slot
`@rebuilt-binaries`	`=cpv`	Binpkg BUILD_TIME ≠ installed BUILD_TIME
`@unavailable-binaries`	`cat/name:slot`	No binpkg for the installed version
`@security`	`=cpv`	GLSA NewAffectedSet (default security set)
`@affected` / `@new-affected` / `@new-glsa`	`=cpv`	Other Portage security-set filters
`@preserved-rebuild`	`cat/name:slot`	Consumers of FEATURES=preserve-libs leftovers
`@changed-deps`	`=cpv`	VDB RDEPEND/PDEPEND drifted from same-version ebuild

`@preserved-rebuild` reads Portage's `preserved_libs_registry` JSON (default: derive from `config:pkg_directory/1` as `…/lib/portage/preserved_libs_registry`; override with `config:preserved_libs_registry_override/1`) and matches consumers via VDB `NEEDED.ELF.2`. It is complementary to the automatic `config:subslot_rebuild/1` pass, which rebuilds `:=` reverse deps when a provider's subslot changes inside a plan.

`@changed-deps` compares installed RDEPEND/PDEPEND (from the on-disk VDB) to the same-version tree ebuild after use-reduce and `:=` stripping, with libc injects removed (emerge `--changed-deps` semantics). The `--changed-deps` flag applies the same test while resolving other targets.

GLSA details for `@security` and siblings: Chapter 20. Full option text: `portage-ng(1)`.

15.6 CI mode

Use `--ci` for non-interactive automation. Exit codes indicate plan quality:

Code	Meaning
0	Plan completed with no assumptions
1	Plan completed with prover cycle-break assumptions only
2	Plan completed with domain assumptions (e.g. missing deps)

Example:

```
portage-ng --ci --pretend sys-apps/portage
echo $?  # 0, 1, or 2
```

By default, portage-ng runs in standalone mode. Other modes (distributed client, server, worker) are covered in the advanced topics chapters.

15.7 The dev wrapper

When running from a source checkout, use the dev wrapper instead of the installed binary:

```
./Source/Application/Wrapper/portage-ng-dev --pretend sys-apps/portage
```

The wrapper sets up the correct load paths, stack limits, and Prolog flags. It also supports `--timeout N` (requires Python 3) to kill the process after N seconds. For reproducible, non-interactive runs, pipe queries via a here-doc:

```
./Source/Application/Wrapper/portage-ng-dev --shell --timeout 60 <<'PL'
resolver:test_stats(portage).
halt.
PL
```

15.8 Tips and tricks

Short recipes that match how people actually use the tool:

- **What does portage-ng think about this package?**
  `portage-ng --pretend --verbose cat/pkg` — full plan with enough detail to compare against emerge-style output.
- **Why is this package in my plan?**
  `portage-ng --pretend --explain cat/pkg` — ask the explainer/LLM path to narrate the plan (see Chapter 17: Semantic Search and LLM Integration).
- **Diagnose a failed build with metacircular LLM repair**
  `portage-ng --diagnose cat/pkg` (optional `--log path`) — propose `feedback:*` learning from the build log; confirm before apply (same chapter).
- **What would change if I enabled this USE flag?**
  `portage-ng --pretend --variants cat/pkg` — surface alternative proofs when USE sets differ.

- **Find packages related to *X***
  `portage-ng --search "X"` — natural-language / semantic search when the query is not structured (requires embeddings; same chapter as above). For an exact package name, use a structured atom such as `name=vim` (the same intent as "`name:X`" in prose, but the CLI grammar uses `=` for equality, not a single `name:X` token). Category and other fields work the same way (`category=…`); see Search query language below.
- **Show me similar packages**
  `portage-ng --similar cat/pkg` — vector similarity from the same embedding stack as semantic search.
- **Quick scripted session**
  Here-doc into the Prolog shell so the full load graph matches interactive use:

```
portage-ng --mode standalone --shell <<'PL'
resolver:test_stats(portage).
halt.
PL
```

- **CI / automation**
  `portage-ng --ci --pretend cat/pkg` — non-interactive; interpret exit codes: `0` no assumptions, `1` cycle-break assumptions only, `2` domain assumptions present.
- **Estimate build time**
  `portage-ng --estimate cat/pkg` — build-time hints from VDB and history.
- **Check for upstream updates**
  `portage-ng --upstream cat/pkg` — Repology-oriented upstream comparison.
- **Draft bug reports**
  `portage-ng --bugs cat/pkg` — prove the target (resolve-only) and print Bugzilla-style bug-report drafts for its domain assumptions.
- **Search Bugzilla**
  `portage-ng --search-bugs term` — look up known issues. With a synced `bugzilla` repository the local bug store answers first (a `cat/name` term uses the package atom index); the live REST quicksearch is the fallback, or the only source with `config:bugzilla_search(rest)`.
- **Keep the local bug store fresh**
  `portage-ng --sync bugzilla` — fetch changed bugs into `Knowledge/bugs.qlf`; `config:repository_sync_limit(bugzilla, 1)` caps the network step at one run per day.

15.9 Search query language

The `--search` flag accepts **structured** queries built from one or more command-line atoms. Each atom is a *key*, a *comparator*, and a *value* (see Fuzzy and wildcard search for the comparators). When the argument list does **not** parse as that structured form, the text is joined and passed to **semantic** (natural-language) search instead.

```
portage-ng --search name=vim category=app-editors
portage-ng --search license=GPL-2 keywords=amd64
portage-ng --search "text editor with syntax highlighting"  # semantic search
```

Semantic search requires Ollama with a loaded embedding model. See Chapter 17: Semantic Search and LLM Integration.

15.9.1 Fuzzy and wildcard search

Structured search uses explicit comparators on the key:

Comparator	Meaning	Example
`=`	Exact match on the value	`name=vim`
`~`	Fuzzy match (approximate / substring-style, key-dependent)	`name~vim`
`:=`	Wildcard match (`*` in the value)	`name:=*vim*`

Exact search — constrain the package name or another field precisely, e.g. `--search name=vim` (exact package name). In documentation you may see this described informally as `name:vim`; on the command line the equality comparator is `=` (`:` introduces the `:=` wildcard operator instead).

Category filter — `category=app-editors` (or combine with other atoms on the same command line).

Natural language — a query that does not parse as structured keys, e.g. `--search "text editor with syntax highlighting"`, uses vector embeddings over the knowledge base (when enabled and indexed).

Wildcard — use `:=` so `*` is interpreted as a glob-style wildcard, e.g. `name:=*vim*` for any package name containing `vim`. Quote the atom if the shell would expand `*` (e.g. `--search 'name:=*vim*'`).

Combined filters — pass several atoms; each narrows the result set, e.g. `category=dev-libs name:=*ssl*`.

15.10 Further reading

- `portage-ng(1)` manpage — exhaustive option reference
- Chapter 2: Installation and Quick Start — first run examples
- Chapter 14: Output and Visualization — what the output looks like
- Chapter 20: Gentoo Linux Security Advisories (GLSA) — `@security` and related GLSA computed sets
- Chapter 3: Configuration — `config:preserved_libs_registry/1` and related paths

Chapter 16

Building and Execution

portage-ng is a self-contained dependency resolver and planner. It ships its own code for every stage up to the point where source code must actually be compiled:

- **Cache generation** — portage-ng includes its own md5-cache generator, so it does not depend on Portage's `egencache` or any other external tool to produce the cache files it reasons over.
- **Dependency resolution and ordering** — the prover and the ordering engine are entirely internal (see Chapters 8-12).
- **Downloading** — source archive fetching, mirror selection, hash verification, and resume are handled by portage-ng's own download module (see Download management below).

The only point where Portage is needed is the **execution of ebuild build phases** (unpack, compile, install, merge, etc.). That hand-off is deliberate. Everything before and after those phases — dependency calculation, plan ordering, downloading, display, and recovery — is owned by portage-ng; the phase bodies themselves remain Portage's.

16.1 Why not our own `ebuild`?

It is natural to ask why a project that already replaced Portage's resolver, planner, cache generation, and download path still shells out to Portage's `ebuild` for the last mile. The short answer is that **portage-ng is a reasoning engine that happens to build packages**, not a reimplementation of Gentoo's build runtime — and those are different problems with different costs.

The hard problem this project set out to solve is *configuration as proof*: which versions, which USE flags, which order, which assumptions, and why. That is where traditional emerge is weakest, and where a Prolog prover earns its keep. Phase execution is a different kind of difficulty. Making `src_compile` and `pkg_postinst` work for thirty thousand ebuilds is less about elegant search and more about decades of accumulated bash helpers, eclasses, sandbox policy, FEATURES knobs, Prefix layout, binary-package merge, and VDB writes. Portage already ships that stack — on the order of many thousands of lines of `ebuild.sh` helpers plus the Python merge path — and every real package in the tree depends on it behaving exactly as maintainers expect.

Replacing it would not make plans better. It would make portage-ng responsible for a moving compatibility surface: every EAPI edge case, every eclass quirk, every sandbox interaction that today "just works" because the same `ebuild` that emerge uses is the one that runs. An in-house layer that is *almost* Portage becomes a permanent tax — a second implementation forever chasing the first — without improving the resolver metrics that actually distinguish this project.

So the design chooses a clean cut instead of a rewrite. portage-ng owns *which* phases run, *across which packages*, with *which USE*, under *which concurrency rules*, and how failures are interpreted or repaired. Portage's `ebuild` owns *what happens inside a phase*. That cut is not an informal shell-out; it is the ebuild contract below — a fixed invocation shape, environment, vocabulary, and exit-code semantics. The builder and the execution modules speak only that interface. `config:ebuild_command/1` already names the backend, so a future in-house layer can satisfy the same contract without touching the planner, jobserver, display, or fixup registry.

In other words: **defer the runtime, own the reasoning.** Shipping our own `ebuild` remains a possible later step — for Prefix hygiene, for dropping the sys-apps/portage dependency, or for tighter isolation — but it is optional precisely because the contract keeps it optional. Until then, the Gentoo tree builds with the ecosystem it was written for, and portage-ng spends its depth where depth changes the outcome of a prove.

16.2 The ebuild contract

16.2.1 Division of responsibility

Concern	Owner	Where
md5-cache, VDB facts, preferences	portage-ng	reader / knowledge base
Dependency proof and ordered plan	portage-ng	resolver / orderer
Distfile fetch + Manifest verify	portage-ng	`Builder/download.pl`
Wave scheduling and `--jobs` tokens	portage-ng	`Builder/jobserver.pl`
Live display, logs, phase stats	portage-ng	`Builder/display.pl`, `ebuild_exec`
Per-package `USE=` string	portage-ng	`ebuild_exec:collect_use_string/4`
Phase sequence for a plan action	portage-ng	`ebuild_exec:action_phases/3`
Eclass stack, sandbox, `econf`/`emake`/`doins`	Portage `ebuild`	`ebuild.sh` + helpers
Image → live FS + VDB write	Portage `ebuild`	`merge` / `qmerge`
`FEATURES`, `ROOT`, `EPREFIX`, `PORTAGE_TMPDIR`, make.conf	Portage config stack	read by `ebuild`, not rewritten by portage-ng
Post-merge toolchain / `ghc-pkg` reactivation	portage-ng	hooks after a successful merge (see below)
Domain exception fixups and replans	portage-ng	`Exceptions/`, `builder:build_loop/2`

portage-ng never reimplements ebuild phase bodies. It chooses *which* phases to run, *in what order across packages*, with *which USE*, and interprets the exit code. Portage's `ebuild` remains responsible for *what happens inside a phase*.

16.2.2 Invocation shape

Every source-build phase is started as:

```
<ebuild_command> --skip-manifest <ebuild-path> <phase>
```

- **`<ebuild_command>`** — `config:ebuild_command/1` (default `ebuild`; override in the per-host config for Gentoo Prefix or a custom path).
- **`--skip-manifest`** — always passed. Distfile integrity is already enforced by portage-ng's download stage; re-checking Manifest inside each phase would duplicate work and fight resume / mirror layouts.
- **`<ebuild-path>`** — absolute path of the `.ebuild` file for `Repo://Entry`, resolved via `Repo:get_ebuild_file/2`.
- **`<phase>`** — one atom from the allowlist in `sanitize:safe_phase/1` (`clean`, `setup`, `unpack`, `prepare`, `configure`, `compile`, `test`, `install`, `package`, `merge`, `unmerge`, …). Unknown phase names are rejected before any process is spawned.

Logged / async runs wrap that argv in a **fixed** `sh -c` script whose data travels only as positional parameters (`$1`…`$4`), never by string interpolation — the same argv contract documented in `Source/Application/Security/sanitize.pl`. Bulk (no-progress) runs call `process_create` on the ebuild binary directly with the phase list as argv elements.

16.2.3 Action → phase sequence

Plan actions map to a phase list in `ebuild_exec:action_phases/3`. Build-shaped actions share one sequence from `ebuild_exec:build_phases/1`:

Plan action	Phases
`install` / `reinstall` / `update` / `downgrade`	see build sequence below
`uninstall`	`[unmerge]`
`run`	`[]` (no ebuild invocation)

Default build sequence (source path):

```
[clean?, setup, unpack, prepare, configure, compile, test?, install, package?,
merge?]
```

Optional segments:

Segment	When included
`clean`	omitted under `--resume` (`ebuild_exec:resuming`) so workdirs are reused
`test`	only when `FEATURES` contains positive `test` (`config:features_test_enabled`)
`package`	when `--buildpkg` or `--buildpkgonly` is set
`merge`	omitted under `--buildpkgonly` (binary package only)

`merge` is the composite Portage phase: `pkg_preinst`, copy into the live filesystem, unmerge of the old version on update/downgrade/ reinstall, VDB write, and `pkg_postinst`. portage-ng therefore uses the **same** phase list for install and for replacement actions — it does not unmerge the old version before building (that would break compile-time use of the installed predecessor).

Which of those phases actually run is further gated by `config:build_live_phases/1`: a leading *live prefix* executes; the trailing *stub tail* is displayed but not run. An empty list stubs the whole build (display-only). The live prefix cannot skip a middle phase and continue — `compute_live_prefix/4` stops at the first phase absent from the config.

16.2.4 Environment contract

portage-ng injects a small, explicit environment; everything else is whatever Portage's `ebuild` would read on that host.

Variable	Set by	Meaning
`USE`	portage-ng	Space-separated tokens (`flag` or `-flag`) matching the resolver's effective USE for this entry, including proof-context overrides (`build_with_use`, REQUIRED_USE assumptions, `suggestion(use_ch`
`MAKEOPTS`	portage-ng (retry only)	Forced to `-j1` on the serial-make retry path; otherwise left to Portage / make.conf.
`FEATURES`, `ROOT`, `EPREFIX`, `PORTAGE_TMPDIR`, `PKGDIR`, …	Portage config stack	Not rewritten by the builder. Prefix, temporary dirs, and feature flags come from the same `make.conf` / profile / `/etc/portage` that a bare `ebuild` would see.

Agreement between planner and builder on `USE` is part of the contract: `collect_use_string/4` resolves each IUSE flag through `use:effective_use_for_entry/3` — the same predicate the

prover uses — then overlays proof-context forcing. A mismatch here is a contract bug (the plan says one thing; the ebuild runs another).

16.2.5 Outcomes

Each action ends as one of:

Outcome	Meaning
`done`	every live phase exited 0
`failed(ExitCode)`	a phase exited non-zero (or setup could not locate the ebuild: `failed(no_ebuild)`)
`failed(qmerge_exit(N))`	binary path: `ebuild … qmerge` failed

Per-phase progress callbacks (live display) use the states `active`, `progress(Pct)`, `done`, `failed(ExitCode, LogPath)`, `stub` (beyond live config), and `skipped` (after a failure). Exit status is the sole success signal; log size is used only for progress estimates, never to infer whether a phase completed.

16.2.6 Concurrency

Within a wave, independent packages may run phases in parallel under the jobserver. Two rules preserve Portage invariants:

1. **Compile-side parallelism is allowed.** `clean` through `install` (and `package`) may overlap across workers.
2. **`merge`/`qmerge` are exclusive.** Both take the `portage_pkg_merge` mutex. Merge is not a pure file copy: `pkg_preinst` collision-protect reads the live tree, and `pkg_postinst` runs process-global updaters (`ldconfig`, `env-update`, preserved-libs, mime/icon/schema caches). Traditional `emerge --jobs N` parallelizes compilation and serializes the merge into `${ROOT}`; portage-ng restores that invariant for the bare `ebuild` CLI.

16.2.7 Post-merge hooks (outside the ebuild binary)

The bare `ebuild` CLI does not perform everything `emerge` does between packages. After a successful install-family action, portage-ng runs domain hooks that keep the live root coherent for later phases in the same plan:

- **Toolchain reactivation** (`maybe_reactivate_toolchain/4`, gated by `config:toolchain_reactivation/1`) — after merging `sys-devel/gcc` (and related toolchain CNs), re-select the gcc-config profile and/or run `env-update` so a later package's `pkg_setup` sees the new compiler (portage-ng#86, especially under `FEATURES=ccache`).
- **`ghc-pkg recache`** (`maybe_register_ghc_pkg/4`, gated by `config:ghc_pkg_register/1`) — after merging `dev-lang/ghc`, refresh the package DB so boot libraries are visible to subsequent `dev-haskell/*` configures (portage-ng#108).

These hooks are part of the *builder-side* half of the contract: a replacement ebuild backend that already performed equivalent work could no-op them, but today's Portage `ebuild` path relies on them.

16.2.8 Binary packages (same contract, different entry)

When `config:use_binpkg/1` is enabled and `binpkg_exec:available_for/4` finds a USE-/slot-/keyword-compatible gpkg, the builder short-circuits the source phase list and drives:

```
ebuild --skip-manifest <source-ebuild-path> qmerge
```

with environment including the resolver's `USE`, plus `MERGE_TYPE=binary`, `PORTAGE_BINPKG_FILE=<gpkg>`, and `PORTAGE_BUILDDIR=<builddir>`. Extraction is owned by `binpkg_extract`; VDB write and `pkg_*inst` remain inside Portage's `qmerge`. Failure outcomes and the merge mutex are shared with the source path. Producing binpkgs (`package` phase under `--buildpkg`) stays on the source path in `ebuild_exec`.

`binpkg_extract` lists each tar (the outer gpkg, `image.tar.zst`, and `metadata.tar.zst`) and refuses it before extraction when a member is an absolute path, contains `..`, is a device/fifo node, is a hardlink that escapes, or would write through a symlink whose target leaves the destination directory (Gentoo bug 982208). Metadata members must be regular files or directories.

`binpkg_exec:entry_allowed/2` also skips a binpkg when `/etc/portage/patches` would apply to the corresponding ebuild (`${PN}`, `${P}`, `${PF}` or `${P}-${PR}`, each optionally `:${SLOT}`), matching emerge's `--binpkg-respect-user-patches` (default **y**). `--usepkg-include` does not override that skip. Pass `--binpkg-respect-user-patches n` (or set `config:binpkg_respect_user_patches(false)`) to accept the binpkg anyway.

16.2.9 What a replacement backend must provide

To swap Portage's `ebuild` for an in-house layer without touching the planner, a backend must:

1. Accept `--skip-manifest <ebuild-path> <phase>` (or be wrapped so the builder's argv stays unchanged via `config:ebuild_command/1`).
2. Honour the phase vocabulary and ordering above, including composite `merge` / `qmerge` semantics for live FS + VDB.
3. Treat process exit status as the sole success signal.
4. Read host Portage configuration for `FEATURES` / `ROOT` / `EPREFIX` / temporary directories until those knobs are lifted into portage-ng.
5. Tolerate `USE` (and optional `MAKEOPTS`) injected by the parent environment.
6. Remain safe under parallel compile-side workers with exclusive merge/qmerge.

Until that backend exists, `config:ebuild_command/1` points at sys-apps/portage's `ebuild` (or a Prefix wrapper of it).

16.3 Build orchestration

When executing a plan (via `--merge` rather than `--pretend`), the builder walks the plan wave by wave, respecting the parallelism from the ordering pass. Within each wave, independent actions run concurrently under the jobserver. Each build-shaped action is handed to `ebuild_exec:execute/5` or `execute_with_progress/6`, which speak only the contract above.

16.4 Build resilience: the per-phase retry chain

A failed phase is not necessarily a failed build. After every phase, `ebuild_exec` runs a chain of retry hooks, each keyed on a *signature* found in the log segment written by the failed phase (never earlier phases), so deterministic build failures never match and keep their original semantics. The chain has two layers:

1. **Environmental retries** (in `ebuild_exec.pl` itself) — failures caused by the build environment, not by the package:

Retry	Signature	Recovery	Gate
Transient (bash PID reuse)	`wait: pid N is not a child of this shell`	re-run the phase once	`config:build_transient_retry/1`
Serial make (parallel-make race)	failed compile/test/install phase	re-run with `MAKEOPTS=-j1`	`config:build_serial_retry/1`

2. **Domain exception fixups** (see next section) — failures caused by a problem that should really be fixed at the ebuild or metadata level. The chain ends in a single generic dispatch, `fixup:maybe_phase_retry/9`, which offers the failure to every registered exception mechanism.

Every retry appends a marker line to the build log, so a recovered build is never silent about how it recovered.

16.5 Domain exception fixups

Some build failures are *packaging exceptions*: the build is failing because of a gap in the ebuild or its metadata, not because of the environment or the user's configuration. Traditional emerge either refuses such packages up front or fails and defers to a manual repair tool. portage-ng recovers them in-transaction through a small registry of **exception mechanisms** under `Source/Domain/Gentoo/Exceptions/`:

- **`fixup.pl`** — the generic registry and dispatcher. A mechanism registers itself with three multifile hooks:
  - `fixup:mechanism/1` — identity (load order is dispatch and display order);
  - `fixup:phase_retry_hook/10` — the repair-and-retry logic for a failed phase;
  - `fixup:mechanism_note/3` — the note printed above the affected packages in the build summary.

  Applied fixups are recorded via `fixup:record/3` and reported generically by the build printer — adding a new exception mechanism never touches the builder or the printer.

Mechanisms come in two flavours. Most (collision, GHC ABI, OCaml ABI) are **in-place repairs**: they rebuild something mid-flight and re-run the failed phase. The missing-provider mechanism is different — it **diagnoses but never repairs in place**: it records what it learned and lets the pipeline re-derive a fresh plan (see Missing provider feedback below).

16.5.1 File collision deconfliction

Traditional emerge refuses, at the plan stage, to install a package whose files are owned by a different installed provider — it is told so by an explicit blocker atom in metadata (e.g. installed `sys-apps/util-linux[hardlink]` carries `!app-arch/hardlink`). When that blocker atom is *missing*, the conflict only surfaces at merge time as Portage's `pkg_preinst` collision-protect abort. Gated by `config:deconflict_collisions/1` (`off` | `report` | `override`), the mechanism recognises the collision signature and re-runs the merge with `FEATURES="-collision-protect -protect-owned"`, letting the package overwrite the colliding files. The plan printer already announces this behaviour next to the soft-blocker list, and the build summary lists every package that needed it. The same recovery is applied to binary-package `qmerge` merges.

16.5.2 Haskell ABI repair

Gentoo encodes a Haskell package's identity in `ghc-pkg`'s ABI hash (the suffix in e.g. `bifunctors-5.6.3-9AmA3NO9963FDwV9BBcxcZ`), not in the ebuild sub-slot. When a `dev-haskell` library is rebuilt, its installed reverse-dependencies keep referencing the old hash, and the next Haskell consumer aborts in `pkg_setup`/`configure` with haskell-cabal.eclass's check:

```
installed package semigroupoids-5.3.7 is broken due to missing package
bifunctors-5.6.3-9AmA3NO9963FDwV9BBcxcZ
 * Detected broken packages: semigroupoids-5.3.7 semialign-1.3
 * //==-- Please, run 'haskell-updater' to fix broken packages --==//
```

Because the hash lives only in ghc-pkg's registry, there is no sub-slot delta for the resolver to observe, and traditional emerge fails the same configure and defers to a manual `haskell-updater` run. portage-ng does better: gated by `config:ghc_abi_repair/1`, the mechanism parses the broken package list from the failed phase's log, rebuilds each broken package from source at its installed version and with its VDB-recorded USE configuration (never from a binary package — a stale binpkg ABI is exactly what may be broken), and re-runs the failed phase. One additional bounded round covers cascading breakage exposed by the repair itself.

The mechanism is bounded and observable: each package is rebuilt at most once per session (it can never loop), repairs are serialized across parallel build workers, and every repair leaves markers in both the consumer's and the rebuilt package's build logs plus an entry in the build summary.

16.5.3 OCaml ABI repair

OCaml has the same problem: package identity lives in the compiled interface digests (`.cmi` CRCs) checked by the compiler and in findlib's registry, not in the ebuild sub-slot. Unlike Haskell there is no single eclass check enumerating the broken packages — a stale consumer fails with heterogeneous compiler and ocamlfind messages:

```
Error: The files /usr/lib64/ocaml/site-lib/res/res.cmi
       and /usr/lib64/ocaml/stdlib.cmi
```

```
       make inconsistent assumptions over interface Stdlib
Error: Unbound module Camlp5
ocamlfind: Package `camlp5' not found
```

Gated by `config:ocaml_abi_repair/1`, the mechanism extracts the stale compiled-unit paths and findlib package names from the failed phase's log, maps them to their installed owners through the VDB CONTENTS records (the active enumerator this domain lacks an eclass check for), rebuilds those owners from source at their installed version, and re-runs the failed phase — with the same boundedness guarantees as the GHC repair: at most one rebuild per package per session, at most two retry rounds, repairs serialized across workers, and markers in every involved build log plus the build summary. The package being built and `dev-lang/ocaml` itself are never rebuild candidates.

16.5.4 Missing provider feedback

The three mechanisms above all repair reality in place: they rebuild a package mid-transaction and re-run the failed phase. That pattern is wrong for a whole class of failures — a build that dies because a required *provider* is missing (a command, header, library, or pkg-config module that some package would supply but that the ebuild never declared as a dependency). The canonical case is `sec-policy/selinux-base`, whose compile dies with

```
semodule_package: command not found
```

because `selinux-policy-2.eclass` never lists `sys-apps/semodule-utils` in `BDEPEND`. portage-ng built exactly what it was told to; the dependency simply is not in the metadata, so the resolver never saw it. Repairing this in place would decide ordering imperatively in the builder, could not chase the provider's own transitive needs, would break the invariant that the plan equals `prove_plan(Goals, KB)`, and would forget the discovery so the next run fails again.

So `missing_provider.pl` (portage-ng#102), gated by `config:missing_provider_feedback/1`, does the opposite of a repair: **it emits a structured diagnostic, that diagnostic becomes learned knowledge, the pipeline re-derives a fresh provable plan that orders the provider before the target, and the builder resumes that new plan.** Plans are derived, never patched. It threads the failed phase's exit code through unchanged — the phase legitimately still failed.

The diagnosis is split into two pluggable layers, each a multifile registry so new failure shapes and resolution strategies are added as clauses rather than by editing the dispatcher:

- **Detector registry** (`missing_provider:detector/3`) normalises the failed phase's log tail into a `symbol(Kind, Name)`. Ships detectors for missing commands (bash `command not found`, dash `not found`, `env` exec failures), headers (`fatal error: X.h`), libraries (`cannot find -lX`), sonames, pkg-config modules, and python/perl modules.
- **Resolver chain** (`missing_provider:provider_of/4`) maps a symbol to a concrete `Category/Name` package: first the authoritative VDB `CONTENTS` reverse-owner index (the `qfile`/`equery belongs` equivalent, for providers that happen to be installed), then a small curated seed table (for the common case where the provider is *not* installed — that is precisely why the command was missing). A symbol that maps to no concrete in-tree package is written to an unresolved backlog and the target fails cleanly — no guessing.

A concrete discovery is recorded through the `feedback` module (`Source/Knowledge/feedback.pl`) as a durable `discovered_dep/4` fact, persisted to `Knowledge/feedback.pl` (gitignored, consulted at startup like the QLF cache) so a one-time runtime discovery becomes permanent knowledge. The only resolver change is in `query.pl`, which unions `feedback:discovered_dep(Target, Provider, bdepend, _)` into the target's build-dependency model. The mechanism also distinguishes an *undeclared* dependency (the upstream-gap case above — mint a discovery) from a *declared-but-unbuilt* one (a genuine resolver ordering bug — logged loudly, never papered over).

The control loop lives in the builder: `builder:build/1` is a bounded replan loop (`builder:build_loop/2`, capped by `config:missing_provider_max_replan/1`). When a build pass fails *and* recorded a new discovery, the builder re-enters the pipeline; on the re-proof the provider is part of the closure, so the ordering pass orders it — and its own transitive dependencies — before the target. Everything already built satisfies from the VDB via the existing reconciliation fast path, so the retry pass only builds the provider and recompiles the target. Walkthrough for the selinux case:

1. `selinux-base` compile → `semodule_package: command not found`.
2. `missing_provider` maps the command to `sys-apps/semodule-utils`, records a `discovered_dep`, and persists it; the phase still fails.
3. The wave ends with `selinux-base` failed; `build_loop` sees the new discovery and re-enters the pipeline.
4. `rules`/`query` now yield `BDEPEND(selinux-base) ⊇ {sys-apps/semodule-utils}`; the prover proves it, the orderer places `semodule-utils` (and its transitive `sys-libs/libsepol`) first.
5. Retry pass: the provider builds, `selinux-base` recompiles, and the 300+ downstream `selinux-*` packages never fail — the discovery is persisted before their turn.

