Towards ASP-Based Composition of Scientific Workflows

Nikolas Bertrand[1,*,†], Charlotte Grunert[1,‡], Mario Frank[1,‡] and Anna-Lena Lamprecht[1,‡]

[1]University of Potsdam, Department of Computer Science, An der Bahn 2, 14476 Potsdam, Germany

Abstract

Research across all disciplines generates increasingly large and complex data that require data analysis workflows (DAWs) for processing and interpretation. Constructing these workflows manually can be challenging, especially given the growing numbers of computational tools and libraries that are available to implement the different steps in the workflows. The Automated Pipeline Explorer (APE) addresses this workflow composition challenge by synthesizing workflow candidates for a given specification, based on a semantic domain model that provides an ontology of terms and functional characterizations of the available workflow components. APE has been used for workflow composition in different domains, including bioinformatics, geosciences, and computational materials science. However, the current implementation of APE, which constructs a temporal finite automaton, faces extensibility and scalability issues that hinder its application to comprehensive domains. In this paper, we present an ASP-based architecture for automated workflow composition, framing the synthesis problem as a search for stable models that represent valid workflow candidates. Our approach uses multi-shot incremental solving as a basis for future interactive configuration, combined with different optimizations motivated by domain-specific features. Our evaluation with diverse benchmarks shows grounding and memory improvements on small and medium domains. The 1:N decompression variant matches APE's workflow space on the completed benchmarks, but does not scale to the densest full `bio.tools` case; the compressed default and K-cluster variants trade completeness for smaller encodings. On the largest full `bio.tools` cases, the optimized ASP backend remains memory intensive, exposing the next scalability bottleneck.

Keywords

Scientific Workflows, Data Analysis Workflows, Answer Set Programming,

CEUR-WS.org/Vol-4311/ASPOCP_regular_09.pdf

1. Introduction

The increasing volumes of data generated by research across all disciplines require increasingly complex computational analysis processes. These processes are often implemented as *scientific workflows* (also called data-analysis workflows or computational pipelines) [1]. A scientific workflow describes how data move through a set of usually preexisting computational tools: the output produced by one step becomes input to subsequent steps, collectively transforming initial data into a desired scientific result [2].

The working scientist, such as a bioinformatician analyzing a proteomics run, a geoscientist correlating event catalogs, or a materials researcher screening candidate structures, is typically less concerned with the workflow itself than with the scientific result it enables. They have initial data, a desired result, and preferences about how the analysis should be conducted (e.g., which tools they trust and which parameter regimes they want excluded). These ideas about the data-analysis process provide scaffolding for any intelligent system assisting with the workflow composition.

The *Automated Pipeline Explorer (APE)* [3, 4] has been developed for assisting in exactly this situation. Given a semantic domain model that describes the available computational tools with their names, operations, inputs, outputs, and data-type relationships (using terms from a domain ontology), and a workflow specification consisting of initial inputs, desired outputs, and possible additional constraints

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026), co-located with the Federated Logic Conference (FLoC 2026), July 18–19 and 24–25, 2026, Lisbon, Portugal.

*Corresponding author.

† Main author.

‡ These authors contributed equally

nikolas.bertrand@uni-potsdam.de (N. Bertrand); charlotte.grunert@uni-potsdam.de (C. Grunert); mario.frank@uni-potsdam.de (M. Frank); anna-lena.lamprecht@uni-potsdam.de (A. Lamprecht)

0000-0001-7259-2111 (N. Bertrand); 0009-0007-1871-5409 (C. Grunert); 0000-0001-8888-7475 (M. Frank); 0000-0003-1953-5606 (A. Lamprecht)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-03