Because the discovery carries structured evidence (the symbol, phase, exit code, and log excerpt), the printer proposes a Gentoo Bugzilla bug report draft at the end of the build for every dependency worked around this session — the record doubles as an upstream ebuild/eclass bug report (see Chapter 19).

16.5.5 USE-enable feedback

A closely related gap is when the provider *is* declared and even installed, but was built with the wrong USE set — e.g. `KX11Extras: No such file or directory` because `kde-frameworks/kwindowsystem` was merged `-X` on a headless profile (portage-ng#110). Re-adding a bare `cat/name` BDEPEND (the #102 path) is a no-op: the package is already in the plan/VDB. What is missing is a HARD `[flag]` usedep — an unconditional USE dependency, as opposed to a `[flag?]` / `[flag=]` one that follows the consumer (terminology box, Chapter 13).

`useenable.pl`, gated by `config:use_enable_feedback/1`, mirrors the #102 three seams: detect a compile/configure symbol, resolve it via a curated seed table to `Provider + HARD usedeps`, record a durable `feedback:discovered_usedep/4`, and let `builder:build_loop/2` re-derive. On the next proof `query.pl` unions a `package_dependency(..., UseDeps)` edge so the existing BWU / `bwu_force` machinery rebuilds the provider with the flag. Plans stay derived — the hook never writes `/etc/portage/package.use` itself (any `suggestion(use_change)` that the re-derived plan emits is a consequence of proving, not an imperative patch).

16.5.6 Build summary reporting

At the end of a build, the printer renders one block per mechanism that applied fixups, using the mechanism's own note:

```
Total: 46 completed.

Deconfliction: collision protection was disabled to merge 1 package over
               files owned by other installed packages (portage-ng#90):
  - app-arch/hardlink-0.3.2

GHC ABI repair: 2 broken packages rebuilt in-transaction after a
                dependency ABI-hash change (portage-ng#93, haskell-updater
equivalent):
  - dev-haskell/semialign-1.3
  - dev-haskell/semigroupoids-5.3.7

Missing provider: 1 package had an undeclared build dependency discovered at
                  build time and learned as BDEPEND (portage-ng#102):
  - sec-policy/selinux-base
```

Followed, for a missing-provider discovery, by the bug report draft:

```
>>> Missing build dependencies discovered (bug report drafts)

---
Summary: sec-policy/selinux-base: missing BDEPEND=sys-apps/semodule-utils (command
semodule_package not found)

Affected package: portage://sec-policy/selinux-base
Missing dependency: sys-apps/semodule-utils (build-time / BDEPEND)
Observed:
  command semodule_package not found during the compile phase (exit 127):
    semodule_package: command not found
Potential fix (suggestion):
  Add BDEPEND="sys-apps/semodule-utils" to the ebuild or the responsible inherited
eclass.
  (discovered by portage-ng missing-provider feedback, portage-ng#102)
```

16.6 Live build display

During a `--merge` run, portage-ng keeps the terminal display up-to-date so you can see exactly where the build process stands at any moment. The static plan that was printed during the `--pretend` phase is reprinted once, and below it a live "Executing" area shows the current state of every active build slot.

The following example shows the pretend output for `sys-kernel/gentoo-sources`. The plan has three steps: download the source tarball plus patches, install the package, and register the runtime phase.

```
>>> Emerging : portage://sys-kernel/gentoo-sources-6.19.11:run?{[]}
```

```
These are the packages that would be merged, in order:

Calculating dependencies... done!

 └─[step  1]─┤ download  portage://sys-kernel/gentoo-sources-6.19.11
             │           └─ file ─┤ 877.73 Kb   genpatches-6.19-10.base.tar.xz
             │                    │ 4.22 Kb      genpatches-6.19-10.extras.tar.xz
             │                    │ 148.84 Mb    linux-6.19.tar.xz

 └─[step  2]─┤ install   portage://sys-kernel/gentoo-sources-6.19.11
             │           └─ conf ─┤ USE = "build symlink -experimental"
             │                    │ SLOT = "6.19.11"

 └─[step  3]─┤ run       portage://sys-kernel/gentoo-sources-6.19.11

Total: 3 actions (1 download, 1 install, 1 run), grouped into 3 steps.
       149.70 Mb to be downloaded.
```

When the same target is merged with `--merge`, the display turns into a live view. Each step gains a phase line showing the individual ebuild phases and their current state. A snapshot mid-build might look like this:

```
These are the packages being merged, in order:

Executing 3 actions, grouped into 3 steps...

 └─[step  1]─┤ download  portage://sys-kernel/gentoo-sources-6.19.11    ✓
             │           └─ file ─┤ 877.73 Kb   genpatches-6.19-10.base.tar.xz     ✓
             │                    │ 4.22 Kb      genpatches-6.19-10.extras.tar.xz
✓
             │                    │ 148.84 Mb    linux-6.19.tar.xz
✓

 └─[step  2]─┤ install   portage://sys-kernel/gentoo-sources-6.19.11    ⣿
             │           └─ exec ─┤ ACTION = setup → unpack → prepare   (42%) 2/7
             │                    │ LOG = /var/log/portage/sys-kernel:gentoo-
sources.log

 └─[step  3]─┤ run       portage://sys-kernel/gentoo-sources-6.19.11
```

In this snapshot, step 1 (download) has completed — each file shows a green check mark on the right edge. Step 2 (install) is active: the action line shows a spinning indicator, and the phase line reveals that `setup` and `unpack` have finished (shown in cyan) while `prepare` is the current phase. The right edge displays the accumulated progress (`42%`) and a phase counter (`2/7`). Step 3 (run) is still pending in dark grey, waiting for the install to finish.

16.6.1 Slot states and colours

Each slot in the live display represents one concurrent build. The slot line changes colour and icon as the build progresses:

State	Colour	Indicator
Pending	Dark grey	Waiting for prerequisites
Active	Cyan (action) + green (target)	Spinning indicator on the right edge
Done	Green	Check mark
Failed	Red	Exclamation mark
Stub	Grey	Phase skipped (already satisfied)

16.6.2 Per-ebuild phase tracking

Below each slot line, the display shows the individual ebuild phases (setup, unpack, prepare, configure, compile, install, merge — or `qmerge` on the binary path) with their current status. Each phase word is coloured independently:

- **Dark grey** — pending (not yet started)
- **Cyan** — active or in progress
- **Green** — completed successfully
- **Red** — failed

The builder tracks phase state through `builder:exec_phase_state/3`, which is updated by a callback from the ebuild execution module as each phase starts, progresses, and finishes.

16.6.3 Progress indicators

portage-ng shows progress at multiple levels:

- **Per-phase percentage** — during long phases like `compile`, the builder polls the build log every 0.5 seconds and computes a progress estimate. This blends two signals: the growth of the log file (bytes written) and historical data from previous builds of the same package (stored in `Knowledge/phase_stats.pl`).
- **Overall progress** — the right edge of the display shows an accumulated percentage and a counter (`Current/Total`) reflecting how many actions have completed out of the total plan. Stub actions (already satisfied) are excluded from the total.
- **Download progress** — for parallel downloads, each file shows a percentage and transfer speed. Git clones show a separate percentage based on the git progress output.

16.6.4 Log file locations

Each build action writes its output to a log file. The path is computed from the build log directory (`config:build_log_dir/1`) and the ebuild name. When `--logs` is enabled, the log path is displayed below the phase line for each slot. If a phase fails, the log path turns red so you can quickly find the relevant output.

16.6.5 Terminal refresh

The live display uses ANSI cursor movement to update individual lines in place: the builder moves the cursor up to the target line, redraws it, and moves back down. This avoids flooding the terminal with repeated full-screen redraws. All display mutations go through a `build_display` mutex to prevent concurrent workers from interleaving their output.

In non-TTY environments (e.g. CI pipelines), cursor movement is disabled and the builder falls back to sparse status lines.

16.7 Build time estimation

The `buildtime.pl` module predicts build duration from two data sources:

1. **VDB sizes** — the installed file sizes from `/var/db/pkg/*/SIZE` correlate with build complexity.
2. **emerge.log history** — historical build times from `/var/log/emerge.log` provide empirical timing data for packages that have been built before.

The `--estimate` CLI option shows predicted build times in the plan output.

16.8 Jobserver

The `jobserver.pl` module manages parallel build execution. It implements a token-based jobserver that limits concurrent builds to the number of available cores (or a user-specified `--jobs` count).

16.9 Download management

The `download.pl` module handles source archive fetching:

- Mirror layout detection via `curl`
- Parallel downloads across multiple mirrors
- Hash verification via `openssl dgst`
- Resume support for interrupted downloads

Downloads are scheduled as early as possible in the plan — `:download` actions have no unmet requirements, so they land in the earliest waves and packages can download while others are building.

16.10 Snapshot support

Upgrades can go wrong — a new version may fail to compile, introduce regressions, or break other packages. portage-ng's snapshot module (`Source/Pipeline/Builder/snapshot.pl`) lets you freeze the current system state before a merge and roll back to it afterwards.

16.10.1 How a snapshot is created

When a merge begins with `--snapshot` (or with `config:snapshot_enabled` asserted in the per-machine config — snapshots are disabled by default), portage-ng creates a snapshot identified by a timestamp (e.g. `20260405-143012`). The snapshot directory contains three files:

- **`manifest.pl`** — a Prolog fact file listing every package currently installed in the VDB, with category, name, version, and slot.
- **`world`** — a copy of the current world set file, so the set of explicitly requested packages can be restored exactly.
- **`actions.pl`** — the planned actions for the merge, recorded so that a rollback knows which packages were touched.

16.10.2 Quickpkg: preserving the old version

The key to rollback is preserving the **binary package** of each package that is about to be replaced. Before portage-ng merges a new version, the builder calls `snapshot:quickpkg_old/2`. This runs `ebuild --skip-manifest <old-ebuild> package` with `PKGDIR` pointed at the snapshot's `binpkgs/` directory. The result is a tarball (`.tbz2` or `.gpkg.tar`) that contains the currently installed files of the old version — essentially the same operation that Gentoo's `quickpkg` tool performs.

Because this happens **per package, just before the upgrade**, the snapshot accumulates exactly the set of binary packages needed to reverse the merge. Packages that were not touched are not quickpkg'd; they remain unchanged on the system.

16.10.3 Listing and diffing snapshots

`--snapshots` shows all available snapshots with their timestamp, installed package count, and the number of binary packages stored:

```
Available snapshots:
  20260405-143012       2026-04-05 14:30:12   1847 pkgs   12 binpkgs
  20260402-091544       2026-04-02 09:15:44   1843 pkgs    5 binpkgs
```

`--rollback <id> --pretend` compares a snapshot's manifest against the current VDB and shows what changed — packages installed since the snapshot, packages removed, and packages whose version changed:

```
Diff against snapshot "20260405-143012":

  Installed since snapshot (2):
    + dev-libs/newlib-4.5.0
    + dev-util/newtool-1.0

  Version changed since snapshot (3):
    ~ sys-libs/glibc  2.40-r2 -> 2.41
    ~ dev-lang/python  3.12.8 -> 3.13.1
    ~ app-editors/vim  9.1.1652-r2 -> 9.1.1700
```

```
Summary: +2 −0 ~3 (3 binpkgs available for rollback)
```

16.10.4 Rolling back

`--rollback <id>` reinstalls the saved binary packages from the snapshot's `binpkgs/` directory and restores the world set file. Each binary package is merged back onto the system via `ebuild <binpkg> merge`, downgrading the affected packages to their pre-upgrade versions. Combined with `--pretend`, the rollback shows what would be reinstalled without actually making changes.

16.10.5 Lifecycle

After the merge completes (whether successfully or not), the snapshot remains on disk so it can be used for rollback at any later time. There is no delete flag; to reclaim disk space, call `snapshot:delete(Id)` from `--shell` (or remove the snapshot directory by hand).

16.11 Further reading

- Why not our own `ebuild`? (this chapter) — design rationale for delegating phase execution
- The ebuild contract (this chapter) — invocation, environment, outcomes, and what a replacement backend must provide
- Chapter 13: Ordering — Plans as Proofs — how the plan is constructed
- Chapter 14: Output and Visualization — plan display and `.merge` file generation
- Chapter 15: Command-Line Interface — `--merge`, `--jobs`, `--estimate` flags
- `Source/Domain/Gentoo/Ebuild/ebuild_exec.pl` and `Source/Domain/Gentoo/Binpkg/binpkg_exec.pl` — implementation of the contract
- `Source/Application/Security/sanitize.pl` — argv / phase allowlist rules shared with download and other spawn sites

Chapter 17

Semantic Search and LLM Integration

portage-ng integrates with large language models for three purposes: **semantic search** over the knowledge base using vector embeddings, **plan explanation** using natural-language generation, and **metacircular self-repair** that proposes build-time learning from failed build logs (human-confirmed `feedback:*` records, optional fixup drafts).

17.1 Semantic search

The semantic search module (`Source/Application/Llm/semantic.pl`) enables natural-language queries over the package knowledge base.

17.1.1 How it works

1. Package descriptions are converted to vector embeddings via Ollama's embedding API (default endpoint: `http://localhost:11434`).
2. Embeddings are stored in an in-memory index.
3. At query time, the search query is embedded and compared against all package embeddings using cosine similarity.
4. Results are ranked by similarity score.

17.1.2 Usage

```
# Natural-language search
portage-ng --search "text editor with syntax highlighting"

# Find packages similar to a known package
portage-ng --similar app-editors/neovim
```

On Apple Silicon, Ollama leverages the GPU and Neural Engine for accelerated embedding computation.

17.1.3 Prerequisites

Semantic search requires: - A running Ollama instance - A loaded embedding model (configured via `config:semantic_model/1`)

17.2 LLM-assisted plan explanation

The `--explain` / `--llm` flags send proof artifacts to an LLM for human-readable interpretation of build plans and assumptions.

17.2.1 Provider backends

portage-ng supports multiple LLM providers, each implemented as a separate module in `Source/Application/Llm/`:

Module	Provider	Notes
`ollama.pl`	Ollama	Local inference; also provides embeddings
`claude.pl`	Anthropic Claude	Requires API key
`chatgpt.pl`	OpenAI ChatGPT	Requires API key
`gemini.pl`	Google Gemini	Requires API key
`grok.pl`	xAI Grok	Requires API key

The default provider is set via `config:llm_default/1`. API keys and endpoints are configured in `Source/config.pl` or via `Source/Config/Private/` template files.

17.3 Calling LLMs

portage-ng offers three ways to interact with an LLM.

17.3.1 Interactive chat (`--llm`)

The `--llm` flag opens an interactive chat session with the default (or a named) LLM service. The session maintains conversation history, so follow-up questions have context:

```
# Chat with the default LLM (configured in config.pl)
portage-ng --llm

# Chat with a specific service
portage-ng --llm claude
portage-ng --llm ollama
```

Inside the session, the LLM streams its response word by word to the terminal. Type `quit` or `exit` to leave.

17.3.2 Plan explanation (`--explain`)

The `--explain` flag resolves a target first, then feeds the full build plan to the LLM as structured context. The user can then ask questions about the plan in a conversational loop:

```
# Explain a build plan interactively
portage-ng --pretend --explain dev-libs/openssl
```

```
# Single-shot question
portage-ng --pretend --explain "Why is zlib in the plan?" dev-libs/openssl
```

The plan context includes targets, every package with its action type and USE flags, dependency chains traced back to the root, and any assumptions the prover made. The LLM sees the full picture and can answer questions like "why is this package included?", "what would happen if I disabled this USE flag?", or "which assumptions were made?"

17.3.3 Programmatic access

From the Prolog shell, any LLM can be called directly:

```
% Call a specific service
claude("Explain what dev-libs/openssl does", Response).
ollama("What is a USE flag?", Response).
grok("Compare DEPEND and RDEPEND", Response).

% Or use the generic dispatcher
explainer:call_llm(claude, "Your question here", Response).
```

Each service maintains its own conversation history, so subsequent calls build on previous context.

17.4 Grounding: how the LLM learns portage-ng

Ordinary chat injects short `config:llm_capability/2` primers (`context`, `architecture`, `chat`, `code`). Those are not enough to invent APIs safely, so the LLM is steered to a **read-only knowledge pack**:

Predicate	Purpose
`llmknowledge:list_topics/0`	Catalogue curated digests
`llmknowledge:print_topic/1`	`architecture`, `proof`, `assumptions`, `learning`, `code_map`, …
`llmknowledge:print_handbook/1`	Bounded Handbook chapter (`prover`, `rules`, `building`, …)
`llmknowledge:print_source/3`	Whitelisted `Source/…` or Handbook excerpt by line range

Call these from `<call:swi_prolog>` (sandboxed). Paths outside the whitelist and `..` traversal are rejected. Caps: `config:llm_knowledge_max_bytes/1`, `config:llm_knowledge_max_source_lines/1`.

Metacircular `--diagnose` additionally injects the `learning` and `code_map` digests into its prompt. A future step can embed Handbook + source for vector retrieval; the knowledge pack is the always-on path that does not require Ollama.

Module: `Source/Application/Llm/knowledge.pl`.

17.5 Code execution in the sandbox

One of portage-ng's most distinctive LLM features is that an LLM can **execute Prolog code** inside the running system and receive the output. This turns the LLM from a passive question-answerer into an active investigator that can query the knowledge base, inspect proof artifacts, and verify its own answers.

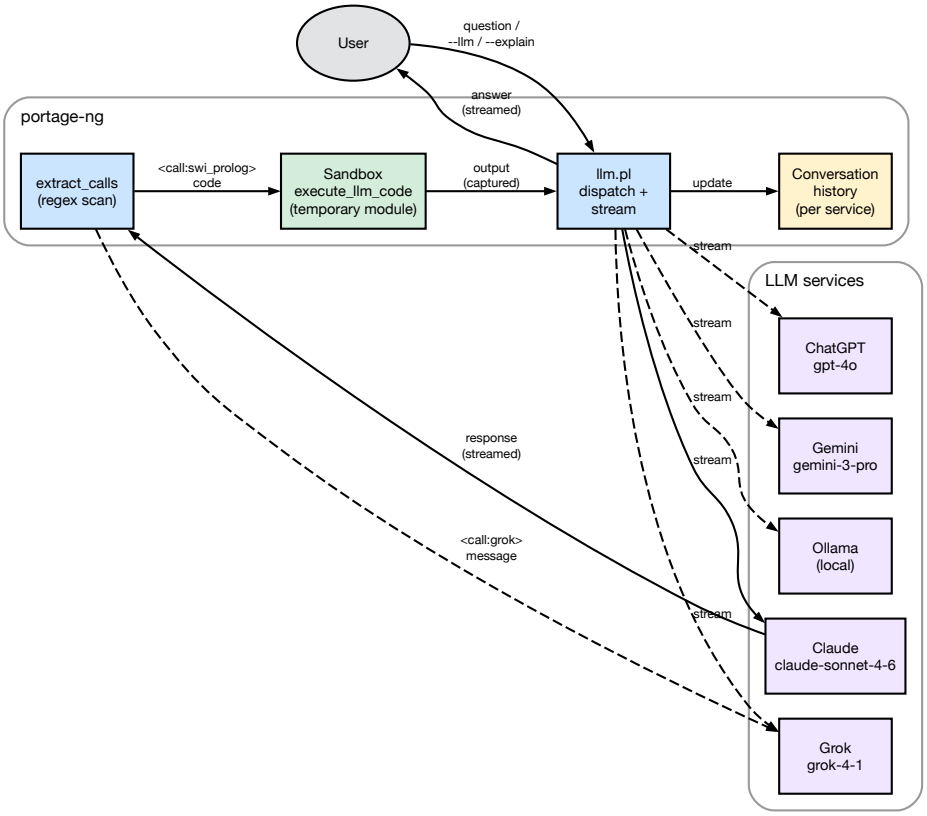


Figure 46 — LLM interaction and code execution

17.5.1 How it works

When an LLM responds, portage-ng scans the response for XML-style call tags. Two kinds are recognized:

- `<call:swi_prolog>` **...** `</call:swi_prolog>` — the enclosed Prolog code is executed in a temporary, sandboxed module. The output (both standard output and error) is captured and sent back to the LLM as a function response.
- `<call:claude>` **/** `<call:grok>` **/ etc.** — the enclosed message is forwarded to another LLM service, and that service's response is sent back. This allows LLMs to consult each other.

The LLM is informed of these capabilities through a system prompt assembled from `config:llm_capability/2` clauses. The prompt explains the available tags and how to use them, without the user needing to know about the mechanism.

17.5.2 Sandbox safety

Code execution is sandboxed by default (`config:llm_sandboxed_execution(true)`). The code runs in a temporary module created by SWI-Prolog's `in_temporary_module/3`, with `sandboxed(true)` ensuring that only safe predicates are accessible. The module is destroyed after execution. This means the LLM can:

- Query the knowledge base (`cache:ordered_entry/5`, `query:search/2`)
- Inspect proof artifacts if they are in scope
- Perform computations and formatting
- Write output that gets captured and returned

But it **cannot** modify the file system, execute shell commands, or alter the running system's state.

17.5.3 Example interaction

The diagram below traces a single round trip. The user asks a question; portage-ng forwards it (with a system prompt listing the available capabilities) to the LLM. The LLM responds with embedded Prolog code wrapped in a `<call:swi_prolog>` tag. portage-ng detects the tag, executes the code in a sandboxed temporary module, captures the output, and sends it back to the LLM as a function result. The LLM then incorporates the verified fact into its final answer, which is streamed back to the user.

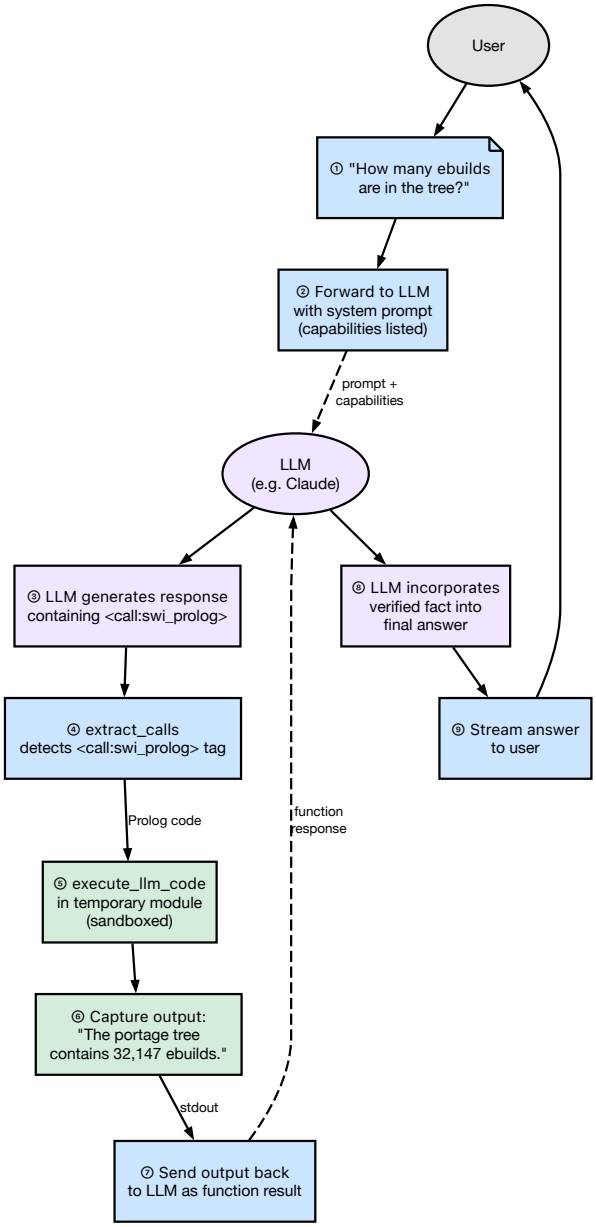


Figure 47 — Code execution round trip

Concretely, a user asks "How many ebuilds are in the tree?" The LLM responds with:

```
<call:swi_prolog>
:- aggregate_all(count, cache:ordered_entry(portage, _, _, _, _), N),
   format('The portage tree contains ~w ebuilds.~n', [N]).
</call:swi_prolog>
```

portage-ng executes this code, captures the output ("The portage tree contains 32,147 ebuilds."), and sends it back to the LLM. The LLM then incorporates this verified fact into its final answer to the user.

17.5.4 Inter-LLM communication

The same tag mechanism allows LLMs to delegate questions to each other. The diagram below shows the flow: the primary LLM (Claude in this example) embeds a `<call:ollama>` tag in its response. portage-ng detects the tag, forwards the enclosed message to Ollama, collects Ollama's response, and sends it back to Claude as a function result. Claude then weaves both perspectives into its final answer.

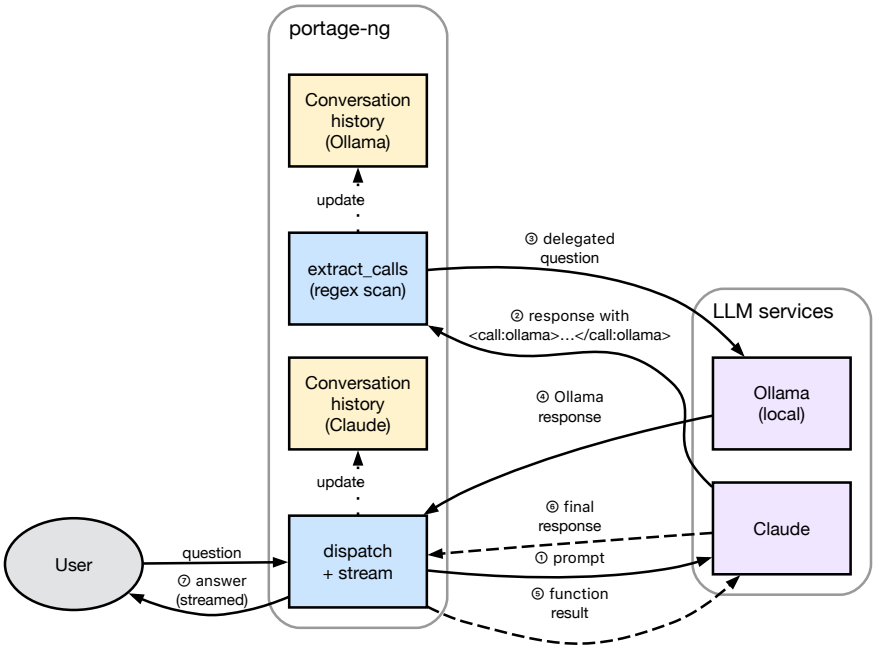


Figure 48 — Inter-LLM communication

For example, a Claude session might embed `<call:ollama>Summarize this dependency chain</call:ollama>` to get a local model's perspective, or `<call:grok>What is the latest version of openssl?</call:grok>` to cross-check information. Each delegated call maintains the target service's own conversation history.

17.6 Suggestions and explanations

Beyond answering questions, the LLM integration supports two practical workflows: **plan explanation** and **failure diagnosis**.

17.6.1 Plan explanation

When `--explain` is active, portage-ng assembles a structured context from the proof artifacts that includes:

- **Targets** — the packages the user requested
- **Plan entries** — every package in the merge plan with its step number, action type (install/run/download), USE flags (with annotations for user-set, profile-forced, and resolver-changed flags), slot information, and dependency chain

- **Reverse dependency paths** — for each package, the chain of dependencies tracing back to the root target ("zlib is needed by openssl, which is needed by python, which is the target")
- **Assumptions** — any domain assumptions or cycle breaks, with classified reasons (missing package, masked, keyword-filtered, slot mismatch, version conflict)

This context allows the LLM to give precise, grounded answers rather than generic advice. When a user asks "why is zlib in the plan?", the LLM can point to the exact dependency chain rather than guessing.

17.6.2 Failure diagnosis

When the resolver cannot satisfy a dependency, it records an `assumption_reason` in the proof context. The explainer's `assumption_reason_for_grouped_dep/6` predicate diagnoses the failure by progressively filtering candidates through existence checks, mask checks, slot restrictions, version constraints, and keyword acceptance. The classified reason (e.g. `missing`, `masked`, `keyword_filtered`, `version_conflict`) is attached to the assumption and included in the LLM context.

When the `--llm` flag is combined with a failing target, portage-ng sends a specialized diagnostic prompt (`config:llm_support/1`) to the LLM, providing the failure details and asking for help identifying the correct package atom or diagnosing the issue.

17.7 Metacircular self-repair

When a build phase fails and deterministic fixups (`fixup:phase_retry_hook/10`) do not mint new `feedback:*` knowledge, portage-ng can ask the configured LLM to diagnose the failure and propose structured repair actions. The module is `Source/Application/Llm/metacircular.pl`.

17.7.1 Loop

1. Builder records failed `Repo://Entry` + log path (`builder:last_failed/3`).
2. If `should_replan` would not fire (no new deterministic discoveries), and `config:llm_metacircular(true)` with an interactive TTY (not `--ci`), metacircular assembles context: plan summary, polarity-tagged assumptions, fallback tier, feedback backlog, learned `cn_domain`, and a bounded log tail.
3. The LLM receives `config:llm_capability(metacircular, …)` (excluded from ordinary `--llm` chat prompts) and must reply with a single term:

```
repair_proposal([
  action(record_discovery, Repo://Entry, 'cat/pkg', bdepend, Evidence),
  action(record_usedep, Repo://Entry, 'cat/pkg', [use(enable(foo),none)], Evidence),
  action(record_excluded_version, Cat, Name, Ver, Evidence),
  action(record_kernel_config, Repo://Entry, ['CONFIG_FOO'], Evidence),
  action(draft_fixup, MechanismName, Synopsis, SketchBody)
]).
```

4. Actions are validated (in-tree providers only; at most `config:llm_metacircular_max_actions/1`, default 3). Each accepted action is confirmed in-

teractively; confirmed feedback writes go through `feedback:record_*` so the existing replan loop re-derives the plan.

5. `draft_fixup` writes a sketch under `Knowledge/drafts/` (gitignored); it is never auto-loaded. A human must review and commit under `Source/Domain/Gentoo/Exceptions/`.

17.7.2 Safety model

- Mutations happen **host-side after confirm**, never via `<call:swi_prolog>`.
- Sandbox may **read** `feedback:*`, `missing_provider:package_in_tree/1`, and parse helpers; it cannot call `feedback:record_*`.
- New package edges belong in `feedback:*`, not `prover:learn/3` (version/USE domains only).
- Kill-switch: `config:llm_metacircular(false)`.
- Skip loading LLM modules entirely: `config:load_llm_modules(false)`. Builder and CLI then no-op diagnose paths (stubs + soft `explainer:call_llm/3`); no existence errors.
- **`explainer:call_llm/3` on the server is opt-in.** Default `config:llm_server_calls(false)` keeps it off the Pengines safelist so authenticated clients cannot burn server API keys or exfiltrate prompts. LLM features are then **host-local**: run `--explain` / `--diagnose` / `--llm` on standalone or client processes (own `Source/Config/Private/api_key.pl`). To allow server-side LLM via RPC on a trusted cluster, set in a host config or `Source/Config/Private/`:

  ```
  :- asserta(config:llm_server_calls(true)).
  ```

  The server must also load LLM modules (`config:load_llm_modules(true)`) and hold valid keys in its `api_key.pl`.

17.7.3 CLI

```
# Offline diagnose of a failed package (uses default log path)
portage-ng --diagnose cat/pkg

# Explicit log
portage-ng --diagnose --log /var/tmp/portage-ng/logs/cat--pkg-1.2.3.log cat/pkg
```

During an interactive `--build`, diagnose runs automatically after a failed pass with no new deterministic discoveries (same confirm gate).

17.8 Explainer architecture

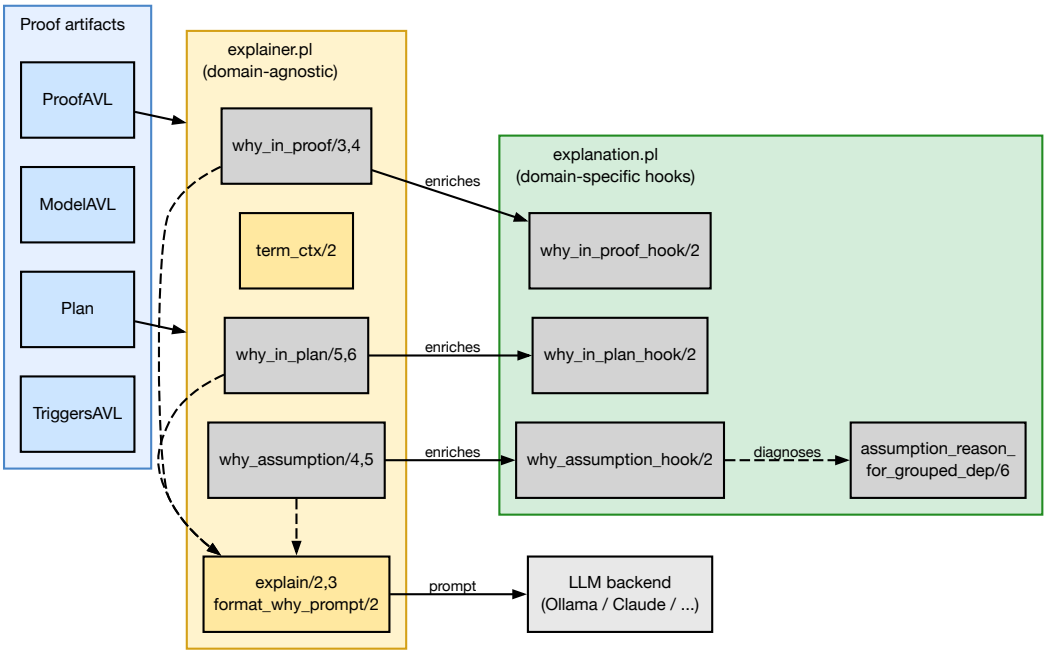


Figure 49 — Explainer architecture

The explainer is split into two modules with clearly separated responsibilities.

17.8.1 Domain-agnostic introspection

`explainer.pl` answers "why" questions by inspecting the proof artifacts (ProofAVL, ModelAVL, Plan, TriggersAVL) without knowing anything about Gentoo or Portage. It provides three families of queries:

- **`why_in_proof/3,4`** — given a literal, look it up in the ProofAVL and report how it was proved: via a normal rule, a domain assumption, or a prover cycle-break. Returns the body literals and the proof-term context.
- **`why_in_plan/5,6`** — given a literal and a plan, locate it in the wave schedule and trace a reverse-dependency path (via TriggersAVL) back to a root target. The result explains *why* this package is in the plan (e.g. "required by X, which is required by the target Y").
- **`why_assumption/4,5`** — given an assumption key, classify it (domain assumption, cycle-break, or model-only) and extract any `assumption_reason` tags from the context.

The utility predicate **`term_ctx/2`** extracts the `?{Ctx}` context list from any literal-shaped term. This is how the explainer accesses the structured tags (USE state, slot info, suggestions, assumption reasons) attached to each literal during the proof.

17.8.2 Domain-specific hooks

`explanation.pl` provides **enrichment hooks** that inject Gentoo-specific context into the generic Why terms produced by `explainer.pl`. The hooks are multifile predicates, so the domain layer plugs into the generic layer without modifying it:

- `why_in_proof_hook/2` — if the proof context contains domain reasons (masking info, keyword filtering, slot constraints), it appends a `domain_reasons(Reasons)` tag to the Why term.
- `why_in_plan_hook/2` — reserved for future plan-level Gentoo annotations (currently identity).
- `why_assumption_hook/2` — enriches assumption Why terms with domain reasons extracted from the assumption's context.

The module also provides `assumption_reason_for_grouped_dep/6`, which diagnoses *why* a grouped dependency resolution failed. It inspects the candidate pool and classifies the failure (missing package, all candidates masked, keyword-filtered, slot mismatch, version conflict, etc.). This diagnosis is cached and feeds the `assumption_reason` tags that appear in the proof context.

17.8.3 LLM dispatch

Both modules produce structured Prolog terms. When the user wants a human-readable explanation, the `explain/2,3` predicates in `explainer.pl` convert the structured Why term into a text prompt via `format_why_prompt/2` (which uses `term_to_atom/2` and prepends a system preamble), then dispatch it to an LLM backend via `call_llm/3`. The LLM backend is configurable (`config:llm_default/1`) and can be Ollama, Claude, or any other service that implements the expected interface.

A separate, richer LLM path exists in `explain.pl` (`Source/Application/Llm/explain.pl`), which builds a multi-section text context from the entire plan (targets, actions, USE flags, assumptions) and sends it as a conversational prompt. This is used by `--explain` and the interactive `explain_plan_interactive/5` mode, where the user can ask follow-up questions about the plan.

17.9 Query families

Three families of queries are supported:

- **why_in_proof**: given a literal, find how it was proven (normal rule, domain assumption, or prover cycle-break) and extract its body/deps.
- **why_in_plan**: given a literal and a plan, locate it in the wave-plan and trace a reverse-dependency path (via TriggersAVL) back to a root.
- **why_assumption**: given an assumption key, classify it (domain vs cycle-break vs model-only) and extract any reason tags.

17.10 Usage

All predicates are called with the `explainer:` module prefix.

17.10.1 Step 1: Obtain proof artifacts

Run the pipeline to get the proof, model, plan, and triggers:

```
Goals = [portage://'dev-libs/openssl-3.5.4':run?{[]}],
pipeline:prove_plan_with_fallback(Goals, ProofAVL, ModelAVL, Plan, TriggersAVL).
```

Or from a `--shell` session after loading a repository:

```
pipeline:prove_plan_with_fallback([portage://'dev-libs/openssl-3.5.4':run?{[]}],
                                  Proof, Model, Plan, Triggers).
```

17.10.2 Step 2: Ask "why" questions

Why is a package in the proof?

```
Target = portage://'dev-libs/libffi-3.5.2':install,
explainer:why_in_proof(ProofAVL, Target, Why).
% Why = why_in_proof(
%          portage://'dev-libs/libffi-3.5.2':install,
%          proof_key(rule(portage://'dev-libs/libffi-3.5.2':install)),
%          depcount(3),
%          body([portage://'sys-devel/gcc-15.2.0':install, ...]),
%          ctx([...]),
%          domain_reasons([...]))      % <-- added by explanation hook
```

Why is a package in the plan?

```
Proposal = [portage://'dev-libs/openssl-3.5.4':run?{[]}],
explainer:why_in_plan(Proposal, Plan, ProofAVL, TriggersAVL,
                      portage://'sys-libs/zlib-1.3.1-r1':install, Why).
% Why = why_in_plan(
%          portage://'sys-libs/zlib-1.3.1-r1':install,
%          location(step(1), portage://'sys-libs/zlib-1.3.1-r1':install?{...}),
%          required_by(path([portage://'sys-libs/zlib-1.3.1-r1':install,
%                            portage://'dev-libs/openssl-3.5.4':install,
%                            portage://'dev-libs/openssl-3.5.4':run])))
```

Why is something assumed?

```
Key = assumed(portage://'dev-foo/bar-1.0':install),
explainer:why_assumption(ModelAVL, ProofAVL, Key, Type, Why).
% Type = domain,
% Why  = why_assumption(
%          assumed(portage://'dev-foo/bar-1.0':install),
%          type(domain),
%          term(portage://'dev-foo/bar-1.0':install?{[assumption_reason(missing)]}),
%          reason(missing),
%          domain_reasons([...]))      % <-- added by explanation hook
```

17.10.3 Step 3 (optional): Get a human-readable explanation via LLM

```
explainer:why_in_proof(ProofAVL, Target, Why),
explainer:explain(claude, Why, Response).
```

```
% Response = "openssl requires libffi as a build dependency because..."

% Or use the default LLM (from config:llm_default/1):
explainer:explain(Why, Response).
```

Available LLM services: claude, grok, chatgpt, gemini, ollama. The default is set via `config:llm_default/1`. See `config.pl` for API keys, models, and endpoints.

17.11 Assumption diagnosis

`explanation:assumption_reason_for_grouped_dep/6` is called on the fallback path when no candidate satisfies all constraints. It progressively filters candidates through:

1. Existence check → `missing`
2. Self-hosting restriction → `installed_required`
3. Mask check → `masked`
4. Slot restriction → `slot_unsatisfied`
5. Version constraints → `version_no_candidate(0,V)` / `version_conflict`
6. ACCEPT_KEYWORDS → `keyword_filtered`
7. Fallback → `unsatisfied_constraints`

Example:

```
explanation:assumption_reason_for_grouped_dep(
  install,                                        % Action
  'dev-libs', 'missing-pkg',                      % Category, Name
  [package_dependency(install,no,'dev-libs','missing-pkg',
                      none,version_none,[],[])],
  [self(portage://'app-misc/foo-1.0')],           % Context
  Reason).
% Reason = missing
```

17.12 Hook mechanism

The explainer module calls `explanation:why_*_hook(Why0, Why)` after building its generic Why term. If the hook succeeds, the enriched Why replaces the generic one. Each hook extracts `domain_reason(cn_domain(C, N, Tags))` tags from the proof context and appends them as `domain_reasons(Reasons)`.

The hooks are called automatically — no direct invocation needed.

17.13 Further reading

- Chapter 9: Assumptions and Constraint Learning — assumption taxonomy that the explainer queries
- Chapter 8: The Prover — proof artifacts consumed by the explainer
- Chapter 15: Command-Line Interface — `--search`, `--similar`, `--explain` flags

Chapter 18

Distributed Proving

For testing purposes, the full Gentoo tree must be walked through the resolver — tens of thousands of ebuilds — and even on capable hardware that adds up. On a single machine, `resolver:test_stats(portage)` typically finishes proving every single ebuild in **a few minutes** on a twenty-eight-core workstation — fast enough for day-to-day development, but not the only shape the problem takes.

What if you want the full tree proved **faster** than one box allows, or you want to drive resolution from a **thin client** that does not carry the whole knowledge base? portage-ng answers with a **client–server–worker** architecture: a **server** holds the Portage knowledge base and hands out proof jobs; **workers** pull work, run the proving pipeline, and push results back; **clients** submit targets and collect outcomes over the network. The sections below explain how that stack is wired, how discovery and TLS secure it, and how the same repository abstractions work whether you run standalone, on the server, or on a worker.

18.1 Architecture

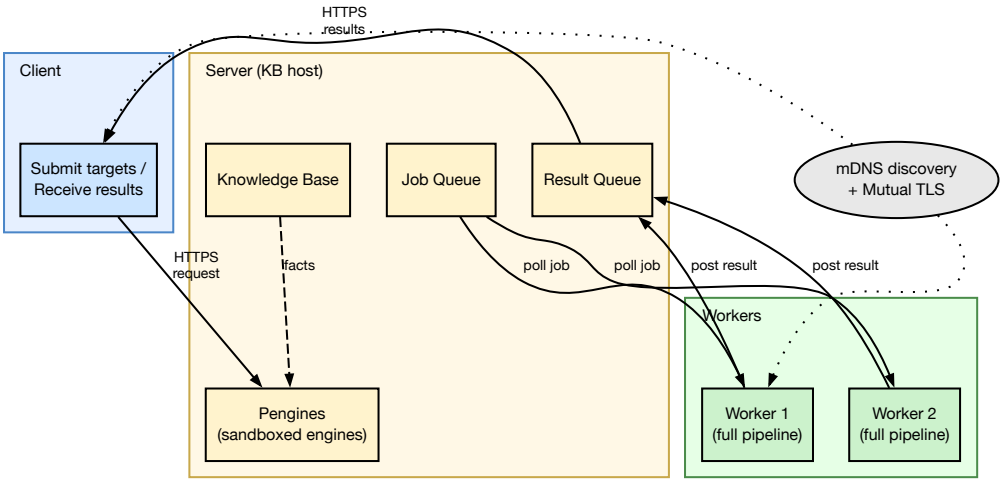


Figure 50 — Cluster architecture

The diagram above shows the three roles. The **server** is the central hub: it holds the in-memory knowledge base, exposes an HTTP interface via Pengines (described below), and multiplexes proof jobs across workers through a job queue and result queue.

Workers are symmetric compute nodes. Each worker carries its own copy of the knowledge base and runs the full proving pipeline locally. You can add as many workers as you need — they scale horizontally, and the server distributes work evenly.

The **client** submits target packages and collects results. It does not need the knowledge base itself; it talks to the server over HTTPS and receives completed proofs as JSON.

18.1.1 Interaction flow

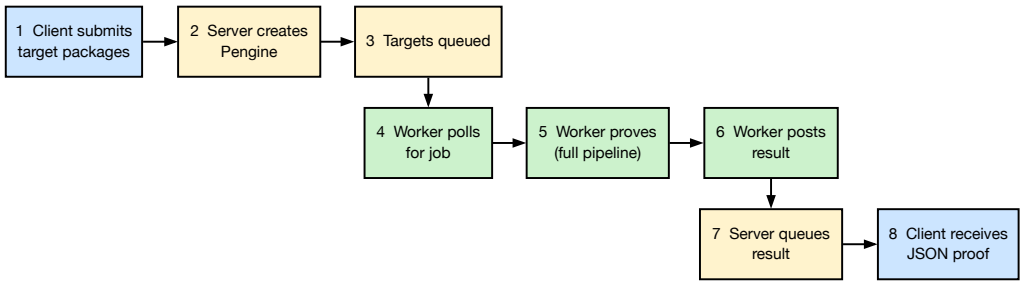


Figure 51 — Cluster interaction flow

The numbered steps show how a proof request travels through the system:

Steps 1-3 (client to server). The client sends a target package (e.g. `sys-apps/portage`) to the server over HTTPS. The server creates a Pengine (a sandboxed Prolog engine) to handle the request and adds the target to its job queue.

Steps 4-6 (worker loop). Workers continuously poll the job queue for work. When a worker picks up a job, it runs the full proving pipeline locally — proof search (resolving) and ordering — using its own copy of the knowledge base. When the proof is complete, the worker posts the result back to the server.

Steps 7-8 (results back to client). The server stores the completed proof in its result queue. The client retrieves the result as a JSON document over HTTPS.

18.1.2 Server

The server is the central coordination point. It runs an HTTP server backed by SWI-Prolog's Pengines library and manages four things: the knowledge base (the full Portage tree loaded in memory), a job queue of proof targets waiting to be processed, a result queue of completed proofs, and the Pengine sandbox that controls what remote callers can execute.

18.1.3 Worker

Each worker is an independent OS process with its own Prolog VM. On startup it loads the knowledge base, then enters a poll loop: it asks the server for the next job, runs the full pipeline (resolve and order), and posts the result back. Workers are stateless between jobs, so you can add or remove them at any time without affecting other workers or the server.

18.1.4 Client

The client is a thin request layer. It does not carry the knowledge base — it simply submits target packages to the server and collects the completed proofs. This makes it suitable for lightweight machines or scripts that want to drive resolution remotely.

18.1.5 Cluster orchestration

The cluster module sits above the individual roles and provides high-level orchestration: distributing a batch of targets across available workers, collecting results as they arrive, and handling failures (retrying jobs that a worker did not complete).

18.2 Pengines: Prolog as a network service

Pengines ("Prolog engines as a web service") is a library that ships with SWI-Prolog. It turns Prolog query execution into an HTTP-friendly protocol, and portage-ng uses it as the communication layer between clients and the server.

When a client sends a proof request, the server does not run the query in its main thread. Instead, it creates a **Pengine** — a fresh, isolated Prolog engine dedicated to that interaction. The Pengine can read the shared knowledge base (the Portage tree loaded in the server process), but it runs inside a **sandbox** that prevents it from modifying the knowledge base or calling dangerous predicates. Remote callers cannot reshape the server's state.

Answers are streamed back as JSON over HTTP. This means a client does not need to be a full Prolog application — any language that can make HTTP requests and parse JSON can drive proof search. From the outside, portage-ng looks like an ordinary web service that happens to do dependency resolution internally.

18.3 Repository state across modes

An important design question for distributed proving is: how does each mode access the Portage tree? The answer is portage-ng's object-oriented context system (see Chapter 21). Each repository is an instance created through that system, and methods like `portage:read` populate the cache facts.

In **server mode**, repository instances live in the shared server process. All Pengine threads see the same instances and the same loaded knowledge — one tree in memory, many sandboxes reading it.

In **worker mode**, each worker is a separate OS process with its own Prolog VM. It creates its own repository instances and loads its own copy of the knowledge base. Nothing is shared with the server's address space.

The benefit is that the call sites are identical everywhere: the same `portage:read` call appears in standalone, server, and worker code. The object system dispatches to the right backing store depending on the mode, so distributed proving does not require a separate set of data-loading predicates.

18.4 Client installed state: `--import-vdb`

By default the server proves against its **own** VDB (the in-memory `pkg` repository loaded by its `kb:load`). For a remote client whose installed set differs from the server's, that would produce wrong `[nomerge]`, rebuild, and `replaces` decisions. The `--import-vdb` client action closes this gap:

```
$ portage-ng --mode client --import-vdb
```

The client parses its local VDB (`config:pkg_directory/1`, typically `/var/db/pkg`) through the regular repository sync path, serializes the resulting `cache:` facts as a Prolog term stream with a snapshot stamp (entry count + content hash), and POSTs the payload to the authenticated `/import-vdb` endpoint (same digest + client-certificate TLS as `/sync`).

The server validates the payload (whitelisted fact shapes, sanitized hostname, capped counts) and registers the facts atomically as a per-client repository named `pkg@<clienthost>`, recording the stamp.

At prove time, every rule that consults the installed set goes through a single accessor, `knowledgebase:vdb_repository/1`:

- In **standalone**, **server**, and **worker** mode it resolves to the local `pkg` repository — the behaviour is unchanged (the lookup is memoized per thread, so there is no hot-path cost).
- Inside a **Pengine** serving a client request it resolves to that client's `pkg@<clienthost>` repository, using the repository name and import stamp the client ships with each RPC.

Stale or missing imports are loud, never silent: if a client has not imported its VDB (or the stamp no longer matches what the server holds), the server prints an explicit warning and falls back to its own `pkg` repository, telling the user to (re-)run `--import-vdb`.

With `config:client_auto_import_vdb(true)` (the default) the import happens automatically: before the first RPC of a client command (`--pretend`, `--merge`, `--search`, …) the client re-imports its VDB whenever no import record exists for the target server or the local VDB changed since the last import. Freshness is detected with a cheap mtime check over the VDB root and its category directories, so an up-to-date import adds no noticeable overhead. Set the flag to `false` to ship the VDB only on an explicit `--import-vdb`.

Filesystem-level VDB reads (CONTENTS listings, on-disk SIZE files, binpkg live-VDB re-stat) can never work for a remote client; those paths detect a per-client repository and degrade to the imported in-memory snapshot, or are skipped. Worker mode is out of scope: workers load their own knowledge base and assume a homogeneous fleet.

18.5 mDNS/Bonjour discovery

Workers and servers discover each other automatically via **mDNS/Bonjour** service advertisement, so there is no need to hand-configure IP addresses.

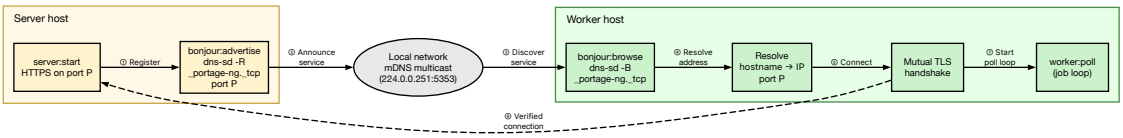


Figure 52 — Bonjour discovery flow

The discovery protocol has three steps:

1. **Register** — when the server starts, it advertises a `_portage-ng._tcp` service on the local network, making its hostname and port visible to any device on the same link.
2. **Browse** — workers browse for that service type and automatically receive the server's address and port.
3. **Connect** — a connection is established from the discovery data, with no manual configuration required.

This is the same zero-configuration networking mechanism used by AirPrint, AirPlay, and many other network services.

18.5.1 Platform support

On **macOS**, the `dns-sd` command ships with the system. The `bonjour.pl` module uses `dns-sd -R` to register the service and `dns-sd -B` to browse for peers.

On **Linux**, the same `dns-sd` command is available through **Avahi** (typically in the `avahi-utils` package). The `bonjour` module hides the platform difference behind a single Prolog interface (`subprocess:dns_sd/...`), so the rest of portage-ng does not need to know which implementation is in use.

After discovery, all traffic is encrypted and mutually authenticated via TLS (see the following sections).

18.6 Sandbox and security

Because Pengines allow remote Prolog execution, the server must control what clients can do. The sandbox module (`Source/Application/Security/sandbox.pl`) enforces a whitelist: only predicates explicitly registered as safe via `sandbox:safe_primitive/1` and `sandbox:safe_meta/2` can be called remotely. Everything else is blocked.

A separate sanitise module (`Source/Application/Security/sanitize.pl`) validates the structure of incoming queries before they reach the sandbox, rejecting malformed or unexpected input early.

18.7 TLS certificates

All communication between server, workers, and clients is encrypted and mutually authenticated using TLS. Mutual authentication means that **both sides** present a certificate during the handshake — the server proves its identity to the client, and the client proves its identity to the server. This prevents unauthorized nodes from joining the cluster.

18.7.1 Certificate hierarchy

portage-ng uses a private Certificate Authority (CA) that acts as the trust root for the entire cluster. Every host-specific certificate is signed by this CA, so any node holding the CA's public certificate can verify any other node's identity.

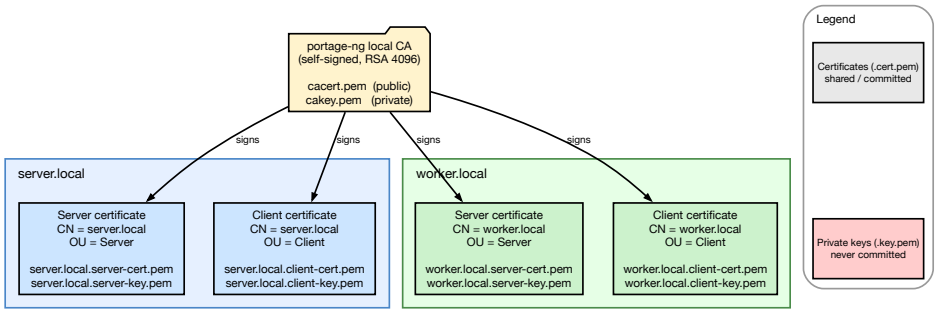


Figure 53 — Certificate hierarchy

The CA is a self-signed RSA 4096-bit certificate valid for 10 years (`CERT_DAYS=3650`). Each host receives two certificates signed by the CA:

- A **server certificate** — presented when the host runs in `--mode server`. The Common Name (CN) is set to the hostname and the Organizational Unit (OU) to "Server".
- A **client certificate** — presented when the host connects to a server as a worker or client. The CN is the local hostname and the OU is "Client".

Both certificates use RSA 2048-bit keys and SHA-256 signatures.

18.7.2 File layout

All certificate files live under the `Certificates/` directory at the project root. The table below shows which files are tracked in git (public certificates) and which are excluded (private keys):

File	Tracked	Description
`cacert.pem`	Yes	CA public certificate (shared trust root)
`cakey.pem`	No	CA private key (never distributed)
`<host>.server-cert.pem`	Yes	Server certificate for `<host>`
`<host>.server-key.pem`	No	Server private key
`<host>.client-cert.pem`	Yes	Client certificate for `<host>`
`<host>.client-key.pem`	No	Client private key
`passwordfile`	No	HTTP digest authentication passwords (`make passwordfile`)

Private keys and the CA serial file are excluded via `.gitignore`. The public certificates are committed so that nodes can verify each other without manual file copying — only the shared `cacert.pem` needs to be distributed out-of-band.

18.7.3 Mutual TLS handshake

When a worker or client connects to the server, a mutual TLS handshake takes place. Both sides verify the other's certificate against the shared CA before any data is exchanged.

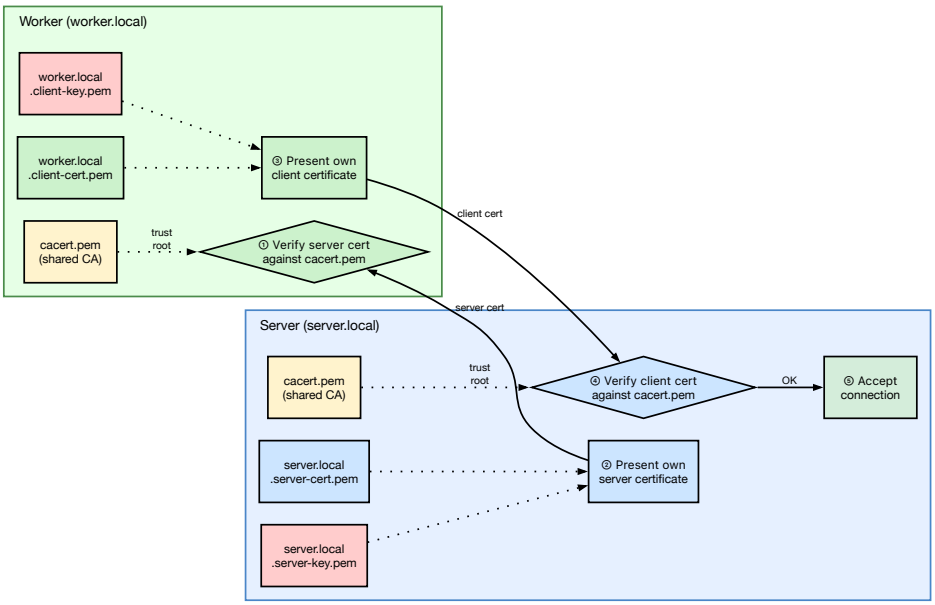


Figure 54 — Mutual TLS handshake

The handshake proceeds in five steps:

1. The **server** presents its server certificate (`server.local.server-cert.pem`), signed by the CA.
2. The **worker** verifies this certificate against its local copy of `cacert.pem`. If verification fails (wrong CA, expired, CN mismatch), the connection is refused.
3. The **worker** presents its client certificate (`worker.local.client-cert.pem`), also signed by the same CA.
4. The **server** verifies this certificate against its own `cacert.pem`. This step is what makes the authentication *mutual* — the server confirms the worker is a legitimate cluster member, not just any TLS client.
5. Both sides accept the connection and begin encrypted communication.