expressed in the temporal logic SLTL^x [5], APE searches for valid tool compositions and enumerates workflow candidates, up to a user-specified maximum number of solutions. APE has been successfully used in different domains, such as proteomics [6], GIS [7], and computational materials science [8], and is a key component of the Workflomics platform for integrated composition and benchmarking of scientific workflows [9, 10]. A user-facing limitation of the current implementation of APE, which uses SAT-based finite-automaton construction, is iteration speed. We have observed that workflow composition in practice is often an iterative exploration process rather than a one-shot query with a fully specified configuration. Typically, the scientist would run a composition request, scan the top candidates, decide that one tool should be excluded or a constraint relaxed, and re-run. When the domain model is small, this loop is fast and comfortable. When the domain model becomes larger (for example using the full `bio.tools` registry with more than 32,000 tools [11]), APE's SAT-based encoding can spend so long on each invocation that the iteration loop breaks down; the user either waits minutes between attempts or falls back to manually narrowing the registry before each run. This bottleneck arises because APE's current architecture does not support caching closely related query results, so even minor configuration changes require complete recomputation. *Answer Set Programming (ASP)* [12], with its modeling primitives and solvers such as `clingo` [13], offers a basis for improving this iteration loop. *Multi-shot incremental solving* in particular maps naturally onto the user's mental model: each iteration step corresponds to one extra tool in the workflow, and the solver interface can in principle retain grounded program parts while the active horizon, goal set, or constraint list changes.

In this paper we present an ASP-based architecture for automated workflow composition. It is built around multi-shot incremental solving and designed as a foundation for future interactive edit-and-rerun workflows. To handle the large domain models specifically, we introduce an optimized compressed-candidate backend that shifts the compatibility and bindability reasoning out of the ASP grounder and into a Python pre-computation layer; the runtime then operates on a flat candidate set that can fit within `clasp`'s identifier budget [14]. From the user's perspective this means that future integrations can expose solver-state reuse across related composition requests. In this paper, however, we evaluate cold-start single-query runs against APE across several benchmark families, focusing on semantic coverage and the grounding, solving, and memory costs of the ASP backend.

The remainder of the paper is structured as follows. Section 2 introduces our ASP-based approach to workflow composition and its implementation. Section 3 presents the benchmarking study and discusses its results. Section 4 positions our study within related work, particularly in the ASP community. Section 5 concludes the paper. The implementation and benchmark code are available under an Apache License 2.0 on Zenodo [15].

2. Approach: ASP-Based Workflow Synthesis

We address the *workflow synthesis problem*: generating viable scientific workflow candidates from a domain model containing a workflow configuration (specifying inputs, goals, and workflow length), tool annotations, constraints, and an ontology that defines the hierarchy of data types and tool operations. In this section, we introduce our ASP-based approach to workflow synthesis, *clindaws*, named after its use of clingo [13] for composing data-analysis workflows. Our primary goal is semantic parity with APE: given the same domain model and workflow specification, `clindaws` should generate the same workflow candidates.

As Figure 1 illustrates, APE follows an iterative approach. Given a run configuration containing, among other settings, a maximum solution count, workflow length, and runtime, APE attempts to find workflows that satisfy the constraints. The process is initiated by encoding the constraints, specified in a JSON file, as SLTL^x expressions. In each step `i`, solutions of length `i` are generated. This is done by encoding solution candidates as automatons, transforming them together with the SLTL^x constraints into a CNF encoding and delegating the SAT problem to MiniSAT [16] (synthesis execution).

Each valid workflow is then added to the overall result set. The iterative search stops as soon as the maximum solution count, workflow length or runtime was exceeded in the previous iteration.

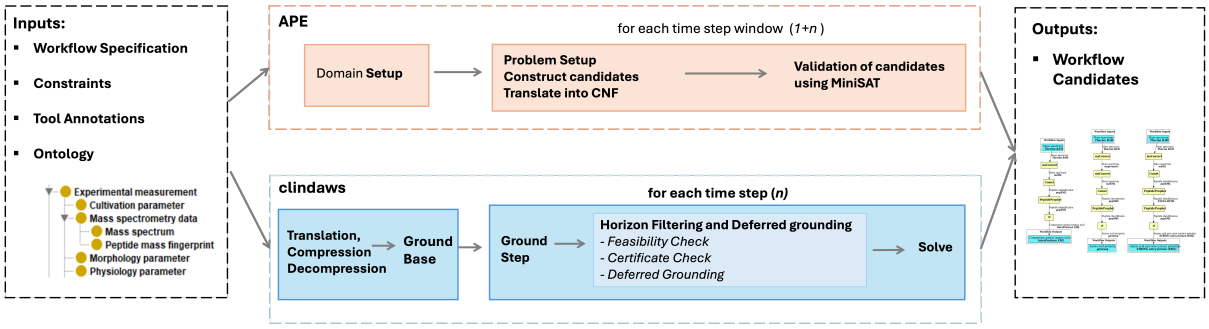


Figure 1: Workflow synthesis with APE and clindaws. Decompression in the clindaws pipeline is variant-specific (skipped by default; applied only by the K-cluster and 1:N variants).