On the server side, the TLS context is configured in `server:start_server` with `peer_cert(true)` to require client certificates, and `cacerts([file(CaCert)])` to set the trust anchor. On the client/worker side, `client:rpc_execute` configures the same CA and presents the local client certificate.

An additional layer of security is provided by **HTTP digest authentication** (`passwordfile`), so even a node with a valid certificate must also know the correct username and password. Set

the plaintext once in `Source/Config/Private/passwords.pl`, then derive the hashed server file (it is not committed):

```
cp Source/Config/Private/template_passwords.pl \
   Source/Config/Private/passwords.pl
# edit config:digest_password/2
make passwordfile
```

Ship the same `passwords.pl` to every client and worker.

18.7.4 How certificates are resolved at runtime

Certificate paths are computed at runtime by `config:certificate/2` and `config:certificate/3`. Given a certificate name like `server-cert.pem`, the predicate prepends the installation directory and `Certificates/` to form the full path. For host-specific certificates, the hostname is prepended to the filename (e.g. `mac-pro.local.server-cert.pem`).

When TLS files are missing, both `server:require_tls_files/4` and `client:require_tls_files/4` print an error message listing the expected file paths and the `make certs` command needed to generate them.

18.8 Generating certificates

To generate a full set of certificates for a host:

```
make certs HOST="$(hostname)"
```

If your hostname includes a `.local` suffix (common on macOS), pass the full name so it matches `config:hostname/1`:

```
make certs HOST="mac-pro.local"
```

This runs `Certificates/Scripts/generate.sh`, which creates three things:

1. A **self-signed CA** (`cacert.pem` + `cakey.pem`) — created only if it does not already exist, so adding a second host reuses the same CA.
2. A **server certificate and key** signed by that CA, with the hostname embedded as the Common Name (CN).
3. A **client certificate and key** signed by the same CA.

18.8.1 Checking and renewing

Certificates are valid for 10 years, but the generation script also supports health checks and renewal:

```
make certs-check                # show expiry status for all hosts
make certs-renew                # renew certs expiring within 30 days
```

The `--check` subcommand prints each certificate's expiry date and flags any that are missing or expiring soon. The `--renew` subcommand regenerates only the certificates that need it, reusing the existing CA and private keys.

18.9 Encrypted two-node cluster: step-by-step

The following walkthrough shows how to set up a minimal cluster with one server and one worker on a local network.

Step 1 — Generate certificates on each machine.

On the server host (e.g. `server.local`):

```
make certs HOST="server.local"
```

On the worker host (e.g. `worker.local`):

```
make certs HOST="worker.local"
```

Each machine now has its own certificate and key, signed by a locally created CA.

Step 2 — Share the trust root.

By default, each machine creates its own CA. For mutual authentication to work, all nodes must trust the same CA. Copy `cacert.pem` from the server to the worker (or designate one machine as the cluster CA and distribute its `cacert.pem` to all nodes). Each node keeps its own host-specific certificate and key.

Step 3 — Start the server.

```
portage-ng --mode server
```

The server loads the knowledge base and listens on `config:server_bind/1`:`config:server_port/1` (default **localhost:4000**). Mutating POSTs (`/sync`, `/clear`, `/load`, …) therefore stay off the public network. Widen the bind (`config:server_bind(*)`) only on a trusted VPN/LAN — with a shared cluster CA, mTLS alone is not a strong gate; digest + bind scope are.

Step 4 — Start the worker.

```
portage-ng --mode worker --host server.local
```

Prefer an explicit `--host` (or a matching `config:server_host/1` pin). Bonjour discovery is allowed only as a convenience lookup that still must match that pin — the worker never connects to the first untrusted advertisement on a hostile LAN.

Before tree sync, fetch the portage git objects from your **trusted remote** (not from the Pengine server). The server advertises a full commit SHA; the worker checks it out only when that object already exists locally (`git cat-file -e`). There is no fetch-from-server path.

Step 5 — Discovery (optional).

If mDNS/Bonjour is available, the worker may resolve `_portage-ng._tcp` advertisements, but only a host that matches `config:server_host/1` is accepted.

Step 6 — Mutual TLS handshake.

When the worker connects, both sides present certificates signed by the shared CA. portage-ng verifies the Common Name and role, so only nodes with valid credentials can join the cluster. Digest auth is still required on top of mTLS.

Step 7 — Proving.

The worker polls the server's job queue, runs the full proving pipeline for each target, and posts results back. The server collects completed proofs and makes them available to clients.

18.10 Cluster usage

To run a distributed cluster, every node needs two things:

- A copy of the same `cacert.pem` (the shared trust root).
- Its own host-specific certificate and key pair.

The server is started with `--mode server`, and each worker with `--mode worker --host <pinned-server>`. Pin `config:server_host/1` on workers; Bonjour is only a lookup aid against that pin. TLS ensures peers share the same CA, and digest auth supplies the shared secret. You can add more workers at any time — they connect to the pinned server and start picking up jobs immediately.

18.11 Further reading

- Chapter 2: Installation and Quick Start — dns-sd and openssl prerequisites
- Chapter 15: Command-Line Interface — `--mode` flags
- Chapter 4: Architecture Overview — module load order for different modes

Chapter 19

Upstream and Bug Tracking

portage-ng integrates with external services to check upstream versions and search for known issues, helping users identify outdated packages and known dependency bugs.

19.1 Git repository integration

portage-ng can connect directly to a Git repository and turn Git metadata into Prolog facts. This means it can inspect commit history, changelogs, and file-level changes for any ebuild without relying on separate tools. The Git metadata is ingested alongside the regular cache data, so queries like "when was this ebuild last updated?" or "which ebuilds changed in the last sync?" can be answered from within the resolver.

19.2 Upstream version checking

The upstream module (`Source/Domain/Gentoo/upstream.pl`) checks package versions against upstream releases via the Repology API.

19.2.1 Usage

```
portage-ng --upstream sys-apps/portage
portage-ng --upstream @world
```

19.2.2 How it works

1. For each target package, the module queries the Repology API (`<config:repology_url>/api/v1/project/<name>`, default `https://repology.org`) for version information. A transport failure is reported as unreachable (and aborts the remaining checks) rather than "not found". While `repology.org` DNS is on registrar hold, set `config:repology_address/1` to the IPv4/IPv6 published in https://github.com/repology/repology-rs/issues/560.
2. The response includes version data across multiple distributions, which is compared against the version in the local Portage tree.
3. Results are categorized:
   - **Up to date** — local version matches or exceeds upstream
   - **Outdated** — a newer upstream version exists
   - **Unknown** — package not tracked by Repology

19.2.3 Output

The upstream check displays a comparison table showing the local version, the latest upstream version, and the status for each package.

19.3 Gentoo Bugzilla integration

The bugs module (`Source/Domain/Gentoo/bugs.pl`) plays two roles: it is the backend of the `bugzilla` repository type (a local, synced copy of the public bug tracker), and it answers `--search-bugs` queries.

19.3.1 The `bugzilla` repository type

Like `eapi` (a Portage tree), `vdb` (installed packages) or `binpkg`, a repository instance can be declared with type `bugzilla`. Its remote is a Bugzilla instance, its location holds the raw pages fetched from the REST API plus a resume state file, and its cache slot names the qcompiled store the rest of portage-ng reads:

```
:- bugzilla:newinstance(repository).
:- bugzilla:init('/var/cache/bugzilla',             % pages/ + state.pl
                 '/root/prolog/Knowledge/bugs.qlf',  % the bug store
                 'https://bugs.gentoo.org','rest','bugzilla').
:- kb:register(bugzilla).

config:repository_sync_limit(bugzilla, 1).           % network syncs per day
```

`--sync` (or `--sync bugzilla`) then runs the usual three phases:

1. `sync(repository)` — the network step. The first run walks the whole public bug space by bug-id keyset (`f1=bug_id&o1=greaterthan`, ordered by id, `config:bugzilla_page_size/1` bugs per page, a `config:bugzilla_request_delay/1` pause between pages). Every page is written to `<location>/pages/` and the state file advances after each one, so an interrupted crawl resumes where it stopped. Once complete, later runs only fetch bugs whose `last_change_time` moved since the previous run (with a ten-minute overlap).
2. `sync(metadata)` — a no-op; the pages are the metadata.
3. `sync(kb)` — folds the pending pages onto the store (a bug delivered again replaces its earlier row), writes `Knowledge/bugs.raw`, qcompiles it to `Knowledge/bugs.qlf` and deletes the consumed pages. With `config:bugzilla_scope(open)` closed bugs are dropped at this point; the default `all` keeps every public bug (about 600k rows, ~100 MB qlf).

The store is separate from `kb.qlf` and loaded lazily (`bugs:ensure_loaded/0`) by its consumers. Two fact families live in module `bugsdata`:

- `bug(Id, Product, Component, Status, Resolution, Severity, Priority, Assignee, Created, Changed, Keywords, Summary)`
- `bug_atom(Id, Category, Name, Version)` — every `category/name[-version]` atom found in the summary or in `cf_stabilisation_atoms`, with Version a `version/7` term or `version_none`. Categories are validated against the loaded tree so prose such as `usr/bin` is not indexed.

19.3.2 Daily sync cap

`config:repository_sync_limit(Repository, PerDay)` bounds the number of network syncs of any registered repository within a rolling 24-hour window; stamps are kept in `Knowledge/<Repository>.sync`. When the cap is reached the network step is skipped with a notice and the metadata / kb steps still rebuild from local data. Repositories without a declaration are unlimited. The repository is opt-in per host (it is not registered in `Source/Config/default.pl`); hosts that register it declare `bugzilla, 1`, which is the intended way to stay well within Gentoo's bot policy while keeping the local store fresh.

19.3.3 Searching (`--search-bugs`)

```
portage-ng --search-bugs dev-lang/rust
portage-ng --search-bugs "openssl segfault"
```

`config:bugzilla_search/1` selects the policy:

- `cache_first` (default) — a `category/name` term is answered from the atom index, anything else by a case-insensitive summary match, both against the local store (the output names the store's sync time). The live REST quicksearch is only used when nothing matches locally or no store has been synced.
- `rest` — always query the Bugzilla REST API directly.

19.3.4 Consumers of the store

- `--graph` renders a **bugs** page per ebuild (`<entry>-bugs.html`, `Source/Application/Output/Grapher/tracker.pl`): every bug naming the package, newest first, open bugs and bugs naming the page's exact version highlighted, each folding open to its stored columns. Facet chips, folded away behind the toolbar's **Filters** button (its badge counts the groups that differ from the defaults), filter the list client-side: status (open / resolved) and, for resolved bugs, resolution; version relevance (names this version / names another version / no version); component; severity; keyword (lit chips *require* one of them); last-changed age (any / 30 / 90 / 365 days); plus a free-text box over id, summary, assignee, component and keywords. Chips are OR-ed within a group and AND-ed across groups. Defaults show open bugs only and hide the `Stabilization` and `Keywording` components, which are workflow tickets rather than defects. The filter state is kept in the URL hash (`…-bugs.html#status=open,resolved&severity=major,critical&q=clang`) so a filtered view can be linked (opening such a link unfolds the chips), and a `#bug-<id>` fragment lights whatever chips are needed to show that bug and folds it open. The navigation bar's `bugs` pill counts the open bugs naming the package. See Chapter 14.
- The plan printer lists up to three known open bugs (exact-version matches first) under every domain assumption that names a package, gated by `config:bugzilla_annotate/1`. This is informational only: it changes neither the assumption nor the exit code.

Without a synced store all consumers stay silent, so hosts that never register a `bugzilla` repository see no change.

19.4 Automatic bug report drafts

The issue module (`Source/Domain/Gentoo/issue.pl`) generates structured Gentoo Bugzilla bug report drafts when the prover detects unsatisfiable dependencies.

A generated report includes:

- **Summary** — one-line description of the issue
- **Affected package** — the package atom
- **Unsatisfiable constraints** — the specific dependency that cannot be met
- **Observed state** — what the prover found (missing package, version conflict, REQUIRED_USE violation)
- **Suggested fix** — recommended action (add keyword, unmask, fix dependency)

These drafts can be used as starting points for filing bugs with the Gentoo bug tracker.

19.5 Bug report drafts from build-time discoveries

The prover-driven drafts above are generated at plan time from unsatisfiable dependencies. A second source of drafts comes from the **missing-provider feedback loop** (portage-ng#102): when a build fails because of an *undeclared* build dependency — a command, header, library, or pkg-config module the ebuild needed but never listed in `BDEPEND` — the builder records the discovery and re-derives a plan that supplies it (see Chapter 16: Missing provider feedback).

Because every discovery carries structured evidence — the missing symbol, the phase it surfaced in, the exit code, and the offending log line — the printer proposes a bug report draft at the end of the build for each dependency worked around this session:

```
>>> Missing build dependencies discovered (bug report drafts)

---
Summary: sec-policy/selinux-base: missing BDEPEND=sys-apps/semodule-utils (command
semodule_package not found)

Affected package: portage://sec-policy/selinux-base
Missing dependency: sys-apps/semodule-utils (build-time / BDEPEND)
Observed:
  command semodule_package not found during the compile phase (exit 127):
    semodule_package: command not found
Potential fix (suggestion):
  Add BDEPEND="sys-apps/semodule-utils" to the ebuild or the responsible inherited
eclass.
  (discovered by portage-ng missing-provider feedback, portage-ng#102)
```

Unlike the prover-driven drafts (which report a dependency that *cannot* be satisfied), these report a dependency that *was* satisfied once portage-ng learned it — so the draft is a ready-to-file "add `BDEPEND=<provider>`" fix against the ebuild or its inherited eclass. Both kinds of draft are gated by `config:bugreport_drafts_enabled/1`.

19.6 Further reading

- Chapter 15: Command-Line Interface — `--upstream` and `--bugs` flags
- Chapter 9: Assumptions and Constraint Learning — how unsatisfiable dependencies are detected
- Chapter 16: Building and Execution — the missing-provider feedback loop that produces build-time bug drafts
- Chapter 20: Gentoo Linux Security Advisories (GLSA) — security advisories and `@security` remediation sets

Chapter 20

Gentoo Linux Security Advisories (GLSA)

Gentoo publishes **GLSAs** — XML advisories that describe which package versions are vulnerable and which versions are safe. Traditional Portage exposes them mainly through `glsa-check` and the `@security` package set. portage-ng treats the same advisories as a **first-class knowledge artifact**: parsed into Prolog facts, optionally qcompiled for fast reload, queryable from the shell, and expanded into ordinary remediation atoms that the existing prove/plan pipeline consumes unchanged.

20.1 Design choice: knowledge, not a repository

A natural first idea is to register GLSAs as another `repository://entry` and search them with `query:search`. That fights the architecture.

Package repositories (`portage`, `pkg`, `binpkg`) identify entries by **CPVN** — category, package name, and `version/7` — via `cache:ordered_entry/5`. Every prover, orderer, and rules consumer assumes that shape. A GLSA id such as `202501-03` is not a package version; inventing a fake `glsa://…` CPVN would either pollute those paths or require permanent exclusion filters.

Non-package knowledge already lives outside the package-repo model:

Concern	Pattern
Profile	Own facts + `Knowledge/profile.qlf`
News	Filesystem read (display-only)
Named / computed sets	`preference:set/2` / `sets:expand/2`

GLSAs follow the profile pattern: a sibling knowledge store with its own facts and cache file, plus a small query surface. Package repos stay the only `Repo://Entry` units. Security sets are a *view* over that store, not the primary API.

```
        ├── glsa:search/2          (advisory queries)
        ├── query:search bridges   (vulnerable/1, glsa/1 on pkg entries)
        └── sets:expand/2          (@security → =cat/pkg-ver atoms)
                                        │
                                        ▼
                                prove_plan_with_fallback
```

20.2 On-disk source and cache

Advisories live in the Portage tree at `$PORTDIR/metadata/glsa/` as files named `glsa-YYYYMM-NN.xml`. Override the directory with `config:glsa_dir/1` when needed.

During `--sync`, portage-ng calls `glsa:cache_save/0` next to `profile:cache_save/0`. That walk:

1. Parses every `glsa-*.xml` (DTD-safe string extraction — no network DTD fetch, same rule as `metadata.xml` maintainers).
2. Writes `Knowledge/glsa.raw` and qcompiles `Knowledge/glsa.qlf`.
3. Loads the facts into the running process.

At runtime, `glsa:ensure_loaded/0` prefers the qlf cache and falls back to a live parse of `metadata/glsa/` when the cache is missing. Cold live parse of a full tree (~3.8k advisories) is well under a second; qlf reload is near-instant.

Parsing skips non-`ebuild` product types and tolerates individual malformed files so one bad advisory never aborts `@security` or sync.

20.3 Fact schema

The hot store is three dynamic predicates (also serialized into the `glsadata` module inside `glsa.qlf`):

```
glsa:advisory(Id, Title).
glsa:package(Id, Category, Name, ArchSpec).
glsa:range(Id, Category, Name, Kind, Op, Version, Slot).
```

Field	Meaning
`Id`	Advisory id (`'202501-03'`)
`Kind`	`vulnerable` or `unaffected`
`Op`	GLSA range token: `le`, `lt`, `eq`, `gt`, `ge`, `rge`, `rle`, `rgt`, `rlt`
`Version`	Bound as a `version/7` term
`Slot`	Slot atom, or `*` for any
`ArchSpec`	`*` or a space-separated arch list

Synopsis and body text are intentionally omitted from the hot store; set expansion and vulnerability checks only need package/range rows. Consumers that want the prose read one

advisory's XML on demand through `glsa:detail/2` (see Advisory detail below), so the cache path never grows with it.

20.4 Matching installed packages

Vulnerability is decided by joining advisory ranges against the **VDB** (installed packages) and the **tree** (visible upgrades):

1. **ARCH** — `*` always matches; otherwise the host arch from `userconfig:current_arch/1` (or `ARCH` / `ACCEPT_KEYWORDS`) must appear in the package's arch list. Unknown arch ⇒ only `*` matches (conservative).
2. **Vulnerable range** — installed version matches a `vulnerable` range for that C/N/slot.
3. **Not unaffected** — the same installed version must *not* match an `unaffected` range.
4. **Upgrade exists** — a visible tree ebuild in the same slot matches an `unaffected` range and is greater than the installed version. Among such upgrades, portage-ng picks the **least-change** (lowest) version, matching Portage's `getMergeList(least_change=true)`.

Ordinary comparisons (`le`/`lt`/`eq`/`gt`/`ge`) use `eapi:version_compare/3` on `version/7` terms. Revision-limited ops (`rge`/`rle`/`rgt`/`rlt`) require the same base version and compare only the revision field — Portage's `revisionMatch` semantics.

Remediation atoms are exact pins: `=category/name-version`. Those atoms enter the normal target rules; the prover is not GLSA-aware.

20.5 Security computed sets

Portage's `sets.conf` defaults `@security` to `NewAffectedSet`. portage-ng registers four computed set names in `Source/Domain/Gentoo/Preference/sets.pl`:

Set	Portage class	Meaning
`@security`	`NewAffectedSet` (default)	Vulnerable installs from GLSAs not yet applied
`@new-affected`	`NewAffectedSet`	Same, explicit name
`@affected`	`AffectedSet`	Vulnerable installs, including applied GLSAs
`@new-glsa`	`NewGlsaSet`	Unapplied GLSAs (atoms still only appear when an upgrade exists)

Related non-GLSA computed sets live in the same registry (`Preference/sets.pl`); see Chapter 15: CLI — Package sets for the full table (`@preserved-rebuild`, `@changed-deps`, `@installed`, …).

Expansion is **VDB-driven**: walk installed packages, look up matching advisory rows, emit upgrade atoms, then reduce per `cat/name:slot` to the highest remediation version (Portage `_reduce`). This stays near- linear in installed CPV count rather than scanning every advisory for `is_vulnerable`.

```
portage-ng --mode standalone --pretend @security
portage-ng --mode standalone --list-sets   # includes security sets
```

When no installed package is vulnerable (or every matching GLSA is already injected), `@security` expands to the empty list and the CLI reports that the set is empty — exit 0, not a hard failure.

20.6 Applied / injected tracking

Portage records applied GLSA ids in `$EROOT/var/lib/portage/glsa_injected`. portage-ng mirrors that with `config:glsa_injected_file/1` (default: `Source/Knowledge/Sets/glsa_injected/<hostname>`).

Predicate	Role
`glsa:applied(+Id)`	True when `Id` is listed in the inject file
`glsa:inject(+Id)`	Append `Id` if not already present

`NewAffectedSet` / `NewGlsaSet` filters consult this file. A dedicated `glsa-check`-style inject CLI is not required for set expansion; the predicates are ready for a thin follow-up action.

20.7 Query surface

20.7.1 Advisory search — `glsa:search/2`

```
glsa:search([package('dev-python', pip), applied(false), vulnerable(true)], Id).
glsa:search(title(Title), Id).
```

Accepted constraints: `id/1`, `title/1`, `package/2`, `applied/1`, `vulnerable/1`. Queries run against `glsa:*` facts (and VDB joins for `vulnerable`), not against `cache:ordered_entry`.

20.7.2 Package bridges — `query:search`

Two compile-time sugar keys join advisories onto an existing `Repo://Entry` (typically the VDB):

Query	Meaning
`vulnerable(true)`	Some non-filtered GLSA covers this installed entry and an upgrade exists
`glsa(Id)`	Advisory `Id` covers this entry's C/N/version/slot

```
?- knowledgebase:vdb_repository(V),
   query:search([vulnerable(true), category(C), name(N)], V://E).
```

These bridges do **not** invent `glsa://` entries. Prefer `glsa:search/2` inside set logic so the package query hot path stays free of advisory scans unless asked.

20.7.3 Package-centric views

Two helpers serve callers that start from a package rather than from the VDB (the `--graph` security page is the main one):

Predicate	Meaning
`glsa:package_advisories(+C, +N, -Ids)`	Advisory ids naming `C/N`, newest first
`glsa:entry_status(+Id, +Repo://Entry, -Status)`	`vulnerable`, `unaffected` or `unlisted` for one tree entry (ARCH ignored)

`vulnerable` uses the same test as `glsa:entry_covered/2` (in a vulnerable range, not in an unaffected one); `unlisted` means the advisory names the package but neither range mentions this version/slot.

20.8 Advisory detail

`glsa:detail(+Id, -Detail)` returns the full advisory text by reading `glsa-<Id>.xml` from the GLSA source directory — the same DTD-safe string extraction as the parser, no `load_structure/3`. It fails on hosts that only carry `Knowledge/glsa.qlf` without the XML tree, so callers must degrade gracefully (the graph page falls back to the cached title and ranges plus a link to security.gentoo.org).

`Detail` is a list of field terms, each present at most once:

Field	Content
`synopsis(Text)`	One-line summary
`announced(Date)`, `revised(Date, Count)`	Publication dates; `Count` is the `<revised count="…">` value
`access(Text)`, `severity(Level)`	`<access>` text and the `<impact type="…">` level (`high`, `normal`, `low`, …)
`bugs([Bug, …])`	Gentoo bug numbers
`background(Blocks)`, `description(Blocks)`, `impact(Blocks)`, `workaround(Blocks)`, `resolution(Blocks)`	Prose sections
`references([ref(Label, Url), …])`	CVE identifiers and other URIs

`Blocks` is an ordered list of `p(Text)`, `code(Text)` (dedented, line structure kept) and `list([Item, …])` terms. Inline markup (`<i>`, `<b>`, `<uri>`) is stripped and XML entities are decoded, so every `Text` is plain text ready for re-escaping by the consumer. `glsa:advisory_url/2` and `glsa:bug_url/2` build the canonical security.gentoo.org and bugs.gentoo.org links.

20.9 Graph pages

`--graph` writes a `<category>/<name>-<version>-glsa.html` page next to the other per-ebuild pages (type `glsa` in `config:graph_html_type/1`, rendered by `Source/Application/Output/Grapher/security.pl`). Every ebuild page carries a **security** group in its tab bar whose `glsa` link shows the number of advisories naming the package; the pill turns red when one of them places the page's version in a vulnerable range.

The page lists those advisories newest first. Each row shows the GLSA id, title, a status badge for the page's version (*affects this version* / *this version unaffected* / *version not in range*), an *installed copy vulnerable* badge when the VDB copy is in a vulnerable range, the severity and access badges, and the announcement date. A row folds open (native `<details>`) to the synopsis, metadata (announced, revised, access, bug links, advisory link), the affected-package table with vulnerable and unaffected ranges for every package the advisory names, the prose sections and the reference links. The toolbar filters to advisories affecting this version and expands or collapses all rows; a `#glsa-<id>` URL fragment opens that advisory directly.

Because the page is rendered at `--graph` time, it reflects the GLSA tree and VDB of the generating host, exactly like the installed markers on the deptree page.

20.10 Module map

File	Role
`Source/Domain/Gentoo/glsa.pl`	Parse, facts, cache, match, search, set atoms, on-demand `detail/2`
`Source/Domain/Gentoo/Preference/sets.pl`	Registers `@security` and siblings
`Source/Knowledge/query.pl`	`vulnerable/1` and `glsa/1` bridges
`Source/Application/Output/Grapher/security.pl`	`--graph` per-ebuild GLSA page (`-glsa.html`)
`Source/Application/Output/Grapher/navtheme.pl`	`security` tab group with the advisory count pill
`Source/Application/Interface/Action/sync.pl`	Calls `glsa:cache_save` during `--sync`; lists computed sets
`Source/config.pl`	`config:glsa_dir/1`, `config:glsa_injected_file/1`

Loaded with the domain modules (`loader:group(domain_modules)`), after preference and before `sets.pl`.

20.11 What this is not

- **Not a package repository.** No `kb:register(glsa)`, no GLSA rows in `kb.qlf`.
- **Not prover logic.** No new rules, assumptions, or fallback tiers. Remediations are ordinary `=cpv` targets.

- **Not a full `glsa-check` clone (yet).** List/mail/fix modes can wrap the same facts later; set expansion, search and the graph page are the current surface.
- **Not network-fetched.** Advisories arrive with the tree after the user runs `--sync`.

20.12 Further reading

- Chapter 6: Knowledge Base and Cache — package `kb.qlf` vs sibling caches such as `profile.qlf` / `glsa.qlf`
- Chapter 3: Configuration — sync, profile cache, and host-local paths
- Chapter 15: Command-Line Interface — `--pretend`, `--list-sets`, target `@set` syntax
- Chapter 10: Version Domains — `version/7` comparison used by GLSA range matching
- Portage reference: `lib/portage/glsa.py`, `lib/portage/_sets/security.py`

Chapter 21

Contextual Logic Programming

context is an object-oriented programming paradigm for Prolog, implemented in `Source/Logic/context.pl`. It provides contexts (namespaces), classes, and instances with public, protected, and private access control, multiple inheritance, cloning, and declarative static typing of data members.

21.1 Motivation

Standard Prolog uses a flat global namespace. As applications grow, name collisions, uncontrolled access to dynamic predicates, and lack of modularity become obstacles. **context** addresses this by splitting the global namespace into isolated contexts, each with their own facts and rules.

The key insight is that contexts can be **unified** and can serve as feature terms describing software configurations — directly connecting to Zeller's *Unified Versioning through Feature Logic*. This makes **context** both a software engineering tool and a formal foundation for reasoning about configurations.

In practical terms, this is how portage-ng can treat a Portage tree, an overlay, and a VDB as separate objects that share the same interface but carry independent state — and how dependency contexts can be merged, intersected, and propagated through the proof tree.

21.2 How it differs from Logtalk

The syntax is comparable to Logtalk, but the approach is fundamentally different:

	Logtalk	**context**
Approach	Compile-time translation to plain Prolog	Runtime generation of guarded predicates
Overhead	Source-to-source compilation step	No compilation; contexts created dynamically
Thread safety	Varies by backend	Built-in; tokens are thread-local
Feature unification	Not supported	Contexts unify as feature terms

Because **context** works at runtime, contexts can be created, cloned, and composed dynamically — which portage-ng uses extensively to represent repositories, ebuilds, and configurations as live objects.