Our `clindaws` approach to workflow synthesis evolved through three encoding generations: an initial single-shot encoding, a plain multi-shot encoding, and the optimized multi-shot backend on which this paper focuses. The single-shot variant grounds the full program in one pass. The plain multi-shot variant uses clingo's `base/init/step(t)/check(t)` skeleton and improves time to first solution, but still reaches `clasp`'s atom-identifier limit on dense domains. The optimized backend summarized in Figure 1 moves the tool-signature and ontology-dimension cross product into Python precomputation and emits compressed-candidate facts, so per-step program size depends on distinct candidates rather than the full tool-ontology cross product.

2.1. Domain Model

The domain model combines three ingredients: an ontology, tool annotations, optional domain-specific constraints. The ontology provides a hierarchical taxonomy of data types and operations. Tool compatibility is derived by closure over this hierarchy, so producing a subtype also provides its super types, and a tool requiring a super type may consume any compatible subtype. In addition to the domain model, the user must provide a workflow specification that defines the synthesis problem. It includes typed inputs and goal outputs, along with search controls such as the horizon, solution cap, timeout, and technical configuration. Users may also provide an optional constraint file containing custom constraints. Finally, the specification can define general workflow requirements, for example, whether all inputs must be used or whether using any subset is permitted.

Tool annotations describe concrete implementations as typed transformations over ontology terms: each annotation declares a `label`, one or more `taxonomyOperations`, and typed input/output ports. Each port represents an alternative application of the tool; during synthesis, clingo may select a port when all of its required inputs are available. An annotation may also specify an executable function. Listing 1 shows an example.

```
{"label": "create_structure_slab", "id": "create_structure_slab",
"taxonomyOperations": ["CreateStructureSlab"],
"inputs": [{ "Type": ["Project"] },
           { "Type": ["Material"], "Format": ["String"] }],
"outputs": [{ "Type": ["Slab"],
              "Format": ["pyiron_atomistics.atomistics.structure.atoms.Atoms"]}],
"implementation": { "code": "structure = pr.create_surface(Material,
                    surface_type='fcc111', size=(3, 4, 4), vacuum=10,
                    orthogonal=True)\n" }},
```

Listing 1: Example tool annotation taken from the computational materials science domain.

Additional workflow restrictions are expressed as domain-specific constraints. These may be provided either as APE-style templates or native ASP templates, but both are translated into a uniform set of `constraint_*` facts. Selectors may refer to concrete tools or abstract ontology classes; taxonomy-

aware matching is used by default, with exact matching available when required. The emitted constraint atoms are summarized in Table 1.

Table 1
Constraint atom templates usable by the translator. *Source*: APE, clindaws, or Both. The clindaws only constraints are available and used through custom SLTL^x based constraints.

Atom	Semantics	Source	Atom	Semantics	Source
`use_m(A)`	A appears	Both	`itn_m(A, B)`	A forbids later B	Both
`nuse_m(A)`	A excluded	Both	`depend_m(A, B)`	A requires earlier B	Both
`first_m(A)`	A only at step 1	clindaws	`next_m(A, B)`	A requires next-step B	Both
			`prev_m(A, B)`	A requires previous-step B	Both
`at_step(A, t)`	A at step t	clindaws	`used_iff_used(A, B)`	A iff B	clindaws
`max_uses(A, n)`	at most n uses of A	clindaws	`mutex_tools(A, B)`	not both A and B	clindaws
`not_consecutive(A)`	no adjacent A	clindaws	`connected_op(A, B)`	later B consumes earlier A	both
`unique_inputs(A)`	no duplicated input artifact for A	Both	`operation_input(A, X)`	A binds data selector X	both
`ite_m(A, B)`	A implies later B	Both	`use_t(X)`	some artifact carries X	clindaws

2.2. Translation of the Domain Model

The domain model can be translated into ASP facts in two ways. The **direct translation**, used by the single-shot and plain multi-shot backend, emits ontology, tool, workflow-input, goal, and configuration facts without prior computation. The grounder therefore receives the raw taxonomy and port declarations and derives port eligibility and compatibility relations itself. Although this keeps the translation simple, large domain models can cause the grounding phase to derive a substantial amount of atoms, potentially reaching `clasp`'s internal identifier limit. The optimized backend therefore uses a **compressed-candidate translation**. Expensive compatibility and bindability checks are moved into a pre-computation layer before clingo sees the program. The translator expands tool annotations into candidate records, prunes candidates that cannot participate in a workflow from the configured inputs to the goals, factors recurring input requirements into signature profiles, and emits the remaining candidate space as compact `dynamic_*` facts. These facts are the input to the optimized multi-shot backend described in the next subsection. Output ports with identical value spaces are merged to reduce the number of facts passed to the grounder. This compression is sound when all consumers of the merged port can be satisfied by the same value assignment. If consumers require incompatible assignments, the merged representation would lose workflows because the solver chooses values per (*Candidate pair*, *Port*). Selective decompression restores source-port structure where such divergence occurs. `clindaws` exposes three optimized variants around this choice. The default variant keeps merged ports intact and emits the fewest facts, but it can lose workflows that require different values on source ports that were merged.

The **K-cluster** variant is a compact approximation. For a divergent merged port, the translator groups consumers into a small number of compatibility classes by intersecting their per-dimension requirements, then replaces the merged port with one sub-port per class. This keeps the bindability graph small and avoids the full source-port expansion, but it is complete only when the chosen class split can express the source-port allocation used by an APE workflow. If a workflow needs a finer or uneven allocation across the original source-ports, K-cluster can still lose that workflow.

The **1:N** variant is the fidelity-preserving alternative for benchmarks where the expanded representation still fits the grounder. It recreates one singleton sub-port for each original source index and makes each relevant divergent consumer bindable to all of them; incompatible bindings are still rejected later by the check. This matches APE's workflow space on completed runs, but increases the number of output ports, bindable pairs, and support facts roughly with the original multiplicity of the merged port. On the densest full `bio.tools` case this expansion exceeds `clasp`'s identifier budget, so the optimized backend trades precision against grounding size.

2.3. Optimized Multi-Shot Backend

The optimized multi-shot backend follows the standard incremental scheme. It grounds the `base` program once and then iterates over horizons `t` by grounding the corresponding `step(t)` and `check(t)` parts over the compressed-candidate facts emitted by the translator. As shown in Algorithm 1, each horizon first grounds `step_initial(t)` (or `step(t)` for $t > 1$) and `constraint_step(t)`, then runs a translator-side structural feasibility precheck against the precomputed smart-expansion metadata in F_{opt}. If the horizon is feasible, the encoding grounds and solves `certificate_check(t)` as an exact-horizon goal- and must-run-coverage certificate, and only then grounds the deferred `step_query(t)`, `check(t)`, and `check_usage(t)` parts. The loop continues until the workflow specification's maximum length or maximum number of workflow candidates is reached.