21.3 Core concepts

21.3.1 Contexts

A context groups together clauses of a Prolog application. By default, clauses are local to their context and invisible to other contexts unless explicitly exported. Referencing a context is enough to create it (creation ex nihilo).

Context rules are evaluated in the context in which they are defined. An exception are clauses declared as "transparent", which inherit their context from the predicate that is calling them — the same mechanism SWI-Prolog uses for meta-predicates, but applied at the context level.

21.3.2 Classes

A class is a special context that declares public, protected, and private meta-predicates. These declarations control access at three levels:

- **Instantiation** — which predicates are copied into the instance and how they are guarded.
- **Inheritance** — which predicates are visible to subclasses. Private predicates are not inherited; public and protected ones are.
- **Invocation** — which predicates external callers may use. Only public predicates can be called from outside; protected and private predicates throw a `permission_error` if accessed without a valid access token.

A class is declared with `:- class.` (or `:- class([Parent1, Parent2])` for multiple inheritance). Predicate visibility is declared with `:- dpublic(name/1).`, `:- dprotected(age/1).`, and `:- dprivate(secret/1).`

21.3.3 Instances

Instances are dynamically created from a class via `newinstance/1`. The creation process has four stages:

1. **Metadata registration** — the instance is marked as `type(instance(Parent))` in its `$__meta` store.
2. **Predicate inheritance** — all declared predicates from the parent class are copied. Each predicate is wrapped in a **guard** that checks an access token before execution.
3. **Freeze** — the generated predicates are compiled via `compile_predicates/1` for optimal performance (no more interpretation overhead).
4. **Constructor call** — if the class declares a constructor predicate (same name as the class), it is called with the arguments passed to `newinstance/1`.

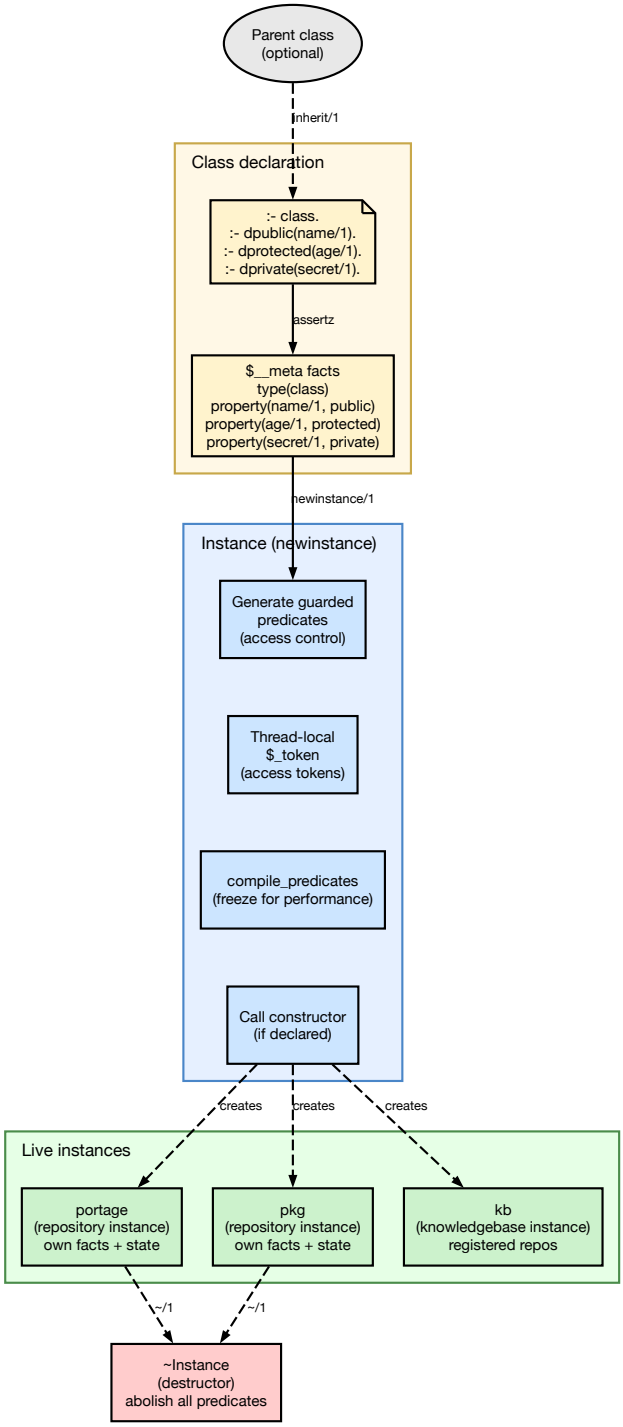


Figure 55 — Class and instance lifecycle

Each instance is a fully independent Prolog module with its own dynamic facts. Multiple instances of the same class coexist without interference — for example, `portage` and `overlay` are both instances of the `repository` class, but each carries its own cached ebuilds, location paths, and sync state.

21.3.4 Destruction

An instance is destroyed by calling `~Instance`. The destructor (if declared) runs first, then all dynamic predicates in the instance module are abolished. Static contexts (built-in Prolog modules) cannot be destroyed — attempting to do so raises a `permission_error`.

21.3.5 Access control and thread safety

The access control mechanism uses **thread-local tokens**. When an external caller invokes a public predicate on an instance, the system asserts a `$_token(thread_access)` fact in the instance's module. The guarded implementations of protected and private predicates check for this token:

- **Public** — the token is asserted before the call and retracted afterwards (via `call_cleanup`). Any code called during the public predicate's execution can access protected and private predicates because the token exists.
- **Protected** — the guard checks that the token exists. If it does (i.e. the call originates from a public predicate on the same instance), execution proceeds. Otherwise, a `permission_error` is thrown.
- **Private** — same check as protected. The difference is conceptual: private predicates are not inherited by subclasses, while protected ones are.
- **Static** — the call is forwarded to the parent class directly, bypassing the instance.

Because the token is `thread_local`, concurrent threads can call public predicates on the same instance without interfering with each other's access grants.

21.4 Operators

context defines several operators for interacting with contexts. The diagram below shows how they relate to an instance's internal structure:

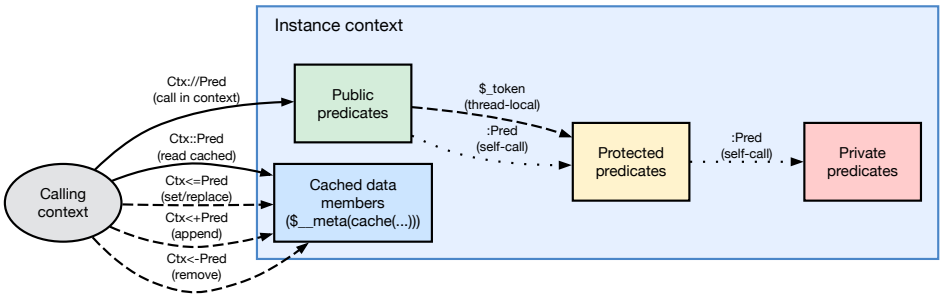


Figure 56 — Context operators

Operator	Meaning
`Head ::- Body`	Declare a method clause (the contextual `:-`)
`:Pred`	Call `Pred` in the current context (self-call)
`::Pred`	Read a cached data member
`<=Pred`	Set a data member (evaluate, retract old, assert new)
`<+Pred`	Add a data member (evaluate, assert if not exists)
`<-Pred`	Remove a data member
`Ctx://Pred`	Call `Pred` in a specific context

The `::-` operator is declared at priority 1200 `xfx` — exactly the priority of Prolog's `:-`. A clause `Head ::- Body` is an ordinary logical rule, but one whose visibility and state are governed by the instance it lives in: at instantiation it is copied into the instance and wrapped in an access guard (see *Instances* above). The operator doubles as the portage-ng logo; Chapter 1, section 1.4.6 tells that story.

21.4.1 Data members

The `::`, `<=`, `<+`, and `<-` operators implement data-member-like behaviour. Under the hood, they work with `$__meta(cache(...))` facts:

- **`Ctx::Pred`** — looks up `cache(Pred)` in the instance's metadata. If found, calls the predicate (which succeeds immediately because the cached value has already been evaluated). This is a **read** operation.
- **`Ctx<=Pred`** — evaluates `Pred` in the instance, retracts any existing cache for the same functor/arity, and asserts the new result. This is a **write-replace** operation.
- **`Ctx<+Pred`** — evaluates `Pred` and asserts the result as a new cache entry without removing existing ones. This is an **append** operation, useful for multi-valued data members.
- **`Ctx<-Pred`** — retracts all cache entries matching `Pred`. This is a **delete** operation.

21.4.2 The `://` operator

The `://` operator is the primary way to call a predicate in another context. It is defined simply as `'://'(Context, Predicate) :- Context:Predicate.` — a thin wrapper that resolves the context module and dispatches the call. This operator appears throughout portage-ng in forms like `portage://entry(E)` or `kb://query(Q, R)`.

21.5 Example: a Person class

A complete worked example of a context class — including a constructor, destructor, public/protected/private members, and instance creation — is presented in Chapter 1, section 1.4.6. The example defines a `person` class with name and age accessors, title management, and demonstrates how instances carry their own state independently.

21.6 How portage-ng uses context

portage-ng uses **context** throughout its architecture. The diagram below shows the two main classes (`repository` and `knowledgebase`), their instances, and how the prover accesses them.

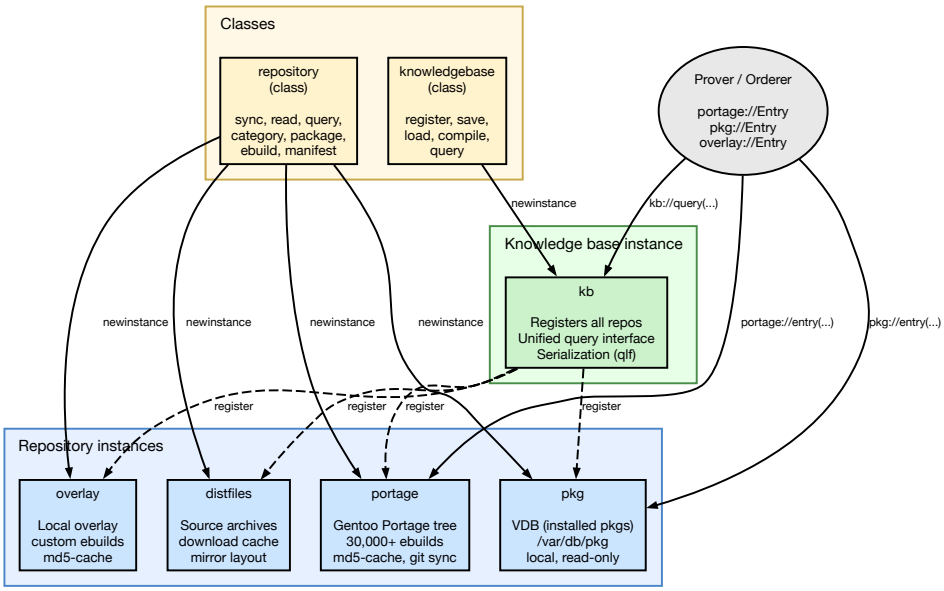


Figure 57 — portage-ng context usage

21.6.1 Repository instances

The `repository` class (`Source/Knowledge/repository.pl`) declares a rich public interface for working with package repositories: syncing from remote sources, reading metadata, querying entries, and generating graphs. Protected members hold configuration (location, cache path, remote URL, sync protocol).

Each repository on disk becomes a named instance:

- **`portage`** — the main Gentoo Portage tree (30,000+ ebuilds), synced via git, with md5-cache metadata.
- **`pkg`** — the VDB (installed packages) at `/var/db/pkg`, a local read-only repository.
- **`overlay`** — a local overlay with custom ebuilds.
- **`distfiles`** — the source archive download cache.

Instance creation and configuration happens in the host-specific config file (e.g. `Source/Config/mac-pro.local.pl`):

```
:- portage:newinstance(repository).
:- portage:init('/Volumes/Storage/Repository/portage-git',
                '/Volumes/Storage/Repository/portage-git/metadata/md5-cache',
                'https://github.com/gentoo/gentoo.git',
                'git', 'eapi').

:- pkg:newinstance(repository).
:- pkg:init('/Volumes/Storage/Repository/pkg', '', '', 'local', 'vdb').
```

Because all instances share the same class, the prover does not need to know whether a dependency comes from the main tree, an overlay, or the VDB — the same `Repo://entry(E)` call works for all of them.

21.6.2 Knowledge base instance

The `knowledgebase` class (`Source/Knowledge/knowledgebase.pl`) acts as a registry. Its `register/1` predicate adds repository instances, and its `query/2` predicate searches across all registered repositories. The instance `kb` is created at startup and populated by the configuration file.

In **client mode**, the knowledge base instance is created with a hostname and port: `kb:newinstance(knowledgebase(Host, Port))`. This switches the instance into proxy mode, where queries are forwarded to the remote server via Pengine RPC — but the calling code sees the same `kb://query(Q, R)` interface as in standalone mode.

21.6.3 Context terms in the prover

Beyond the OOP use, the context system's unification semantics flow into the prover. Every literal in the proof tree carries a context list `?{[...]}` that accumulates constraints as the proof deepens. These context terms are merged via feature unification — the same algebraic operation that the context system uses for its data members. This connection is explored in detail in Chapter 22: Context Terms and Feature Unification.

21.7 Serialization

Context instances support serialization through the knowledge base's `save` and `load` predicates. When `kb://save` is called, the cached facts from all registered repository instances are written to a QLF file (`Knowledge/kb.qlf`). On the next startup, `kb://load` reads the compiled file back, restoring all repository instances to their previous state without re-parsing the Portage tree from disk. This is what makes portage-ng's startup fast — the full knowledge base (30,000+ ebuilds with all metadata) loads from the QLF file in under a second.

21.8 Further reading

- A. Zeller, *Unified Versioning through Feature Logic*, 1997
- `Source/Logic/context.pl` — full implementation
- `Documentation/Handbook/22-doc-context-terms.md` — how context terms flow through the prover

Chapter 22

Context Terms in portage-ng

How contexts are created, propagated, and merged across the dependency graph.

22.1 Overview

Every literal in the prover carries a **context** — a list of tagged terms that records provenance, ordering, constraints, and USE requirements as the proof expands through dependencies. The literal format is:

```
Literal:Action?{Context}
```

Contexts are not opaque blobs; they are structured as **feature-term lists** and are merged using a Zeller-inspired feature-unification algorithm. This gives them lattice semantics: merging two contexts produces a well-defined meet that preserves all non-contradictory information from both sides.

22.2 Anatomy of a context

A context is a Prolog list. Each element is either a plain term or a `Feature:Value` pair. The distinction matters for merging:

Form	Example	Merge behaviour
Plain term	`self(R://E)`	Identity match; duplicates dropped
Feature:Value	`build_with_use:use_state(V1,D1)`	Value merged by `val_hook/3`
Feature:Compound	`slot(C,N,Ss):{...}`	Compound feature key

For example, `self(portage://sys-apps/portage-3.0.77-r3)` is a plain term that identifies the parent ebuild. `build_with_use:use_state([foo],[bar])` is a Feature:Value pair whose value is merged when two contexts meet. `slot(sys-apps,portage,0/0):{Candidate}` is a compound feature key used for sub-slot rebuild tracking.

22.2.1 Common context tags

The following tags appear most frequently in proof-term contexts. Each tag is a Prolog term added to the context list during rule evaluation.

`self(Repo://Entry)` — set by `dependency:add_self_to_dep_contexts`. Identifies the parent ebuild that introduced this dependency edge.

`build_with_use:use_state(En,Dis)` — set by `dependency:process_build_with_use`. Carries bracketed USE constraints from the dependency atom (e.g. `dev-libs/foo[bar,-baz]`).

`slot(C,N,Ss):{Candidate}` — set by `dependency:process_slot`. Records a slot lock from `:=` (subslot rebuild) semantics.

`after(Literal)` — seeded at world-set anchors (`world(Arg):register?{[after(Repo://Ebuild:run)]}`) and propagated into dependency contexts by `featureterm:add_after_to_dep_contexts`. Ordering hint: this subtree should come after `Literal` in the plan. Propagates to children.

`after_only(Literal)` — injected on PDEPEND edges (via `featureterm:add_after_only_to_dep_contexts`). Becomes an `order_after` preference (soft requirement) honored by the pass-2 orderer; does **not** propagate to children.

`replaces(pkg://Entry)` — set by install/update rules. Records which installed package this action replaces.

`assumption_reason(Reason)` — set by the domain assumption fallback. Records why a domain assumption was made (e.g. `missing`, `masked`, `keyword_filtered`).

`suggestion(Type[, Detail...])` — set by the relaxation fallback. Records an actionable suggestion; arity varies with the suggestion type (e.g. `suggestion(unmask)`, `suggestion(accept_keyword, Kw)`, `suggestion(use_change, Ebuild, Changes)`).

`domain_reason(cn_domain(C,N,Tags))` — set by `cnselect:add_domain_reason_context/5`. Diagnostic tags for version domain narrowing.

`constraint(cn_domain(C,N,Slot):{Domain})` — set by the constraint system. Carries an inline per-slot constraint for domain scoping.

22.3 Context lifecycle

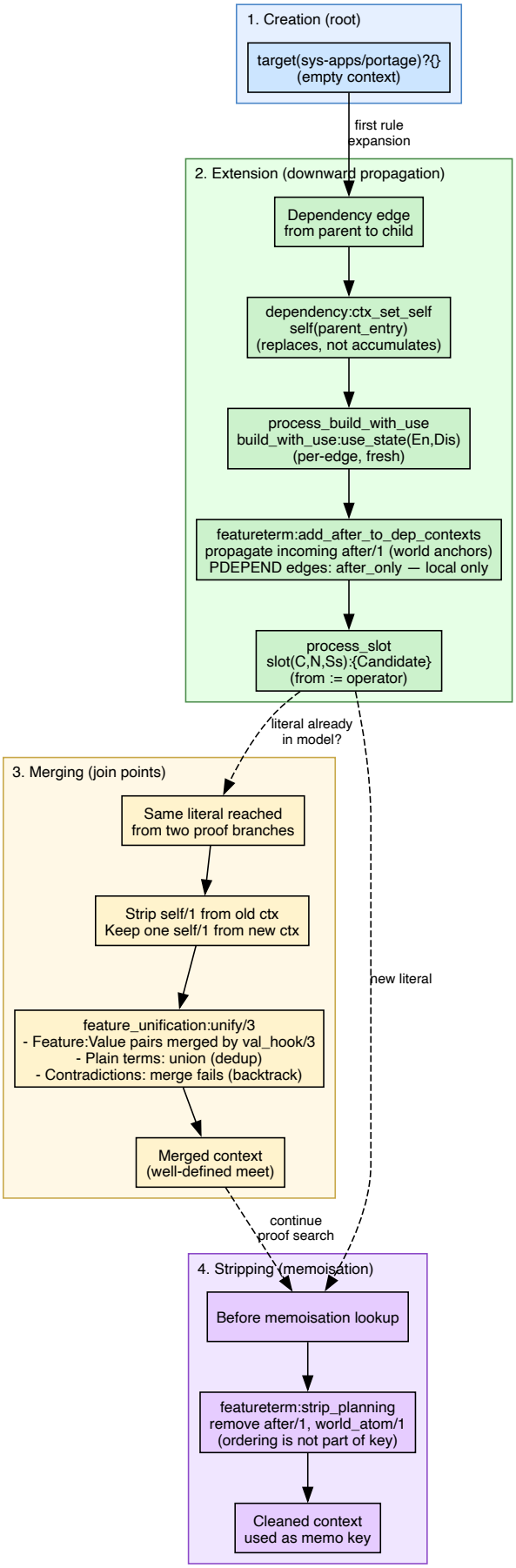


Figure 58 — Context lifecycle

22.3.1 1. Creation (root)

At the top level, the prover starts with an empty context (`{}` or `[]`). The first rule expansion — typically `target/2` → `install` — begins populating it.

22.3.2 2. Extension (downward propagation)

As rules expand dependencies, contexts grow:

```
target(sys-apps/portage)?{}
  └─ install(portage://sys-apps/portage-3.0.77-r3):install?{...}
       ├─ dep(dev-lang/python):install?{self(portage://sys-apps/portage-3.0.77-r3),
       │                                build_with_use:use_state([ssl,threads],[])}
       │    └─ dep(dev-libs/openssl):install?{self(portage://dev-lang/python-3.13),
       │                                     build_with_use:use_state([],[])}
       └─ dep(app-arch/tar):install?{self(portage://sys-apps/portage-3.0.77-r3)}
```

(Had the goal been seeded from a world-set anchor — `world(Arg):register?{[after(Repo://Ebuild:run)]}` — an `after/1` marker would additionally flow down every dependency edge.)

Key propagation rules:

- **`self/1`** is set to the current ebuild at each dependency edge. It does **not** accumulate — each edge replaces the previous `self`.
- **`build_with_use`** is per-edge: the child gets a fresh `build_with_use` from its dep atom, not the parent's `build_with_use`.
- **`after/1`** propagates transitively (children inherit it).
- **`after_only/1`** does **not** propagate (ordering is local to this edge).
- **`assumption_reason`** and **`build_with_use`** are dropped on PDEPEND edges (via `featureterm:drop_build_with_use_and_assumption_reason/2`).

22.3.3 3. Merging (join points)

When the prover encounters a literal that was already proven with a different context, it merges the old and new contexts via feature term unification:

```
sampler:ctx_union(OldCtx, NewCtx, MergedCtx)
```

The merge algorithm:

1. **Strip** **`self/1`** from the old context entirely.
2. **Extract one** **`self/1`** from the new context (keep it aside).
3. **Unify** the remaining lists via `feature_unification:unify/3`.
4. **Prepend** the extracted `self/1` back onto the result.

This guarantees: - At most one `self/1` in the merged result (from the new/incoming side). - Feature:Value pairs with the same key are merged by `val_hook/3`. - Plain terms present in either side appear in the result (union semantics).

22.3.4 4. Stripping for memoisation

Before checking whether a literal has already been proven, planning markers are stripped so they don't pollute the memoisation key:

```
featureterm:strip_planning(Context0, Context)
```

This removes `after/1` and `world_atom/1` — ordering and planning concerns that should not affect whether a proof is reusable.

22.4 Feature unification in detail

`feature_unification:unify/3` implements a **horizontal unification** algorithm inspired by Zeller's feature logic:

1. Normalise both terms (`{}` → `[]`).
2. Walk both lists. For each `Feature:Value` pair in list A, check if list B has the same `Feature`.
3. If both sides have `Feature`, merge values via `val/3` (or `val_hook/3` for domain-specific merge).
4. If only one side has `Feature`, include it in the result.
5. Plain terms are matched by identity; duplicates are dropped.

22.4.1 Value merge rules

V1	V2	Result	Semantics
`{L1}`	`{L2}`	`{Intersection}`	Set intersection (must be non-empty)
`[L1]`	`[L2]`	`[Union]`	Sorted union (fails on contradictions)
atom `V`	`{L}`	`{V}` if `V ∈ L`	Singleton intersection
`V`	`V`	`V`	Identity

22.4.2 Domain-specific hooks (`val_hook/3`)

Feature	Hook in	Merge behaviour
`build_with_use`	`use.pl`	`use_state(En1,Dis1)` ⍰ `use_state(En2,Dis2)` = union of enable/disable sets; **fails** if a flag appears in both enable and disable
`cn_domain`	`version.pl`	`version_domain` meet (intersection of version bounds); `none` is identity

22.5 `self/1` — parent provenance

The `self/1` tag identifies **which ebuild introduced this dependency**. It is critical for:

- **USE evaluation**: `use:effective_use_in_context/3` looks up the USE model of the ebuild in `self/1` to evaluate USE conditionals.
- **Blocker source**: `candidate:make_blocker_constraint/5` uses `self/1` to determine who is blocking whom.
- **Parent narrowing**: `candidate:maybe_learn_parent_narrowing/4` uses `self/1` to learn that the parent version should be excluded when a child dependency cannot be satisfied.
- **REQUIRED_USE**: `query:with_required_use_validate/3` annotates REQUIRED_USE terms with `:validate?{[self(...)]}` so the prover knows the ebuild context.

22.5.1 Invariant: at most one `self/1`

Without bounding, `self/1` would stack along dependency chains:

```
[self(A), self(B), self(C), ...]  ← unbounded growth
```

The system prevents this at two levels:

1. **`dependency:ctx_set_self/3`** replaces any existing `self/1` when setting a new parent.
2. **Feature term unification** (`ctx_union_raw/3`) strips all `self/1` from the old context and keeps only one from the new context.

22.6 `build_with_use` — bracketed USE requirements

When a dependency atom carries USE requirements (e.g. `dev-lang/python[ssl,threads]`), they are recorded as:

```
build_with_use:use_state([ssl, threads], [])
```

The enable list contains flags that must be ON; the disable list contains flags that must be OFF.

22.6.1 Per-edge, not inherited

Each dependency edge computes its own `build_with_use` from the dep atom. The parent's `build_with_use` is **removed** before computing the child's:

```
dependency:process_build_with_use(MergedUse, ContextDep, NewContext, ...)
```

This prevents a grandparent's USE requirements from leaking to grandchildren.

22.6.2 Merge semantics

When feature term unification merges two contexts with `build_with_use`, the `val_hook` in `use.pl` takes the **union** of enable sets and the **union** of disable sets. If a flag appears in both enable and disable, the merge **fails** (contradiction), forcing the prover to backtrack.

22.6.3 Post dependencies

On PDEPEND edges, `build_with_use` is dropped because PDEPEND dependencies are resolved at runtime, not build time, so build-time USE constraints do not apply.

22.7 Constraints vs contexts

The proof system uses two complementary mechanisms for carrying information, and it is easy to confuse them. **Contexts** are local: each literal in the proof carries its own context list (`?{...}`), recording where it came from, what USE flags were requested, and how it should be ordered. **Constraints** are global: they live in a shared ConstraintsAVL that spans the entire proof and track cross-cutting invariants like "only one version of this package may be selected" or "this package is blocked by another."

The table below summarises the key differences:

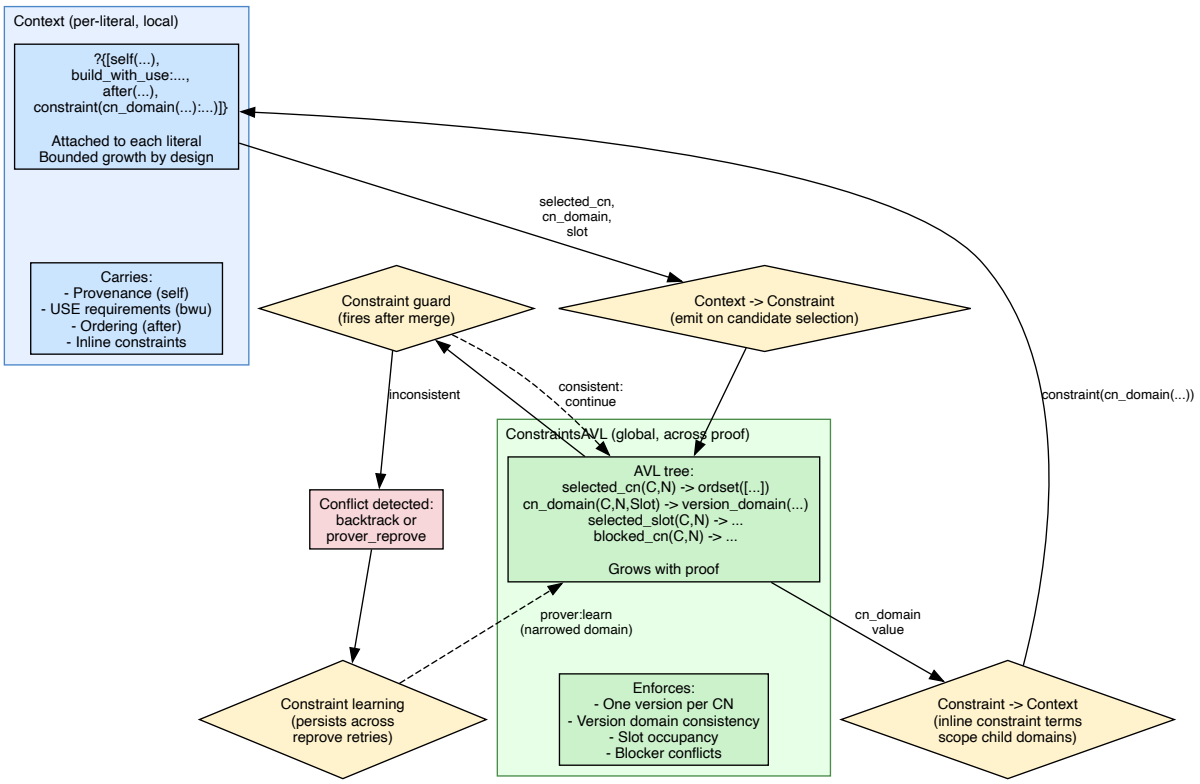


Figure 59 — Context vs constraint interaction

Aspect	Context	Constraint
Scope	Per-literal (local)	Global (across proof)
Storage	List attached to `?{...}`	AVL in ConstraintsAVL
Growth	Bounded by design	Grows with proof
Purpose	Provenance, ordering, USE	Version selection, slot locks, blockers

22.7.1 How they interact

Although contexts and constraints have different scopes, they are not isolated — information flows between them in both directions.