Algorithm 1 Optimized Multi-Shot Runtime Execution

Require: Fact bundle F_{opt}, max horizon $H_{\max}$, solution cap S
1: ctl.add(base, F_{opt}); ctl.ground([base]); $\mathcal{W} \leftarrow \emptyset$
2: **for** $t = 1, \ldots, H_{\max}$ **do**
3: Ground `step_initial`(t) if $t = 1$, else `step`(t); ground `constraint_step`(t)
4: **if** not FeasibilityPrecheck(t, F_{opt}.backend_stats) **then continue** ▷ Python-side structural screen
5: ctl.ground([`certificate_check`(t)])
6: **if** not ctl.solve(`certificate_check`(t)) **then continue** ▷ exact-horizon goal + must-run coverage
7: ctl.ground([`step_query`(t), `check`(t), `check_usage`(t)]) ▷ deferred goal validation
8: $A_t \leftarrow \{\texttt{query}(t') \mapsto \bot : t' < t\} \cup \{\texttt{query}(t) \mapsto \top\}$
9: **for** each model $m \in ctl$.solve(A_t) **do**
10: $\mathcal{W} \leftarrow \mathcal{W} \cup \{$Canonicalize($m$)$\}$; add blocking clause for m
11: **if** $|\mathcal{W}| \geq S$ **then break**
12: **end if**
13: **end for**
14: **end for**
15: **return** $\mathcal{W}$

Ground base: ontology closure and goal-relevant candidates

The `base` program contains time-invariant domain facts and derived predicates, while `step(t)` adds the per-horizon generator and is grounded once for each `t`. The `check(t)` part contains the volatile goal-satisfaction integrity constraints that must hold at the query horizon. Instead of introducing the active horizon via a clingo `#external`, the encoding uses a one-element choice rule and an implication. This allows the Python control layer to activate horizon `t` purely through solve-time assumptions, without re-grounding. At each solve call, `clindaws` passes $A_t = \{\texttt{query_assumption}(t') \mapsto \texttt{False} \mid t' < t\} \cup \{\texttt{query_assumption}(t) \mapsto \texttt{True}\}$ as assumptions. As a result, all `query(t)`-guarded rules in already grounded `check` parts collapse except those of the current horizon. A similar assumption-driven scheme without externals or horizon-specific re-grounding was used in [17]. The contribution here is its application to workflow synthesis together with the `clindaws`-specific ordering of the Python structural precheck and exact-horizon certificate before the deferred `step_query(t)`, `check(t)`, and `check_usage(t)` grounding. The translator emits only taxonomy edges as ontological facts; all transitive reasoning is carried out inside the encoding by recursive ASP rules. Thus, the `ancestor` closure shown in Listing 2, terminal-class detection, and the directional `compatible` relation used by the goal-matching rules are all defined in the `base` program.

```
ancestor(Child, Parent)   :- taxonomy(_, _,(_,Child, Parent)).
ancestor(Child, Ancestor) :- ancestor(Child, Mid), ancestor(Mid, Ancestor).
is_terminal(V) :- taxonomy(_, _, (_, V, _)), not taxonomy(_,_,(_,_,V)).
compatible(V, V) :- is_terminal(V).
compatible(V, Ancestor) :- is_terminal(V), ancestor(V, Ancestor).
```

Listing 2: Transitive ontology closure and compatibility (`base.lp`)

The `is_terminal` rule uses stratified negation-as-failure over the taxonomy relation to identify ontology leaves. The `compatible` relation is intentionally directional: a terminal value V is compatible

with itself and with every ancestor of V, but never with a descendant. This matches goal-matching semantics, where a tool emitting a subtype still satisfies a goal stated at a supertype, and keeps the goal checks as pure conjunctive queries without explicit ontology traversal. The `base` program also defines a recursive predicate that pre-filters the candidate space to candidates whose output ports can satisfy every synthesis goal. This filter is used by the horizon-1 selector in `step_initial(t)` when exact-horizon mode is active: rather than ranging over every (*tool*, *candidate*) pair, the first step then ranges only over candidates already known to be goal-relevant, as shown in Listing 3. The general per-step selector at later horizons ranges over the wider `dynamic_candidate_allowed_at_step` set. The facts `dynamic_candidate_output_singleton` and `dynamic_candidate_output_choice_value`, emitted by the translator, enumerate either the unique output value of a port or the retained choice values after output-choice compression.

```
dynamic_candidate_output_goal_match(Candidate, Port, GoalID, Category) :-
    goal_output(GoalID, GoalV, Category), compatible(ActualV, GoalV),
    dynamic_candidate_output_singleton(Candidate, Port, ActualV, Category).
dynamic_candidate_goal_satisfies(Candidate, GoalID) :-
    dynamic_candidate_output_port(Candidate, Port),
    dynamic_candidate_output_goal_satisfies(Candidate, Port, GoalID ).
dynamic_initial_goal_candidate(Candidate) :- dynamic_tool_candidate(Candidate,_),
    dynamic_candidate_goal_satisfies(Candidate, GoalID) : goal_id(GoalID),
    not dynamic_initial_goal_candidate_output_mismatch(Candidate).
```