From context to constraint. When the rules layer selects a candidate version for a dependency, it emits constraint terms into the global ConstraintsAVL. For example, selecting `dev-libs/openssl-3.1.4` produces a `selected_cn(dev-libs, openssl)` constraint and a `cn_domain` constraint recording the version domain. These global constraints ensure that if another dependency path also needs `dev-libs/openssl`, the prover will detect any conflict.

From constraint to context. Sometimes a parent dependency wants to narrow the version domain for a child before candidate selection even begins. It does this by placing an inline constraint term like `constraint(cn_domain(C,N,Slot):{Domain})` directly in the context list. When the child's rule fires, it reads this term and applies the domain restriction.

Constraint guards. After each new constraint is merged into the global store, `heuristic:constraint_guard/2` fires to check consistency. The guard verifies that version domains are compatible with selected candidates, that each slot has at most one selected version, and that no selected package is blocked by another.

Constraint learning. When a guard detects an inconsistency, it can record a narrowed domain via `prover:learn/3`. This learned constraint persists across reprove retries, preventing the prover from repeating the same dead-end choice (see Chapter 9 and Chapter 10).

22.8 Ordering: `after` vs `after_only`

Both influence ordering in the plan, but they differ in origin and propagation:

Marker	Propagates to child deps?	Origin / use case
`after(Lit)`	Yes	World-set anchors: the package and all its deps should come after `Lit`
`after_only(Lit)`	No	PDEPEND completion: only this package (not its deps) prefers to come after `Lit`

In `after_only` mode the marker is rewritten to a `constraint(order_after(...):{[]})` term — an ordering-only **preference** (a soft requirement; it travels through the constraint store but is not a constraint on the model) that the pass-2 orderer (`prefers/2` in `Source/Domain/Gentoo/Rules/ordering.pl`) honors when it lies on no loop (the exact policy is the `--optimize` strategy, Chapter 13). Neither marker is minted per DEPEND/RDEPEND edge; build-time vs runtime ordering is decided in pass 2 by the ordering rule set (`requires/2` / `prefers/2`), not by context markers.

22.8.1 Extraction

```
featureterm:get_after_with_mode(Context, After, AfterForDeps, ContextRest)
```

- If `after(X)` → `After = X`, `AfterForDeps = X` (propagate).
- If `after_only(X)` → `After = X`, `AfterForDeps = none` (don't propagate).
- If neither → both `none`.

22.9 Example: full context evolution

The following example traces how context evolves as the prover walks from a user target (`sys-apps/portage`) through two levels of dependencies. The diagram shows the key context tags at each step.

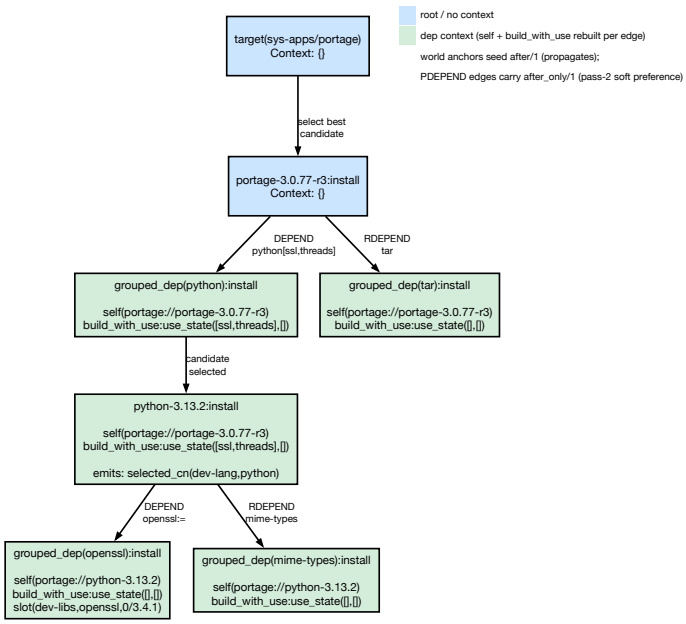


Figure 60 — Context evolution through a dependency chain

22.9.1 Step 1 — Target resolution

The user runs `emerge sys-apps/portage`. The target rule selects the best visible candidate (`portage-3.0.77-r3`). At this point the context is empty — there is no parent, no USE requirement, and no ordering marker.

22.9.2 Step 2 — Expanding portage's dependencies

The install rule for portage expands its DEPEND and RDEPEND. Each dependency atom gets its own context, built from three operations:

- **`add_self_to_dep_contexts`** adds `self(portage://portage-3.0.77-r3)` to record that portage is the parent.
- **`process_build_with_use`** translates bracketed USE flags from the atom. For `dev-lang/python[ssl,threads]`, this produces `build_with_use:use_state([ssl,threads],[])`. For `app-arch/tar` (no brackets), the USE state is empty.

- `featureterm:get_after_with_mode` + `featureterm:add_after_to_dep_contexts` propagate any incoming `after/1` marker (e.g. from an `@world` anchor) into the dep contexts. No new markers are minted here — DEPEND and RDEPEND atoms are treated alike; their relative ordering is a pass-2 concern.

22.9.3 Step 3 — Resolving python

The candidate `python-3.13.2` is selected. A `selected_cn` constraint is emitted into the global ConstraintsAVL (not the context). The context itself is passed down unchanged from the grouped dependency.

22.9.4 Step 4 — Expanding python's dependencies

Python's own dependencies get fresh contexts. Notice how each tag is rebuilt at this level:

- `self` now points to `python-3.13.2`, not to portage. The `self` tag always records the immediate parent, never accumulates.
- `build_with_use` is replaced based on the new atom. `dev-libs/openssl:=` has no bracketed flags, so the USE state becomes empty.
- `after` — if an `after/1` marker arrived with the incoming context (e.g. the target was an `@world` anchor), it is propagated onward into python's dep contexts.
- **Slot lock** — the `:=` operator on `dev-libs/openssl:=` adds a `slot(dev-libs,openssl,0/3.4.1)` tag to the context, recording the sub-slot for rebuild tracking.
- `after_only` — had python carried a PDEPEND, that edge would get `after_only(python:install)`, later rewritten to an `order_after` preference that does not propagate to the child's own deps.

22.9.5 Key observations

- `self/1` always points to the immediate parent, never accumulates along the chain.
- `build_with_use` is replaced at each edge based on the dependency atom's bracketed flags.
- `after/1` (from world-set anchors) propagates down the tree; `after_only/1` (from PDEPEND edges) does not. DEPEND-vs-RDEPEND ordering is decided in pass 2, not by context markers.
- **Slot locks** (`:=`) add `slot/3` entries to the context.
- **Constraint emissions** (e.g. `selected_cn`) go into the global ConstraintsAVL, not into the context.

22.10 Design rationale

22.10.1 Why feature unification?

Traditional dependency solvers use flat constraint lists or SAT clauses. portage-ng uses feature-term unification because:

1. **Composability**: Contexts from different proof branches merge naturally at join points without ad-hoc conflict resolution.

2. **Bounded growth**: The `self/1` stripping in feature term unification and the per-edge `build_with_use` replacement prevent unbounded context growth along dependency chains.
3. **Domain extensibility**: New context tags can be added without changing the merge infrastructure — just add a `val_hook/3` clause if domain-specific merge is needed.
4. **Conflict detection**: The merge fails (backtracks) on contradictions (e.g. a flag in both enable and disable), providing natural constraint propagation.

22.10.2 Why separate contexts and constraints?

Contexts are **local** (per-literal, scoped to a proof branch) while constraints are **global** (shared across the entire proof). This separation allows:

- **Contexts** to carry provenance information that should not leak across unrelated proof branches.
- **Constraints** to enforce global invariants (e.g. only one version of a package can be selected) that must hold across the entire proof.
- **Constraint learning** to persist across reprove retries, narrowing the search space incrementally.

Chapter 23

Dependency Resolver Comparison

23.1 Architecture Overview

All four resolvers solve the same problem: given a set of requested packages, figure out which concrete versions to install and in what order. Where they differ is in how they handle **conflicts** — situations where the first choice turns out to be wrong.

Each subsection below describes the resolver's strategy and illustrates its conflict-resolution loop.

23.1.1 Portage (Python)

Portage takes the most straightforward approach. It builds a dependency graph by walking every dependency and picking the newest stable candidate for each. If two packages end up claiming the same slot, Portage detects the conflict after the graph is already built.

Its recovery strategy is blunt: mask the conflicting package so it won't be picked again, throw away the entire graph, and rebuild it from scratch. Each retry adds one more mask. The masks accumulate across retries, but no other information carries over — the graph starts clean every time. Portage allows up to 20 retries by default (configurable with `--backtrack=N`).

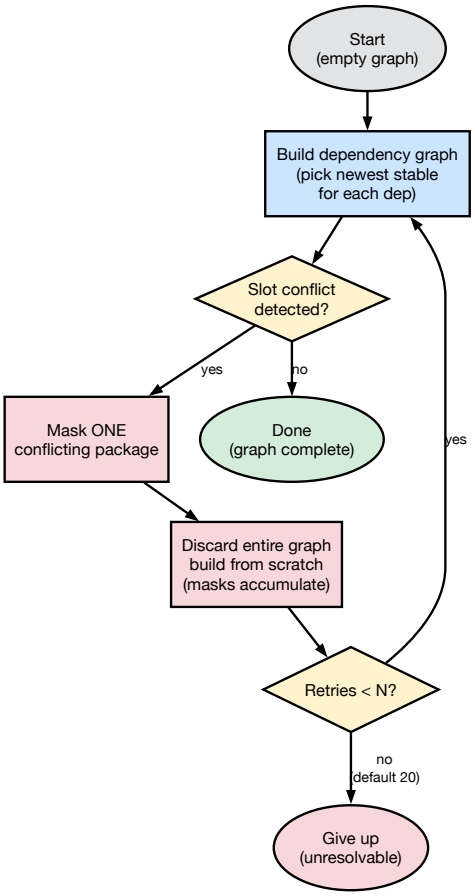


Figure 61 — Portage conflict-resolution loop

Because each retry rebuilds everything, this approach is the slowest of the four. Complex dependency tangles — like the OCaml Jane Street ecosystem — can require more than a dozen retries before Portage finds a consistent graph.

23.1.2 pkgcore (Python)

pkgcore's `pmerge` resolver is also Python, but it does **not** copy Portage's rebuild-with-masks loop. Resolution is a depth-first walk over an explicit **frame stack** (`resolver_stack` / `resolver_frame` in `pkgcore.resolver.plan`): each atom pushes a frame, tries a choice, and walks that choice's dependency set.

When a choice fails — inserting it into the plan state fails, or a dependency under it cannot be satisfied — pkgcore **backtracks to the frame's checkpoint** (`state.backtrack(start_point)`), advances to the next remaining package for that atom (`force_next_pkg`), and continues inside the same `merge_plan`. Failed alternatives can also be pruned from the choice set (`reduce_solutions`). There is no global mask list carried into a fresh graph, and no Paludis-style preload that names the winning candidate for the next full restart.

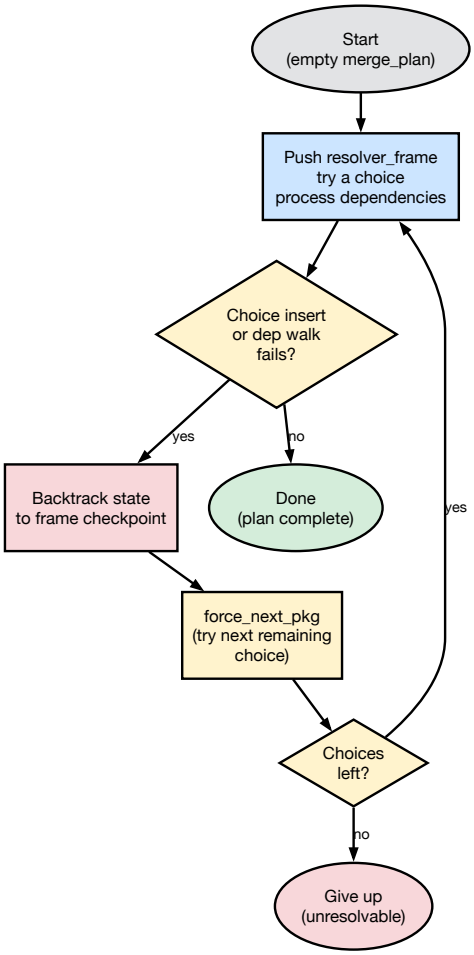


Figure 62 — pkgcore conflict-resolution loop

Relative to Portage, this is a real improvement: work already done above the failing frame is kept, and only the open choice point is revisited. Relative to Paludis and portage-ng, the guidance is still mostly **negative and local** — "try the next candidate" — rather than a positively learned domain or a computed "use this package next time." Deep, blocked search spaces can still explore a large fraction of the choice tree (and historically could blow the Python recursion limit before the frame rewrite moved the stack out of the call stack).

23.1.3 Paludis (C++)

Paludis is smarter about what it remembers. Instead of masking wrong candidates, it identifies the **right** one. When a new constraint conflicts with an earlier decision, Paludis evaluates all accumulated constraints for that package simultaneously and determines which candidate satisfies them all.

It then records a *preload* — an instruction that says "use this specific candidate next time." The resolver is discarded and a fresh one is created, but the preloads travel with it. This means the next attempt starts with positive guidance rather than just a list of things to avoid.

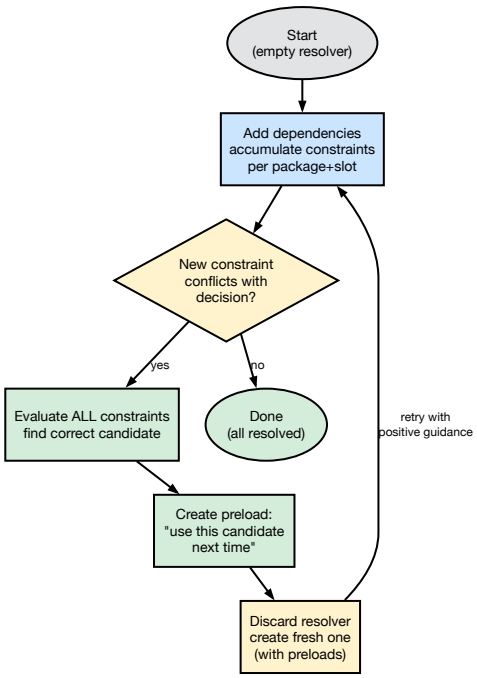


Figure 63 — Paludis conflict-resolution loop

Because Paludis carries forward the right answer instead of just rejecting the wrong one, it typically needs fewer restarts than Portage. However, each restart still creates a brand-new resolver, so the dependency walk itself is repeated.

23.1.4 portage-ng (SWI-Prolog)

portage-ng avoids the restart-from-scratch pattern altogether. It uses a depth-first proof search: each dependency becomes a proof obligation, and selecting a candidate adds constraints to a global store. Constraint guards monitor the store and fire immediately when a conflict appears.

When a guard fires, three things happen in sequence:

1. The conflicting domain is **learned** — the version set for that package is narrowed to exclude impossible choices.
2. The current candidate is **rejected** so it won't be tried again.
3. Only the affected **subtree is retried**, with the learned domain already in place to guide candidate selection.

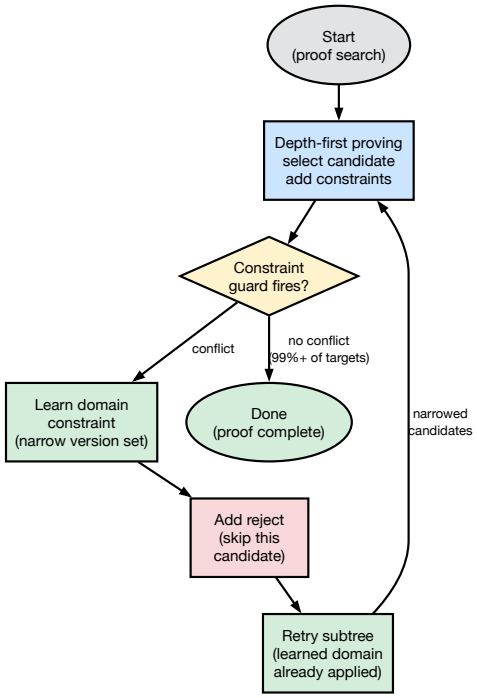


Figure 64 — portage-ng conflict-resolution loop

For the vast majority of packages (over 99%), no conflict arises at all and the proof completes in a single pass. When conflicts do occur, the combination of learned domains (positive guidance) and rejects (negative filtering) resolves them without rebuilding the entire proof tree. This makes portage-ng the fastest of the four resolvers.

23.2 Comparison Table

Aspect	Portage	pkgcore	Paludis	portage-ng
Language	Python	Python	C++	SWI-Prolog
Conflict detection	Post-hoc (after graph built)	Incremental (during frame / choice walk)	Incremental (on constraint add)	Incremental (constraint guard)
What carries across retries	Masks (negative)	Remaining choices in the frame (negative pruning)	Preloads (positive)	Learned domains (positive) + Rejects (negative)
Fresh state each retry?	Yes (new dep-graph)	No — backtrack to frame checkpoint	Yes (new Resolver)	Partial (reject set accumulates, learned store accumulates)
Finding the right candidate	Brute force (mask+retry)	`force_next_pkg` after backtrack	`_try_to_find_decision_for` with ALL constraints	Domain narrowing (Zeller) + priority resolution (Vermeir)
Performance	Slowest (full rebuild)	Faster than Portage (keeps parent frames)	Fast (targeted restarts)	Fastest (single-pass for most targets)
Package-specific code	None	None	None	None

23.3 Slot Allocation: Pigeonhole Reasoning

Gentoo slots turn part of dependency resolution into an allocation problem: every selected version occupies exactly one hole — its (package, slot) pair — no hole may host two occupants, and different holes of the same package may legitimately coexist (`gcc:12` next to `gcc:13`). pkgcore names this structure literally: its slot tracker is a class called `PigeonHoledSlots`. Constraint-programming solvers such as the Glasgow Constraint Solver use the pigeonhole *principle* as a first-class reasoning device. All three systems compared below enforce the same invariant, but at three different strengths: pkgcore **detects** a collision when it happens, CP propagators **preclude** whole families of collisions before search branches, and portage-ng **detects and learns from** each collision. (Portage sits before all three: as described above, it notices two packages claiming the same slot only after the graph is fully built, then masks and rebuilds.)

23.3.1 pkgcore: the hole as an occupancy table

`PigeonHoledSlots` is a mutable registry consulted by `merge_plan` during its frame-stack walk. A dictionary keyed by package maps to the current occupants; `fill_slotting(obj)` scans for an existing occupant with the same slot and, on a hit, returns the conflicting objects instead of inserting. Blockers reuse the same structure as *limiters* — anti-pigeons registered per key via

`add_limiter`, which poison the hole against any matching occupant. On a returned conflict the resolver backtracks to the frame checkpoint and advances to the next candidate.

The name is the metaphor, not the mathematical principle. Detection is eager but **pairwise**: a conflict is noticed only when the second pigeon arrives at the hole. The knowledge gained is **negative and local**: the colliding objects are reported, the choice list is pruned, and nothing narrows future candidate selection.

23.3.2 Constraint programming: the hole as a counting argument

In CP solvers the pigeonhole principle appears as the propagation semantics of global constraints such as `allDifferent`: "these five jobs have only four time slots between them, so by a pigeonhole argument the problem is infeasible." Régin's matching-based propagator (AAAI 1994) and Puget's Hall-interval bounds consistency detect that k variables collectively reach fewer than k values in polynomial time, *before* the search tree branches. This counting argument is exactly where resolution-based SAT solvers struggle — pigeonhole formulas require exponential resolution proofs (Haken 1985) — which is why the CP community treats `allDifferent` propagation, and the Glasgow Constraint Solver's proof-logging work certifying it, as a genuinely different reasoning class rather than an implementation detail.

23.3.3 portage-ng: the hole as a feature dimension

portage-ng has neither an occupancy table nor a cardinality propagator. Slot allocation emerges from three mechanisms of the proof search:

Slots are a dimension of the version domain. Every `version_domain(Slots, Bounds)` carries a slot set next to its version bounds, and `domain_meet` intersects both dimensions at once (Chapter 10). Two requirements on the same package whose slot sets are disjoint meet to `slots([])`, which is structurally inconsistent — the proof fails *before any candidate is enumerated*:

```
version_domain:meet_slot_domains(slots(S1), slots(S2), slots(S)) :-
  ord_intersection(S1, S2, S).

version_domain:domain_inconsistent(version_domain(slots([]), _Bounds)).
```

This is a small pigeonhole-style cut in the CP spirit — infeasibility derived by set algebra rather than by attempting an insertion — though unary (per package), not a cross-package counting argument.

Occupancy is a constraint guard, not a table. The counterpart of `fill_slotting` is the `selected_cn(C,N)` ordset accumulated in the constraint store; `cnselect:selected_cn_unique_or_reprove/4` enforces at most one concrete entry per (C,N) — or per (C,N,slot) where multislot coexistence applies — each time feature unification merges a new selection into the store.

A collision is converted into knowledge. Where pkgcore returns the colliding objects, the portage-ng guard *learns*: it stores a narrowed `cn_domain(C,N,Slot)` via `prover:learn/3`, rejects the conflicting candidate, and re-proves only the affected subtree with the narrowed domain already applied to candidate selection. If the conflict survives all retries it is memoized

(`memo:slot_conflict_/3`) and surfaces as a `slot_conflict` domain assumption — a negative, blocking outcome — rather than a silent failure. Blockers get the same treatment as pkgcore's limiters conceptually, but are tracked as constraints with source snapshots and degrade to actionable blocker assumptions.

23.3.4 Why portage-ng skips cardinality propagation

portage-ng does not perform Glasgow-style cross-package counting: a hypothetical "five packages competing for four holes" is discovered through the collide–learn–retry loop, not refuted up front by a matching argument. The Gentoo domain almost never presents that structure. Slots are scoped per package — `gcc:12` and `gcc:13` are holes belonging to `sys-devel/gcc` alone, never a pool that unrelated packages compete for — so the `allDifferent` pattern (many variables drawing from one shared value set) essentially does not arise. The hole structure is also *ragged*: which slot a candidate occupies is metadata of the chosen version, so the pigeon determines its own hole; and sub-slot (`:=`) rebuilds make occupancy dynamic, beyond static allocation entirely. What the domain actually needs is per-package unary domains with slot as a feature dimension, pairwise consistency for slot operators, and good conflict recovery — which is precisely the narrowing-plus-learning design described above.

Aspect	pkgcore	CP (Glasgow-style)	portage-ng
Pigeonhole meaning	Occupancy table (metaphor)	Counting principle (Hall sets, matching)	Guard invariant + slot-set algebra
Detection moment	Insertion of second occupant	Before branching (propagation)	Constraint merge / guard evaluation
Knowledge from a conflict	List of colliding objects	Pruned domains, certified cuts	Learned `cn_domain` + reject set
Recovery	Backtrack, next candidate	Pruned before search (else backjump)	Re-prove subtree with narrowed domain
Cross-package counting	No	Yes	No (domain rarely needs it)

See Chapter 10 for the domain algebra and Chapter 9 for the learning and reprove mechanics used above.

23.4 Academic Foundations

The notes below name the results that the mechanisms cite. How those lines relate to each other, to smodels, clasp, DLV, the Glasgow solvers and the current package solvers, and what portage-ng takes from each, is Chapter 25: Position in the Solver Literature.

23.4.1 Zeller & Snelting: Feature Logic (ESEC 1995, TOSEM 1997)

"Handling Version Sets through Feature Logic" (ESEC 1995, LNCS 989) and its expanded journal version "Unified Versioning Through Feature Logic" (TOSEM 1997, Vol. 6 No. 4) — version sets are identified by feature terms and configured by incrementally narrowing

the set until each component resolves to a single version. portage-ng's `version_domain` with `domain_meet` (intersection) is essentially Zeller's feature term narrowing. The learned constraint store implements Zeller's feature implication propagation: constraints discovered in one proof attempt propagate to narrow version sets in the next attempt.

23.4.2 Vermeir & Van Nieuwenborgh: Ordered Logic Programs (JELIA 2002)

"Preferred Answer Sets for Ordered Logic Programs" — when rules conflict, a partial order determines which yields. portage-ng's `find_adjustable_origin` implements this: when a domain is inconsistent (two bounds that can't be simultaneously satisfied), the bound from the "adjustable" origin (the package that already has a learned constraint) is dropped, and the origin is narrowed further.

23.4.3 CDCL / PubGrub / SAT-based approaches

Modern package resolvers (libsolv, Resolvo, PubGrub) encode version constraints as boolean satisfiability problems. portage-ng's approach is different: it uses proof search with domain narrowing rather than SAT encoding. The learned constraint store is analogous to CDCL's learned clauses, but expressed as version domains rather than boolean clauses.

23.4.4 Any-of (`||`) arm preference

Portage's `dep_zapdeps` `choice_bins` and portage-ng's `ranking:prioritize_deps_keep_all/3` multi-key sort are compared in detail in Chapter 12, Any-of (`||`) arm selection (including why overlapping `||` DNF, virtual expand, and circular demotion inside `||` are not mirrored as ranking keys).

Chapter 24

Dependency Ordering

When a package manager installs several packages at once, the order in which they are merged matters. A compiler must be installed before the packages that need it for building; a shared library must be present before the programs that link against it can run. The Gentoo Package Manager Specification (PMS) defines five dependency types, each with different ordering strength. Portage, Paludis, and portage-ng all interpret these types, but they differ in how strictly they enforce ordering — especially when cycles make a perfect order impossible.

24.1 Dependency types

The PMS (Chapter 8) groups dependencies into five classes based on *when* the dependency must be available.

- **DEPEND / BDEPEND** — build-time dependencies. These must be installed and usable before `pkg_setup` runs and throughout the `src_*` build phases. BDEPEND (introduced in EAPI 7) targets the build host (CBUILD), while DEPEND targets the target host (CHOST). Both create **hard requirements**: the dependency must be merged before the dependent can be built.
- **RDEPEND** — runtime dependencies. These must be installed before the package is "treated as usable." This is a **soft requirement** — a *preference*: ideally satisfied before the dependent is merged, but it may give way when cycles exist.
- **PDEPEND** — post-dependencies. These only need to be installed "before the package manager finishes the batch." This is the **weakest constraint** — there is no requirement that the post-dependency is merged before the dependent.
- **IDEPEND** — install-time dependencies (EAPI 8+). Needed during `pkg_preinst` and `pkg_postinst`. Treated similarly to runtime dependencies for ordering purposes.

24.2 Phase functions and dependency availability

Different ebuild phases have access to different dependency classes:

Build phases (`src_unpack` through `src_install`) DEPEND, BDEPEND
Install phases (`pkg_preinst`, `pkg_postinst`, `pkg_prerm`, `pkg_postrm`) RDEPEND, IDEPEND
Configuration (`pkg_config`) RDEPEND, PDEPEND

24.3 Portage's approach

Portage builds a single dependency graph where every package is a node and every dependency creates a directed edge. Each edge carries a *priority* that records how binding the edge is:

Dependency	Priority	Breakable?
DEPEND / BDEPEND	buildtime (highest)	Never
RDEPEND with `:=` slot op	runtime_slot_op	Only when cross-compiling
RDEPEND	runtime	Yes, for cycles
PDEPEND	runtime_post (lowest)	Yes, first to break

24.3.1 Progressive relaxation

When Portage runs its topological sort and cannot find a leaf node (a package with no unsatisfied dependencies), it progressively drops weaker edges until a leaf appears. The relaxation proceeds in four passes:

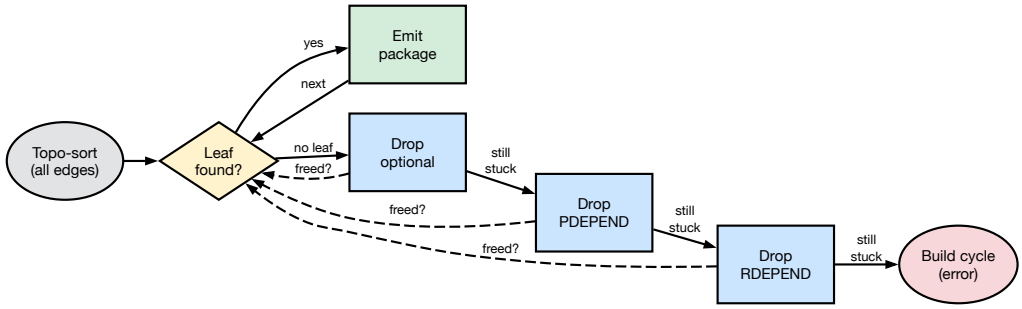


Figure 65 — Portage progressive edge relaxation

1. **All edges respected** — standard topological sort.
2. **Drop optional edges** — removes edges for optional dependencies.
3. **Drop PDEPEND edges** — the weakest real dependency type is discarded.
4. **Drop RDEPEND edges** — runtime edges are relaxed, freeing most remaining cycles.

If no leaf is found even after dropping RDEPEND edges, the remaining cycle involves only build-time dependencies. This is a hard error — Portage cannot resolve it and reports the cycle to the user.

24.3.2 What this means in practice

In the common (acyclic) case, Portage merges all dependencies — including RDEPEND — before the packages that need them. When cycles exist, PDEPEND edges are broken first, then RDEPEND edges. Build-time edges are never broken.

24.4 Paludis's approach

Paludis takes a different view. It builds a Node-Adjacency Graph (NAG) where edges carry two boolean flags: `build()` for build-time dependencies and `run()` for runtime dependencies.

Notably, **PDEPEND creates no edge at all**. The Paludis source comments explain the reasoning: most post-dependencies already depend on the thing requiring them anyway, so adding a backwards edge would just create unnecessary cycles.

24.4.1 SCC-based cycle handling

Rather than relaxing edges progressively, Paludis uses Tarjan's algorithm to find strongly connected components (SCCs) and then classifies each one:

- **Single-node SCC** — no cycle, scheduled directly.
- **Runtime-only SCC** (no build edges) — ordered arbitrarily. Paludis treats runtime-only cycles as having no ordering significance at all.
- **Build-dep SCC** — Paludis tries to break the cycle by removing edges whose dependencies are already satisfied (`build_all_met` or `run_all_met`). If the SCC is still cyclic after that, it is marked as unorderable.

The key insight from Paludis is stronger than Portage's progressive relaxation: runtime-only cycles are not just *breakable* — they are *free to order however is convenient*.

24.5 portage-ng's approach

portage-ng models dependencies as a two-phase proof tree. Each package has an `:install` action (building) and a `:run` action (being usable). The different dependency types map naturally onto this structure.

24.5.1 Intra-group dependency ordering

Before proving the dependencies within a single package, `ranking:dep_priority/2` sorts them by constraint tightness. Tightly constrained dependencies are proved first so that their `selected_cn` locks early, preventing greedy conflicts where an unconstrained sibling picks a version that later clashes. The priority ladder (lower = proved first): tight upper bound (1) → tilde (4) → wildcard (8) → unconstrained (999). Within each tier, slot specificity further refines the order. See Chapter 12 for details.

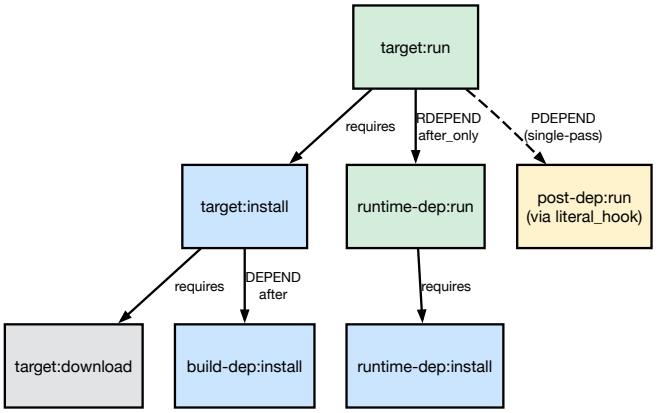


Figure 66 — portage-ng two-phase dependency model

- **DEPEND / BDEPEND** create edges to `:install` actions with `after()` context tags that propagate down the dependency chain. These are hard requirements (`requires/2` in the ordering bindings).
- **RDEPEND** create edges to `:run` actions with `after_only()` context tags that stop at the immediate children. These are soft requirements — preferences (`prefers/2`).
- **PDEPEND** are handled by the `heuristic:proof_obligation/4` hook in a single pass during proof search, without creating explicit ordering edges in the proof.

When cycles appear, the ordering pass (Chapter 13) resolves them at proof time: a requirement whose provider is still being scheduled falls through to a citation of the installed world (VDB), or — when nothing bridges the loop — to an honest `unreachable` assumption. Runtime-only cycles never bind at all, because RDEPEND edges are preferences (matching Paludis's insight) that are void or dropped when they lie on a cycle (Chapter 13, `--optimize`).

24.5.2 PDEPEND completion ordering

Although PDEPEND creates no edge during proof search, a package `P` that declares PDEPEND is only fully functional once its post-dependencies are merged. If another package consumes `P` and starts building before `P`'s PDEPEND closure is installed, its build can fail (e.g. a Ruby extension's `extconf.rb` hits a `LoadError`, or a CMake `CMAKE_C_COMPILER` check fails because a toolchain component is not yet on `PATH`).

The ordering bindings close this gap with a **completion preference** (`ordering:prefers/2` in `Source/Domain/Gentoo/Rules/ordering.pl`): a consumer whose dependency carries an `order_after` anchor on a PDEPEND provider prefers the install heads of that provider's PDEPEND targets — i.e. it is ordered after `P`'s whole post-install group, matching emerge's behaviour (portage-ng#18).

Because this is a *preference*, not a hard requirement, it is inherently cycle-safe: under the default `--optimize parallelism` strategy a preference whose two ends are mutually reachable is void in the rules (`ordering:same_component/2`); under `--optimize soft-requirements` the wave projection accepts each preference exactly when it closes no cycle against the hard edges and the previously accepted preferences (Chapter 13, "Preferences: soft requirements"). A consumer that is itself a member of the provider's PDEPEND group is therefore never bumped — the preference back onto its own group lies on a loop and is void / dropped silently (portage-ng#19). Densely cyclic toolchain closures (e.g. LLVM) are safe for the same reason: no preference can collapse the ordering of an acyclic chain elsewhere in the plan (portage-ng#26).

A per-pass PDEPEND anchor index makes the preference lookup cheap; plans without any PDEPEND provider pay only an empty index probe.

24.6 How the three approaches compare

The diagram below traces how each PMS dependency type is implemented across the three resolvers. Blue indicates hard (build-time) requirements, green indicates soft (runtime) requirements — preferences — and yellow the weakest (post-dependency) edges.

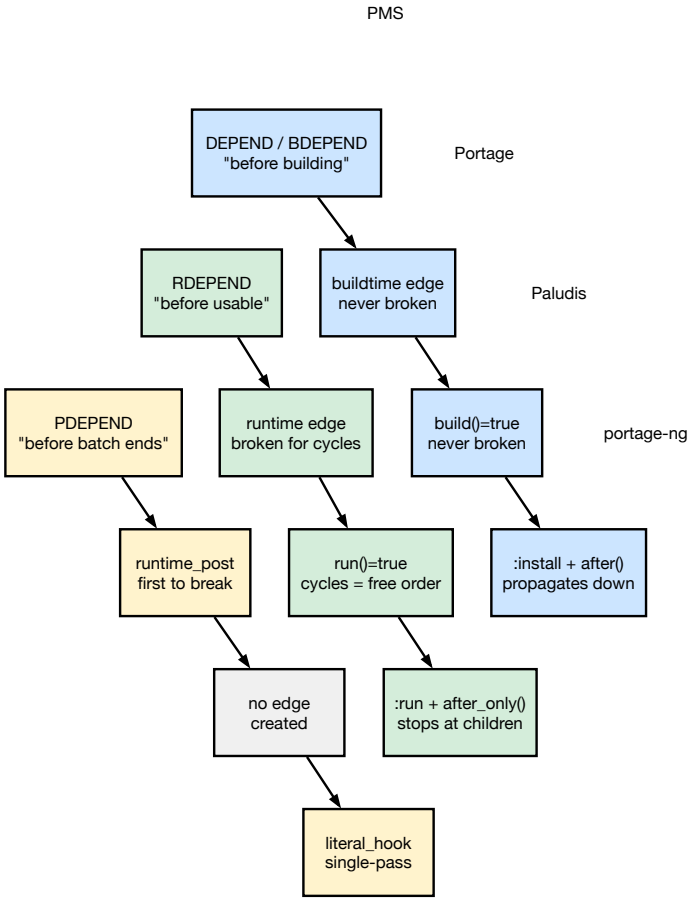


Figure 67 — Dependency type mapping across resolvers

Aspect	Portage	Paludis	portage-ng
DEPEND/BDEPEND	Hard edge, never broken	Hard edge, never broken	`:install` + `after()`, hard
RDEPEND	Soft edge, broken for cycles	Soft edge, cycles freely ordered	`:run` + `after_only()`, soft
PDEPEND	Weak edge, first to break	No edge at all	proof obligation hook, no proof edge; completion preference in ordering pass
Cycle strategy	Progressive relaxation	SCC classification	World citation (VDB) or `unreachable` assumption
Build-time cycles	Error / merge group	Relax met edges, then error	Bridged by installed world; honest bootstrap boundary otherwise

24.7 Annex: overlay test cases

portage-ng ships with a synthetic overlay (`Repository/Overlay/`) containing 80 test cases that exercise the resolver in isolation. Each test uses tiny packages with carefully crafted dependency relationships, making it easy to see exactly how the resolvers behave. The five cases below illustrate the ordering concepts discussed in this chapter. For each case we show the dependency graph, a short description, and the captured output from both Portage (`emerge -vp`) and portage-ng (`--pretend`).

24.7.1 Annex A — Basic ordering (test01)

Four packages with a clean dependency chain: `web` depends on `app`, `db`, and `os`; `app` depends on `db` and `os`; `db` depends on `os`. No cycles, no special dependency types.

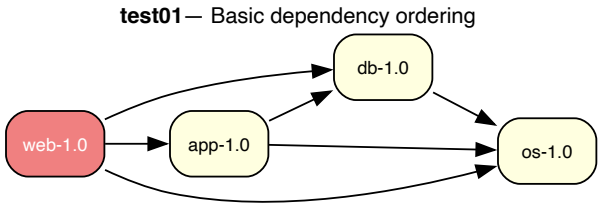


Figure 68 — test01 — Basic dependency ordering

Both resolvers produce the same order: `os` first (no dependencies), then `db` and `app` (whose dependencies are now satisfied), and finally `web`. portage-ng additionally shows the two-phase `:install` / `:run` structure and groups downloads into a parallel first step.

Portage output:

```
[ebuild  N     ] test01/os-1.0::overlay   0 KiB
[ebuild  N     ] test01/db-1.0::overlay   0 KiB
[ebuild  N     ] test01/app-1.0::overlay  0 KiB
[ebuild  N     ] test01/web-1.0::overlay  0 KiB

Total: 4 packages (4 new)
```

portage-ng output:

```
  step  1 | download  overlay://test01/web-1.0
          | download  overlay://test01/os-1.0
          | download  overlay://test01/db-1.0
          | download  overlay://test01/app-1.0

  step  2 | install   overlay://test01/os-1.0
  step  3 | run       overlay://test01/os-1.0
  step  4 | install   overlay://test01/db-1.0
  step  5 | run       overlay://test01/db-1.0
  step  6 | install   overlay://test01/app-1.0
  step  7 | run       overlay://test01/app-1.0
  step  8 | install   overlay://test01/web-1.0
  step  9 | run       overlay://test01/web-1.0
```

```
Total: 12 actions (4 downloads, 4 installs, 4 runs)
```

24.7.2 Annex B — Transitive RDEPEND (test50)

`app` has a compile-time dependency on `foo`, and `foo` has a runtime dependency on `bar`. The question is whether `bar` — a transitive runtime dependency of a build dependency — appears in the merge plan.

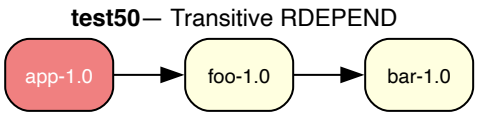


Figure 69 — test50 — Transitive RDEPEND

Both resolvers correctly include all three packages. The ordering is `bar` first (so `foo` can run), then `foo` (so `app` can build), then `app`.

Portage output:

```
[ebuild  N     ] test50/bar-1.0::overlay  0 KiB
[ebuild  N     ] test50/foo-1.0::overlay  0 KiB
[ebuild  N     ] test50/app-1.0::overlay  0 KiB

Total: 3 packages (3 new)
```

portage-ng output:

```
  step  1 | download  overlay://test50/foo-1.0
          | download  overlay://test50/bar-1.0
          | download  overlay://test50/app-1.0

  step  2 | install   overlay://test50/bar-1.0
  step  3 | run       overlay://test50/bar-1.0
  step  4 | install   overlay://test50/foo-1.0
  step  5 | install   overlay://test50/app-1.0
  step  6 | run       overlay://test50/app-1.0

Total: 8 actions (3 downloads, 3 installs, 2 runs)
```

24.7.3 Annex C — Runtime cycle (test07)

Same four-package graph as Annex A, but `os` adds a **runtime** dependency back on `web`, creating a cycle. The back-edge is an RDEPEND, so both resolvers treat the cycle as benign.

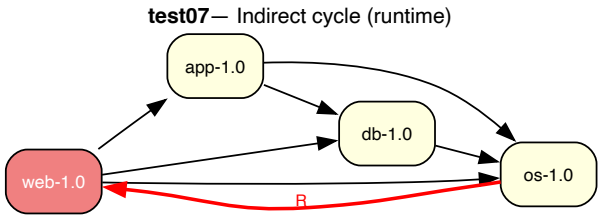


Figure 70 — test07 — Indirect cycle (runtime)

Portage reports the cycle as a warning but still produces a valid merge list (using 1 backtrack). portage-ng classifies it as benign and produces a clean plan with no assumptions — notice that `web`, `app`, and `db` are installed in parallel in step 3 because the runtime cycle makes their relative order irrelevant.

Portage output:

```
[ebuild  N     ] test07/web-1.0::overlay  0 KiB
[ebuild  N     ] test07/app-1.0::overlay  0 KiB
[ebuild  N     ] test07/db-1.0::overlay   0 KiB
[ebuild  N     ] test07/os-1.0::overlay   0 KiB

Total: 4 packages (4 new)

 * Error: circular dependencies:
(test07/os-1.0::overlay) depends on
 (test07/web-1.0::overlay) (runtime)
  (test07/os-1.0::overlay) (buildtime)
```

portage-ng output:

```
  step  1 | download  overlay://test07/web-1.0
          | download  overlay://test07/os-1.0
          | download  overlay://test07/db-1.0
          | download  overlay://test07/app-1.0

  step  2 | install   overlay://test07/os-1.0
  step  3 | install   overlay://test07/web-1.0
          | install   overlay://test07/app-1.0
          | install   overlay://test07/db-1.0
  step  4 | run       overlay://test07/web-1.0
          | run       overlay://test07/app-1.0
          | run       overlay://test07/db-1.0
  step  5 | run       overlay://test07/os-1.0

Total: 12 actions (4 downloads, 4 installs, 4 runs)
```

24.7.4 Annex D — PDEPEND (test66)

`app` depends on `lib` (compile-time), and `lib` declares `plugin` as a PDEPEND. The plugin should be resolved, but it does not need to be installed before `lib`.

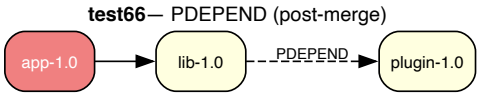


Figure 71 — test66 — PDEPEND (post-merge)

Portage merges `plugin` first (it has no hard dependencies), then `lib`, then `app`. portage-ng installs `lib` and `plugin` in parallel in step 2, since PDEPEND creates no ordering requirement between them here. The plugin's `:run` action comes last, after the main target. (When a package *outside* the PDEPEND closure consumes the provider, the ordering pass's PDEPEND completion preference additionally orders that consumer after the provider's post-install group; this minimal test has no such external consumer.)

Portage output:

```
[ebuild  N     ] test66/plugin-1.0::overlay  0 KiB
[ebuild  N     ] test66/lib-1.0::overlay     0 KiB
[ebuild  N     ] test66/app-1.0::overlay     0 KiB

Total: 3 packages (3 new)
```

portage-ng output:

```
  step  1 | download  overlay://test66/plugin-1.0
          | download  overlay://test66/lib-1.0
          | download  overlay://test66/app-1.0

  step  2 | install   overlay://test66/lib-1.0
          | install   overlay://test66/plugin-1.0
  step  3 | run       overlay://test66/lib-1.0
  step  4 | install   overlay://test66/app-1.0
  step  5 | run       overlay://test66/app-1.0
  step  6 | run       overlay://test66/plugin-1.0

Total: 9 actions (3 downloads, 3 installs, 3 runs)
```

24.7.5 Annex E — PDEPEND cycle (test79)

`server` has an RDEPEND on `client`, and `client` has a PDEPEND back on `server`. This creates a cycle, but since PDEPEND creates no ordering edge, the cycle is naturally broken.

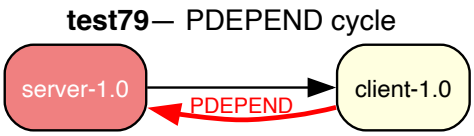


Figure 72 — test79 — PDEPEND cycle

Portage handles this cleanly — the PDEPEND obligation is already satisfied because `server` was merged as part of the same batch. portage-ng's proof obligation mechanism resolves the PDEPEND in a single pass with no assumptions needed.

Portage output:

```
[ebuild  N     ] test79/client-1.0::overlay  0 KiB
[ebuild  N     ] test79/server-1.0::overlay  0 KiB

Total: 2 packages (2 new)
```

portage-ng output:

```
  step  1 | download  overlay://test79/server-1.0
          | download  overlay://test79/client-1.0

  step  2 | install   overlay://test79/client-1.0
  step  3 | run       overlay://test79/client-1.0
  step  4 | install   overlay://test79/server-1.0
  step  5 | run       overlay://test79/server-1.0

Total: 6 actions (2 downloads, 2 installs, 2 runs)
```

24.8 References

- PMS Chapter 8: https://projects.gentoo.org/pms/8/pms.html
- Portage source: `lib/_emerge/depgraph.py` (method `_serialize_tasks`)
- Portage priorities: `lib/_emerge/DepPriorityNormalRange.py`
- Paludis orderer: `paludis/resolver/orderer.cc`
- Paludis classifier: `paludis/resolver/labels_classifier.cc`
- Full overlay test suite: `Documentation/Tests/README.md` (80 test cases)

Chapter 25

Position in the Solver Literature

Chapter 23 compares portage-ng with Portage, pkgcore and Paludis, and ends with a short list of the papers those mechanisms cite. This chapter is the longer version of that list. It says which results the prover actually uses, which neighbouring techniques it leaves alone, and which problems in that literature are still open.

The work falls into two groups. One grounds a logic program and then searches the propositional theory. The other narrows feature domains and propagates before it branches. The package managers in production took the conflict-learning half of the first group and specialised it to versions. portage-ng took the goal-directed half, kept the learned constraint, and stores that constraint as a version domain.

A proof that starts from the target only builds the part of the theory that the target can reach. Packages outside the proof are not in the plan, so grounding them first would add work and no candidates.

25.1 Grounding and goal-directed proof

In 2005 the practical answer-set systems were smodels (Niemelä, Simons, Syrjänen) and DLV (Leone and others). Both compute stable models of a program that a grounder — `lparse`, or DLV's own instantiator — has already flattened. smodels was fast because of lookahead: at each node it trial-assigns the remaining literals, runs the well-founded closure `expand`, and branches on the literal that forces the most. Patrik Simons' thesis, *Extending and Implementing the Stable Model Semantics* (2000), describes the algorithm. That speed was real, and hard to recover from the propagation rules alone. An ordered-logic solver from the same year, OLPS (Van Nieuwenborgh, Heymans and Vermeir, PADL 2005), implements a 9-valued status lattice and loses to smodels as soon as the constraint graph gets dense.

Flattening a Portage tree up front does not fit that design. Every version, every USE flag and every arm of every `||` would be propositional before the first branch, including packages the target will never touch. The alternative is `Source/Logic/context.pl`: a rule is evaluated in the OO context where it is defined, so the global Prolog namespace is never the Herbrand base. See Chapter 21. The prover tightens this further by proving outward from the target. An ebuild the goal never reaches costs nothing.

Answer-set systems are still working on instantiating less of an arbitrary program. clingo and DLV2 remain ground-then-solve. Lazy grounding, body-decoupled grounding, compilation of a rule into a propagator, and papers such as FastFound and Diminution (2025) all try to instantiate less of an arbitrary program. Here the query is the target, and the rules that can

constrain a selection live on the packages that selection names. That only works because a Gentoo dependency is an atom. The same procedure does not ground an arbitrary disjunctive program.

25.2 Feature logic and ordered logic

Two older results are used as mechanisms.

Zeller and Snelting (*Handling Version Sets through Feature Logic*, ESEC 1995; *Unified Versioning through Feature Logic*, TOSEM 1997) identify a version set with a feature term and configure it by narrowing until one version remains. `version_domain(Slots, Bounds)` and `domain_meet` are that narrowing. A learned `cn_domain` is their feature implication, carried from one proof attempt to the next. See Chapter 10.

Van Nieuwenborgh and Vermeir (*Preferred Answer Sets for Ordered Logic Programs*, JELIA 2002; TPLP 2006) let a partial order decide which rule yields when rules conflict. Candidate order in `cache:ordered_entry/5` is the single-order case: newer versions are tried first. The closer paper for the rest of the resolver is *On Programs with Linearly Ordered Multiple Preferences* (Van Nieuwenborgh, Heymans and Vermeir, ICLP 2004). A stack of preference relations keeps a solution that is best at level 1, then best at level 2 among those, and a violation at a lower level is never traded against a higher one. `ranking:choice_criteria/1` is that stack, compared lexicographically. The five-tier fallback — strict, keyword acceptance, blockers, unmask, keyword-and-unmask — works the same way, and so does the split between hard `requires/2` and soft `prefers/2`.

A few neighbouring papers describe machinery that is already in the prover. One of them lines up with an actual Gentoo rule:

Result	What it describes	Where the prover does it
Order and Negation as Failure (ICLP 2003)	Order plus negation-as-failure adds nothing a preference order cannot already say	`naf_cycle` and the `currently_proving` guard. A failed `not` on a cycle is an assumption, not a new kind of rule
Ordered Diagnosis and *Ordered Programs as Abductive Systems* (2003)	A preferred model is the set of rules you are willing to defeat so the rest stays consistent	Domain assumptions. Positive ones are the defeats that repair the plan; negative ones have no such repair. See Chapter 9
A Logic for Modeling Decision Making with Dynamic Preferences (De Vos and Vermeir, JELIA 2000)	Earlier decisions update the preference order	USE forces (`bwu_force`, `eq_follow`), flushed as one batched reprove
Specificity by Default (Geerts and Vermeir, ECSQARU 1995)	When two defaults conflict, the more specific one wins, with no extra priority table	Portage's package.use rule: a more specific atom beats a broader one. `preference:userconfig_use_match/3` is last-wins across matching specs

Weighted answer sets (LPAR 2004) score violations with a number. Preferences here are a declared list of criteria, compared in order, so a new preference is another entry in that list.

25.3 Answer-set solvers

The algorithm that replaced smodels settled around 2012. Gebser, Kaufmann and Schaub describe it in *Conflict-Driven Answer Set Solving: From Theory to Practice*. **clasp**, the solver inside **clingo** (Potsdam), is conflict-driven nogood learning: watched literals, first-UIP nogoods, activity heuristics and restarts, plus one piece kept from smodels — a source-pointer unfounded-set check that learns a loop nogood instead of recomputing the greatest unfounded set. **DLV2** (Calabria) pairs the I-DLV grounder with the WASP solver. It is the other production system, and the stronger of the two on disjunctive programs. smodels itself is historical.

What portage-ng uses from that design is the learned constraint. A learned `cn_domain` is a nogood whose atoms are versions. A deferred USE-force conflict already backjumps: the partial restart prunes the Triggers closure of the forced providers and keeps the rest of a finished pass. On a deep stack that is most of the proof. A `prover_reprove(cn_domain(...))` thrown in the middle of a pass still restarts from scratch, because the pass never finished and the cycle stack holds literals whose bodies are open. Carrying the partial restart over to that case is the clasp technique still missing. See `config:reprove_partial_restart/1`.

Three techniques from the same solvers were left out. Each one is a poor fit for the cost of a node in this search:

- **Lookahead.** Trial-assigning every open literal is how smodels got its speed, and it is what clasp later dropped. One trial here is a `rule/2` expansion plus a dependency-model build.

- **A greatest unfounded set.** smodels decides `not` that way, over the grounded program. The prover's `not` is the catch-all `rule(naf(_), [])` together with `prover:conflicts/2`: `naf(X)` holds unless `X` is already proved or already on the stack. The test is order-dependent, and weaker than well-founded negation. A plan can live with that. It does not need a closed-world false for every package it left uninstalled.
- **Every stable model of a disjunction.** smodels can return both models of a choice without a clause that walks the choice. A `||` here yields a solution because a choice-group rule walks the arms and the criterion list keeps one. The other arm would be a second plan for a dependency the first arm already satisfied.

OLPS's lattice T_9 spells the intermediate statuses out: no information, eventually true, founded true, the two false counterparts, the two "not" values, settled-unknown, and contradiction. A finished proof records three of them. A literal is in the model, assumed, or absent. Contradiction is `fail`, or a reprove exception, rather than a value stored on the literal. Adopting the rest of the lattice would give those statuses names. It would not make the search cheaper. See Chapter 8.

25.4 Why unused ebuilds can be skipped

Leaving the rest of the tree uninstantiated is sound for a reason that does not hold for an arbitrary logic program. An ebuild that no selected package mentions cannot satisfy a dependency, and cannot forbid one. In Gentoo, satisfaction goes through an atom.

What still has to be considered, even when the target never named it, is an installed package tied to something the proof did select. Those are read from the VDB:

- a `:=` consumer of a library being rebuilt, injected as a proof obligation once the provider is proved;
- a `PDEPEND`, injected the same way;
- a blocker atom inside an ebuild already selected, checked against what is installed;
- an installed reverse dependency that cannot accept a candidate, dropped by `candidate:candidate_reverse_deps_compatible_with_parent/2`;
- depclean and `@preserved-rebuild`, which start from the installed set and ask what `@world` still claims.

When one of those reverse edges is missing, the symptom is a wrong plan for a package already in the proof. That is how the hooks above were added. Each of them is an index over the installed system. Papers on lazy grounding worry that a rule never instantiated might have rejected the model. On this tree that rule would have to mention a selected package, and then one of these scans reaches it.

25.5 Constraint programming

The Glasgow Subgraph Solver (McCreesh, Prosser, Trimble) comes at the same family of problems from constraint programming. Variables are vertices of a small pattern. Domains are bitsets of vertices in a large target. Before any branch, propagators shrink the bitsets, including a bit-parallel `allDifferent`: five pattern vertices with only four target vertices between them kill the node with no search. Restarts are frequent and the nogoods are shallow, because a node

is a handful of bitset operations, and early guesses are the ones a heuristic gets wrong. Proof logs from the same group certify that kind of cut. Chapter 23 notes why a SAT solver struggles with the same pattern: a pigeonhole needs an exponential resolution proof.

The prover has the per-package version of that cut. `domain_meet` intersects slot sets, and `slots([])` is inconsistent before any candidate is tried. There is no count across packages, because slots are not a shared pool. `gcc:12` and `gcc:13` belong to `sys-devel/gcc` alone, the chosen version picks its own slot, and a `:=` rebuild moves the occupant after the choice. A collision is learned as a `cn_domain` and the affected subtree is retried. Timed restarts in the Glasgow style would repeat the lookahead problem from the previous section: a node here is a dependency model, not a bitset intersection. Chapter 23 makes the same comparison against pkgcore's occupancy table.

Fail-first variable ordering is used, frozen into a declaration. `ranking:tightness_classes/1` proves the tightest constraint first, so `selected_cn` locks early. Glasgow recomputes "smallest domain" at every node. The criterion list stays put for the whole search, which is why two runs against the same tree produce the same plan.

25.6 Package solvers

Almost no package manager runs clingo. The solvers in production are CDCL specialised to versions:

Solver	Used by	What it kept from the ASP line
libsolv	DNF, Zypper	MiniSat-style CDCL over a pool of concrete packages
resolvo	rattler, pixi	The same algorithm, lazy about metadata
PubGrub	Dart, uv, Poetry, SwiftPM	A learned incompatibility rendered in English. The authors cite Gebser, Kaminski, Kaufmann and Schaub, *Answer Set Solving in Practice*
mccs / CUDF	opam	MaxSAT, so a preference is an objective
clingo	Spack	The production package solver that stayed inside ASP

Spack's concretizer generates an encoding and lets clingo choose compilers, variants and providers. That pays off when the question is the best assignment under many soft preferences, and when the encoding of one spec is the whole program. Gentoo's question is the newest acceptable plan, in Portage's order, including the build order. A Gentoo atom is an awkward boolean variable: USE conditionals, slot operators, `:=` rebuilds and blockers all have to be interpreted, and the preferred solution comes from the order of search.

PubGrub is the closest of these. It looks for one solution, newest version first, and on failure it gives a reason a person can act on. A learned `cn_domain` is that incompatibility, stored as

a domain instead of as a clause. The printed split between positive assumptions (unmask, keyword, license, blocker) and negative ones (missing package, slot conflict, unsatisfiable USE) comes from the same requirement. PubGrub stops at a set of packages. Pass 2 here is a second proof, of `scheduled` and `available`, so the build order is not a separate graph walk over a finished set. See Chapter 13.

25.7 Current work

Recent papers have mostly moved on from the CDCL core. The active work sits in three places.

Grounding. Lazy grounding, body-decoupled grounding, compiling a rule into a propagator instead of instantiating it, and multi-shot grounding that reuses one ground program across queries. On a package tree the proof never builds the theory, which is the outcome that work is aiming at for arbitrary programs. The method here stays tied to a dependency atom that names its package.

Certificates. Checking that a set is an answer set is polynomial for a normal program. Checking that none exists is not, and disjunctive programs sit one level higher. ASP-QRAT (KR 2024) certifies both the consistent and the inconsistent case by a translation to quantified boolean formulas. VeriPB, from the same line as the Glasgow proof logs, is becoming a shared format for SAT, pseudo-Boolean solving, constraint programming and subgraph isomorphism, with a formally verified checker alongside it. The Proof AVL records why a plan was derived. An independent checker still cannot reject a wrong `domain_meet` or a wrong cycle-break. That checker is the piece still missing. The comparison with Portage in Chapter 23 does not need it.

Hybrids. Theory propagators, difference constraints, and ASP modulo SMT keep the stable-model loop and let a foreign theory post nogoods back into it. In clingo, preferences became weak constraints and priority levels. Ordered logic programs are no longer a research programme of their own. The priority levels here are the fallback tiers and the criterion list, evaluated during the proof rather than passed to a MaxSAT solver as an objective.

25.8 Summary

The comparison with Portage, pkgcore and Paludis is in Chapter 23. Most targets finish in one pass. A conflict leaves a narrower domain for the next attempt, as well as any mask that was recorded.

Against the solvers in this chapter, the useful points are these:

- A goal-directed proof at the scale of a repository never builds the ground program. Every constraint that can change the plan is attached to a selected package, or to an installed reverse edge.
- The learned nogood is a version domain. Narrowing it is how a conflict propagates.
- One preferred plan is the result. Other ways of satisfying the same choice are left unexplored, and packages the proof never mentions are simply absent.
- The build order is a second run of the same prover, under the planning rules.

A few of those mechanisms are local to this codebase, rather than a direct reading of Zeller or of clasp. The nogood is a version domain. When a pass has already finished, a USE-force conflict restarts only along that pass's trigger closure. Sub-slot rebuilds, `PDEPEND`, and soft preferences that would close a cycle are all proved, so nothing has to be patched onto the plan afterwards.

A few related problems are left to those other solvers. There is no grounding procedure for an arbitrary answer-set program. Negation is the stack test described under answer-set solvers, which is weaker than smodels' well-founded check. A slot collision is learned and retried; packages do not share a pool of slots the way an `allDifferent` constraint expects. The prover returns one plan. The Proof AVL is the explanation of that plan, and a separate checker cannot yet reject a bad `domain_meet` or a bad cycle-break.

Chapter 26

Testing and Regression

portage-ng uses multiple testing strategies: PLUnit tests for unit logic, overlay regression tests for end-to-end scenario validation, and merge-vs-emerge comparison for correctness measurement against Portage.

26.1 PLUnit tests

Standard SWI-Prolog unit tests, one file per subject under `Source/Test/Unit/` (loaded together by `Source/Test/unittest.pl`):

```
make test
```

These test individual predicates in isolation — version comparison, domain operations, context merging, EAPI parsing, etc.

26.2 Overlay regression tests

The overlay test suite (`make test-overlay`) runs 80 curated scenarios against a test overlay in `Repository/Overlay/`. Each scenario has a specific dependency story and expected behavior.

For onboarding, treat a subset as **policy specimens** (not only CI): see Policy by example and the policy cards.

26.2.1 Running

```
make test-overlay
```

Or from the interactive shell:

```
test:run(cases).
```

26.2.2 Test scenario anatomy

Each test under `Documentation/Tests/testNN/` contains:

- **`README.md`** — description of the dependency story and expected outcome
- **`testNN.svg`** — dependency graph visualization
- **Collapsible transcripts** — `emerge -vp` vs `portage-ng --pretend` output for comparison

26.2.3 Coverage areas

Area	Tests
Basic ordering / default version	01-02
Cycles (self, indirect, 3-way, PDEPEND)	03-08, 47, 61-64, 79
Missing dependencies	09-11
Keywords (stable vs unstable)	12
Version operators (`=`, `>=`, `~`, `<=`)	13, 55-56, 69-70, 80
USE conditionals	14-15
Choice groups (`^^`, `||`, `??`)	17-25
Blockers (strong/weak)	26-31, 60
REQUIRED_USE	32, 40
USE dependencies (`[flag]`, `[-flag]`, `=`)	33-39
Slots (`:*`, `:=`, sub-slot)	41-44
Conflicts (USE, slot, diamond)	45-46, 48-49, 51
USE merge (shared deps)	52-53
Virtuals	57-58
Installed / VDB operations	65, 73-77
PDEPEND	66, 79
BDEPEND / IDEPEND	67, 72
Multi-slot co-install	68
Fetch-only (`:run` proof + print filter)	71
Onlydeps	78

26.2.4 Failure testing

Test 58 is explicitly marked as an expected failure (XFAIL) via `test:xfail/2` — it exercises PROVIDE-based virtuals, deprecated in PMS; a documented limitation that will not be fixed.

26.3 Merge vs emerge comparison

The primary correctness metric is comparison against Portage's `emerge` output across the entire Portage tree. The comparison harness now lives in the tinderbox-ng repository, which drives both engines through identical sessions and analyses the resulting plan logs.

26.3.1 Running a comparison

Per-target compare (plan only, fresh sessions on both sides):

```
tinderbox-ng compare www-servers/apache
```

Whole-tree matrix run plus aggregate analysis:

```
tinderbox-ng new regress
tinderbox-ng exec regress -- \
  tinderbox-matrix resolver \
  /usr/local/share/tinderbox-ng/share/tinderbox-ng/manifest-1000.txt
tinderbox-ng analyze \
  --md5-cache /srv/tinderbox-ng/baseline/var/db/repos/gentoo/metadata/md5-cache
```

`tinderbox-ng analyze` feeds each `portage-ng.plan.log` / `emerge.plan.log` pair through `share/tinderbox-ng/compare-merge-emerge.py` (inside the tinderbox-ng repo) and writes `analysis.json` + `analysis.txt` into the matrix run directory.

26.3.2 Metrics

The comparison produces several accuracy metrics:

Metric	Formula	Meaning
CN	`100 * inter_cn / union_cn`	Category/Name match (ignoring version)
CN+V	`100 * inter_cnv / union_cnv`	Category/Name+Version match
CN+V+U	`100 * inter_cnvu / union_cnvu`	Full match including USE flags
Order%	`100 * (pairs - inversions) / pairs`	Ordering concordance

Additional counts (from `emerge_ok` pairs only):

- `#blockers` — total blocker assumptions
- `#cycle breaks` — total prover cycle-break assumptions
- `#domain assumptions` — total domain assumptions

26.3.3 Targeted comparison

For a single package, use `--target-regex` on `tinderbox-ng analyze`:

```
sudo tinderbox-ng analyze --target-regex '^sys-apps/portage-3.0.77-r3$'
```

Or run a one-off per-target compare directly:

```
tinderbox-ng compare sys-apps/portage
```

26.4 Bulk plan fingerprint comparison

`Source/Test/plancompare.pl` fingerprints the full pipeline (resolve + order) for every ebuild in a repository. Use it to verify that a resolver change produces identical plans before committing:

```
./Source/Application/Wrapper/portage-ng-dev --mode standalone --shell <<'PL'
load_files(portage('Source/Test/plancompare'), [if(true)]).
plancompare:run(portage, '/tmp/plan-compare.tsv').
halt.
PL
```

Compare two TSV files from before/after runs:

```
plancompare:diff('/tmp/before.tsv', '/tmp/after.tsv').
```

26.5 md5-cache extractor regression

`md5cache_validate/0,1` (in `Source/Test/md5cache.pl`) runs the standalone bash extractor at `Source/Domain/Gentoo/Ebuild/ebuild-depend.sh --batch` over every md5-cache entry in the configured Portage tree and diffs the produced metadata against the on-disk cache, key by key.

```
./Source/Application/Wrapper/portage-ng-dev --mode standalone --shell <<'PL'
load_files(portage('Source/Test/md5cache'), [if(true)]).
md5cache_validate([limit(50), verbose(true)]).
halt.
PL
```

Options: `repo(Atom)` (default `portage`), `limit(N)` (0 = all), `verbose(Bool)`, `out(Path)` (writes a Prolog-term report).

26.6 Further reading

- Chapter 2: Installation and Quick Start — `make test` commands
- Chapter 27: Performance and Profiling — `resolver:test_stats` for bulk testing
- Chapter 28: Contributing — development workflow with regression testing

Chapter 27

Performance and Profiling

portage-ng loads on the order of **32,000 ebuilds** into memory and reasons about their dependencies with **formal proof search**. That combination is easy to make slow: naive parsing, interpreted queries, imperative undo stacks, exponential backtracking, and repeated failed branches can each dominate runtime on their own. The design question is not "which single trick wins?" but **how we stack complementary strategies** so the whole pipeline stays responsive.

The answer is **the five pillars of portage-ng performance**: compiled knowledge (qcompiled cache), compile-time query expansion, persistent AVL structures for proof state, prescient proving that avoids redundant work, and incremental learning that narrows the search after failures. Together they explain why the tree can load with sub-second queries and why a full prove across all packages can finish in under a minute on a strong multi-core machine—while leaving room for profiling and targeted optimization.

This chapter walks those pillars in order, then covers **instrumentation** (the sampler), **bulk testing**, and a **pm-bench performance comparison** across emerge, pkgcore, Paludis, and portage-ng IPC.

27.1 Pillar 1: Compiled knowledge (qcompiled `.qlf` files)

The Portage tree is **not** parsed from scratch on every startup. During `--sync`, metadata is read and the knowledge base is written in a form that SWI-Prolog can **qcompile** into a binary load unit—`Knowledge/kb.qlf` (source facts live in `Knowledge/kb.raw`). The next time the application starts, it loads that **binary** representation instead of re-parsing large textual artifacts.

That is the **largest single speedup** in the system: startup drops from **tens of seconds** of parsing and assertion to **under a second** for the compiled cache, after which reasoning works directly over in-memory facts. Everything else in this chapter assumes that this first pillar is in place; without it, no amount of clever proving would feel fast enough.

Companion caches. `Knowledge/profile.qlf` (profile tree data, built by `--sync`) and `Knowledge/preference.qlf` (materialized preference state after `preference:init/0`, built on first startup and invalidated by `--sync` or input changes) further reduce startup work. After `kb.qlf` loads, `knowledgebase:load/0` also primes the JIT index on `cache:entry_metadata/4` for slotted profile-mask lookups so the first `preference:init/0` does not pay a multi-second penalty on atoms such as `dev-qt/qtimageformats:5`.

27.2 Pillar 2: Goal expansion macros

High-level queries in the knowledge layer are written for clarity; at **compile time** they are rewritten into **direct cache access**, so the runtime path never pays for meta-interpretation over generic search.

A module-local `query:goal_expansion/2` hook in `Source/Knowledge/query.pl` performs this rewrite (deliberately *not* `user:goal_expansion/2`, so only code compiled inside the `query` module is affected — portage-ng#59). It expands `search(Query, Repo://Id)` goals at compile time: `compile_query_list/3` / `compile_query_compound/3` translate each query term into direct indexed cache lookups such as `cache:ordered_entry/5` conjunctions.

The expanded code calls the indexed predicate **directly**. SWI-Prolog's **first-argument indexing** on `cache:entry/5` (and related entry predicates) makes those lookups **O(1) amortized** in typical use: the prover's inner loop sees plain deterministic cache reads, not a slow interpretive layer.

For how the knowledge base and query surface fit together, see Chapter 6: Knowledge Base and Cache.

27.3 Pillar 3: Persistent AVL trees

Proof search maintains large associative structures—proof literals, models, constraints, triggers—using `library(assoc)` **AVL trees**. Lookups and updates are **O(log n)**; for about **32,000** entries that is on the order of **fifteen comparisons** per operation, which is cheap enough to live in the inner loop of dependency proving.

The deeper win is **persistence**: AVL trees in Prolog are **immutable structures** threaded through the search. **Backtracking** automatically restores the previous tree without hand-written save/restore stacks or explicit undo logs—the kind of machinery imperative resolvers often maintain by hand. That keeps the prover's control flow simple while remaining safe under deep choicepoints.

Practical caveat: Proof and Model AVLs still **grow with proof size**. Algorithms should avoid **full traversals** when a more local structure suffices; the Triggers AVL (see the next pillar) exists partly so reverse lookups do not devolve into scanning the entire proof tree. That trade-off shows up again in practice when proof trees grow large.

27.4 Pillar 4: Prescient proving (avoiding backtracking)

Naive proof search can exhibit $\mathbf{O(2^n)}$ behaviour in the worst case: each wrong choice is explored and then undone by backtracking. portage-ng pushes hard in the other direction by **merging proof context** when the same literal is encountered again with **refined constraints**—via mechanisms such as **feature term unification**—so the system does not blindly re-prove from scratch every time the dependency graph revisits a head under slightly different assumptions.

In practice, for most real packages, that style of **prescient** handling yields **O(n) amortized** proof steps rather than exponential churn. The **Triggers AVL** complements this: it supports **efficient identification of affected heads** when something downstream changes, instead of linear scans over the whole proof.

The sampler's `ctx_union` **sampling** (documented later in this chapter) exists precisely to spot **hot merge paths**—a sign that context merging is working harder than it should and that some literals may still be reproved more often than necessary.

27.5 Pillar 5: Incremental learning (avoiding repeated failures)

When a proof attempt fails, portage-ng does not always forget what went wrong. **Learned constraints** from failed branches can **persist across reprove retries**, **narrowing domains** so the same conflict is not hit twice the same way. Together with a **reject set** that records candidates already ruled out, the prover avoids thrashing on the same dead ends.

That closes the loop with Chapter 8: The Prover: reprove and learning are part of the same story as performance. If retries explode without narrowing behaviour improving, runtime suffers.

27.6 Sampler module

The sampler (`Source/Application/Performance/sampler.pl`) is the main place to **measure** whether the pillars above are behaving as intended in production-like runs.

27.6.1 Hook performance

```
sampler:phase_walltime(-T)
```

Captures a wall-clock snapshot. The pipeline takes three snapshots — before resolving, between resolving and ordering, and after ordering.

```
sampler:phase_record(T0, T1, T2)
```

Computes and records the per-phase deltas (resolve ms, order ms) from the three snapshots for later retrieval.

27.6.2 Test statistics

```
resolver:test_stats(Repository)
resolver:test_stats_pkgs(Repository, PackageList)
```

Run the resolver across all packages (or a specific list) in a repository and collect aggregate statistics:

- Totals: entries processed, proved, failed
- Share of entries with domain assumptions and with cycle breaks (as percentages)
- Failure and assumption-type breakdowns
- Slowest entries and packages

27.6.3 Feature term unification sampling

The sampler tracks feature term unification operations to identify hot paths in context merging. Excessive merges can indicate redundant re-proving.

27.7 Bulk testing workflow

The standard performance testing workflow uses the `--shell` here-doc pattern:

```
./Source/Application/Wrapper/portage-ng-dev --mode standalone --shell --timeout 60
<<'PL'
resolver:test_stats(portage).
halt.
PL
```

For specific packages:

```
./Source/Application/Wrapper/portage-ng-dev --mode standalone --shell <<'PL'
resolver:test_stats_pkgs(portage, ['kde-apps'-'kde-apps-meta']).
halt.
PL
```

27.8 Performance comparison: package managers (pm-bench)

The numbers below are **process wall-clock** times for a cold `--pretend` / resolve-only invocation of each tool — how long it takes to produce a merge plan, not how long builds take. They were measured on `vm-linux.local` with `Source/Application/Wrapper/pm-bench` against an identical Portage tree, VDB, and `/etc/portage` for every engine.

Tool	Invocation
emerge	`emerge -vp <pkg>`
pmerge	`pmerge -vp <pkg>` (pkgcore)
cave	`cave resolve --lazy <pkg>` (Paludis)
ng-ipc	`portage-ng-dev --mode ipc --pretend <pkg>` (ultralight SWI `ipclient.pl`)
ng-ipc-cpp	`ipcclient --mode ipc --pretend <pkg>` (native C++ client)

portage-ng IPC uses a **warm daemon** (knowledge base already loaded); the timed sample is client connect + prove + print. Standalone portage-ng (full process boot + `kb.qlf` load each time) is much slower and is not included in this table.

27.8.1 Packages where emerge produced a plan

16,313 packages where `emerge -vp` exited 0 (an actual plan). Median speedup is `emerge_median / tool_median`. Total is the sum of per-package wall times (serial equivalent).

Tool	Median	Average	Total	Median speedup	Total speedup
emerge	1,234 ms	1,345 ms	6.09 hours	1.0×	1.0×
cave	520 ms	1,372 ms	6.22 hours	2.4×	1.0×
pmerge	245 ms	282 ms	1.28 hours	5.0×	4.8×
ng-ipc	82 ms	119 ms	32.4 minutes	**15.1×**	**11.3×**
ng-ipc-cpp	55 ms	92 ms	25.0 minutes	**22.4×**	**14.6×**

The C++ IPC client is about **27 ms** faster than the SWI `ipclient` at the median (same daemon; the gap is client process boot). Both IPC tools finished every package in this set with exit code 0. Cave's average/total are inflated by hard timeouts (120 s cap in this run).

27.8.2 All packages in the manifest

Across all **19,430** packages in the pm-bench manifest (including those where emerge exited non-zero):

Tool	Median	Average	Total	Median speedup
emerge	1,240 ms	1,437 ms	7.76 hours	1.0×
cave	544 ms	1,891 ms	10.21 hours	2.3×
pmerge	247 ms	286 ms	1.55 hours	5.0×
ng-ipc	85 ms	161 ms	52.0 minutes	**14.6×**
ng-ipc-cpp	58 ms	133 ms	43.2 minutes	**21.4×**

27.8.3 Why portage-ng IPC is faster

The gap is not primarily "Prolog vs Python." It comes from architectural differences that compound across thousands of packages:

Factor	Traditional PMs	portage-ng (daemon + IPC)
Startup cost	Interpreter + imports + metadata load each invoke	KB loaded once in the daemon; thin client (~3–30 ms)
Graph construction	Build full graph, then check for conflicts	Single-pass proof — no separate graph phase
Conflict recovery	Discard / restart large parts of the search	Retry the affected subtree with learned constraints
Repeated queries	Each pretend starts cold	In-memory facts persist across IPC requests
Parallelism	Sequential graph walk	Ordering pass identifies parallel waves

The largest single factor for interactive use is the **resident daemon** plus **qcompiled cache** (Pillar 1): once loaded, ebuilds are indexed Prolog facts and queries hit first-argument indexing

directly. The second factor is **single-pass proving** (Pillar 4): for over 99% of packages, portage-ng needs no backtracking at all.

Chapter 28

Contributing

This chapter covers the development workflow, coding conventions, and testing practices for contributing to portage-ng.

28.1 Reporting issues

Please report bugs, feature requests, and other issues through GitHub:

https://github.com/pvdabeel/portage-ng/issues

Do not use email or other informal channels for issue tracking.

28.2 Development workflow

1. **Start from clean committed state.** Always begin development with no uncommitted changes.
2. **Make changes** using the project wrapper for testing:

```
./Source/Application/Wrapper/portage-ng-dev --mode standalone --pretend <target>
```

3. **Run tests** to verify correctness:

```
make test            # PLUnit tests
make test-overlay    # Overlay regression tests
```

4. **Run compare analysis** to detect regressions. The compare harness lives in the tinderbox-ng repository and generates its own plan logs in fresh sessions (the legacy `--graph` + `.merge` regeneration loop is no longer part of this workflow):

```
# Whole-tree matrix run (on the tinderbox host):
sudo tinderbox-ng compare-matrix
sudo tinderbox-ng analyze

# Or a quick per-target compare while iterating locally:
tinderbox-ng compare <category>/<package>
```

`tinderbox-ng analyze` produces `analysis.json` + `analysis.txt` in the matrix run directory, replacing the old `compare-<date>-<hash>.json.gz` snapshots.

5. **Review the comparison table** for regressions in CN, CN+V, CN+V+U match percentages, ordering concordance, and assumption counts.
6. **Commit** when regression-free.

28.3 How to run

28.3.1 Dev wrapper

Always use the dev wrapper for testing — never run ad-hoc `swipl -g "..."` snippets, as they miss required operator definitions, libraries, and module load order:

```
./Source/Application/Wrapper/portage-ng-dev --mode standalone --pretend <target>
./Source/Application/Wrapper/portage-ng-dev --mode standalone --shell
```

28.3.2 Scripted sessions (here-doc pattern)

For reproducible, non-interactive debugging:

```
./Source/Application/Wrapper/portage-ng-dev --mode standalone --shell --timeout 60
<<'PL'
resolver:test_stats(portage).
halt.
PL
```

28.3.3 CI mode

For automated checks:

```
./Source/Application/Wrapper/portage-ng-dev --mode standalone --ci --pretend
<target>
echo $?  # 0 = no assumptions, 1 = cycle breaks, 2 = domain assumptions
```

Always include `--pretend` to avoid mutating local state.

28.4 Source file documentation style

Every `.pl` source file follows a strict layout. Use `Source/Application/System/bonjour.pl` as the canonical reference.

28.4.1 File header

```
/*
  Author:   Pieter Van den Abeele
  E-mail:   pvdabeel@mac.com
  Copyright (c) 2005-2026, Pieter Van den Abeele
```

```
  Distributed under the terms of the LICENSE file in the root directory of this
  project.
*/
```

28.4.2 Module documentation (PlDoc)

```
/** <module> MODULE_NAME_UPPERCASE
Short one-line description.

Optional longer description.
*/
```

Module name in the `<module>` tag is UPPERCASE.

28.4.3 Module declaration

```
:- module(modulename, []).
```

28.4.4 Chapter header (one per file)

```
% ============================================================================
% MODULE_NAME_UPPERCASE declarations
% ============================================================================
```

Exactly one `=====` chapter per file, immediately after `:- module`.

28.4.5 Section headers

```
% ----------------------------------------------------------------------------
% Section title
% ----------------------------------------------------------------------------
```

All subsequent sections use `-----` dashes.

28.4.6 Predicate documentation

```
%! module:predicate_name(+Arg1, -Arg2)
%
% Short description of what the predicate does.

module:predicate_name(Arg1, Arg2) :-
  body.
```

28.4.7 Spacing rules

Element	Blank lines after
File header `*/`	1
PlDoc module comment `*/`	1
`:- module(...)` declaration	1
`=====` chapter header	1
`-----` section header	1
Predicate doc + last clause	2
Between clauses of same predicate	0
End of file	0 (no trailing blank line)

28.5 Naming conventions

- Source filenames must NOT contain hyphens (`-`) or underscores (`_`). Use concatenated lowercase words: `knowledgebase.pl`, not `knowledge_base.pl`.
- Exceptions (grandfathered, do not add new ones): `portage-ng.pl` (project entry point / name); `binpkg_exec.pl`, `binpkg_index.pl`, `binpkg_extract.pl`, `ebuild_exec.pl` and `missing_provider.pl` (underscore-named for readability of their prefixes; module names match the filenames). Host-local templates under `Source/Config/Private/` are configuration, not source modules, and are also exempt.
- Prolog module names follow the same rule: `:- module(gentoo, []).`
- Subdirectory names under `Source/` may use CamelCase: `Application/`, `Domain/`, `Config/`, `Pipeline/`.

28.6 Comment guidelines

Do not add comments that just narrate what the code does. Comments should only explain non-obvious intent, trade-offs, or constraints. Avoid:

```
% Get the version     ← redundant
version:get(V).
```

Prefer:

```
% Suffix rank maps PMS suffix ordering to integers for compare/3
suffix_rank('_alpha', 1).
```

28.7 Compare tooling

Regression tooling is hosted in two places:

- **Merge-vs-emerge plan comparison** — driven by tinderbox-ng via `tinderbox-ng compare` / `tinderbox-ng compare-matrix` / `tinderbox-ng analyze`. The underlying Python script lives at `share/tinderbox-ng/compare-merge-emerge.py` in that repository and is invoked automatically by `tinderbox-ng analyze`. Outputs are `analysis.json` + `analysis.txt` in the matrix run directory.
- **md5-cache extractor regression** — `md5cache_validate/0,1` in `Source/Test/md5cache.pl` (re-extracts metadata via `Source/Domain/Gentoo/Ebuild/ebuild-depend.sh --batch` and diffs the result key by key against the on-disk md5-cache).

Do not create ad-hoc compare scripts outside these two locations.

28.8 Further reading

- Chapter 26: Testing and Regression — testing methodology
- Chapter 27: Performance and Profiling — performance testing
- Chapter 2: Installation and Quick Start — build and run instructions

Chapter 29

Closing Thoughts

This book opened with a question that has quietly shaped every chapter since: as software systems grow more complex, can the tools that manage them keep up?

portage-ng's answer is to treat package management not as a logistics problem — downloading and unpacking files — but as a **reasoning problem**. Dependencies become proof obligations. Configuration choices become constraints. Conflicts become learning opportunities that narrow the search space for the next attempt. The result is a system that can walk tens of thousands of packages, resolve their interdependencies, and produce a buildable plan — often in a single pass.

29.1 What we covered

The book traced this idea from concept to implementation:

- **Part I** set the stage: the growing complexity of source-based distributions, the limits of imperative solvers, and how Prolog's backtracking and unification provide a natural fit for dependency reasoning.
- **Part II** unpacked the architecture: how the knowledge base stores and indexes package metadata; how the EAPI grammar parses dependency specifications; how the prover searches for consistent models; how assumptions, constraint learning, and version domains handle conflicts; how rules encode Gentoo's domain logic; and how a second proving pass over planning laws turns a proof into a concrete build order.
- **Part III** covered the features built on top of that foundation: the command-line interface, build execution, semantic search with LLM integration, distributed proving across clusters, and upstream bug tracking.
- **Part IV** explored the theoretical underpinnings: contextual logic programming, feature unification, and the comparison with other resolvers — showing how portage-ng's approach relates to Portage's progressive relaxation, pkgcore's frame-stack backtracking, Paludis's constraint accumulation, and the solver literature from feature logic and ordered logic through answer-set solving to the package solvers that followed.
- **Part V** described the practical side of development: testing strategies, performance profiling, and contribution guidelines.

29.2 Design principles worth remembering

A few recurring themes run through the design and are worth calling out explicitly:

- **Declarative over imperative.** The prover does not maintain mutable state that must be carefully unwound on failure. AVL trees are persistent; backtracking is automatic; learned constraints accumulate naturally. This makes the system easier to reason about and extend.
- **Single-pass where possible, learning where not.** Most packages resolve in one pass. When conflicts arise, the system learns from them — narrowing version domains, recording rejects — so the next attempt is better informed. This is fundamentally different from starting over with a blank slate on each retry.
- **Separation of concerns.** The prover knows nothing about Gentoo. The rules layer knows nothing about proof search. The planning laws know nothing about dependency types. Each layer has a clean interface, and domain-specific knowledge stays in domain-specific modules.
- **Transparency.** Assumptions are not silent failures — they are classified, explained, and reported. The explainer can trace any package's presence in the plan back through the proof to the original dependency that required it. When something goes wrong, the system tells you why.

29.3 Looking ahead

portage-ng is a living project. Several directions remain open for exploration:

- **Broader platform support.** The reasoning engine is not tied to Gentoo — any system that can express its dependencies as structured rules could use the same prover and ordering laws.
- **Richer learning.** The current constraint learning mechanism handles version domains and parent narrowing. More sophisticated strategies — learning across multiple proof runs, or sharing learned constraints between cluster workers — could further reduce proving time.
- **Tighter LLM integration.** The explainer already bridges proof traces and natural language. Future work could let users ask higher-level questions ("why is my build slow?", "what changed since last week?") and receive answers grounded in the formal proof structure.
- **Binary package support.** As Gentoo's binary package infrastructure matures, portage-ng could reason about mixed source/binary strategies — deciding which packages to build from source and which to install from pre-built archives.

29.4 Thank you

If you have read this far, you have a thorough understanding of how portage-ng works and why it was built this way. Whether you are using it to manage a Gentoo system, studying it as an example of applied logic programming, or contributing to its development — thank you for your interest, and welcome to the project.