Listing 3: Goal-relevant initial candidates (`base.lp`)

Per Horizon Grounding

At each horizon, the optimized backend selects exactly one tool and one of its admissible candidates. Unlike the direct multi-shot encoding, this choice ranges only over precomputed candidate identifiers, so the grounder never expands per-tool ontology joins at solve time.

```
1 { occurs(t, run(Tool)) : dynamic_step_allowed_tool(Tool, t) } 1.
1 { occurs(t, use_dynamic_candidate(Tool, Candidate)) :
    dynamic_tool_candidate(Candidate, Tool),
    dynamic_candidate_allowed_at_step(Candidate, t) } 1 :- occurs(t, run(Tool)).
```

Listing 4: Per-step candidate selection (`step.lp`)

At horizon 1, `step_initial(t)` applies a narrower prefilter than the general `step(t)` rule. Depending on the active mode flags, the first step ranges either over tools that can satisfy a goal directly, over tools that support a goal at the user-pinned exact horizon, or over the default `dynamic_step_allowed_tool` set. These cases are mutually exclusive, so only one selector is grounded. The earliest-feasible-step analysis provides `dynamic_candidate_allowed_at_step`, and once a candidate is chosen, each of its input ports is bound either to a workflow input via `eligible` or to a previously produced output via `produced_bind_option`. Both relations are precomputed in Python, so the grounder never expands the full cross product of current consumer ports against all earlier outputs. Rather than materializing those bindings in one large join, the optimized backend derives `produced_bind_option` through a short chain of auxiliary predicates keyed by support class and producer candidate:

```
selected_dynamic_candidate(t, Tool, Candidate) :-
    occurs(t, use_dynamic_candidate(Tool, Candidate)).
selected_candidate_support_class(t,Candidate,Port,SupportClassId) :-
    selected_dynamic_candidate(t,_,Candidate),
    dynamic_candidate_input_port(Candidate, Port),
    optimized_candidate_input_support_class(Candidate,Port,SupportClassId).
selected_candidate_bindable_producer_port(t, Candidate, Port, ProducerCandidate,
    ProducerPort) :-
    selected_candidate_support_class(t, Candidate, Port, SupportClassId),
```

```
    dynamic_support_class_bindable_producer_port(SupportClassId,
    ProducerCandidate, ProducerPort).
produced_bind_option(t,Candidate,Port,TPrev,ToolPrev,ProducerCandidate,ProducerPort):-
    selected_candidate_bindable_producer_port(t, Candidate, Port,
    ProducerCandidate, ProducerPort),
    occurs(TPrev, output(ToolPrev, ProducerCandidate, ProducerPort)), TPrev < t.
```

Listing 5: Predicate-decoupled producer chain (`step.lp`)

Each rule in the chain has at most three body literals, so clingo's grounder produces a number of auxiliary atoms linear in the size of the support graph rather than the Cartesian product of candidate input ports against previously produced outputs.

The Smart Expansion and FastFound patterns demonstrated in [18, 19] are used where every join key is a translation-time identifier (`SupportClassId`, `ProducerCandidate`) rather than an ontology value, so the per-step combinatorics stay independent of the annotation size. A second symmetry-breaking rule canonicalizes consumer to output sub-port bindings for ports split by the 1:N decompression, mirroring the same post-solve binding canonicalization applied after workflow reconstruction so the solver does not enumerate output-port permutations that would later collapse to the same workflow. Constraint handling is shared across encodings. The translation phase (Table 1) lowers each high-level template into a uniform set of `constraint_*` facts, and the optimized `base` program resolves their selectors against the tool taxonomy.

At horizon `t`, `constraint_step(t)` derives the cumulative relation `constraint_selected_time`, and the queried `check` part evaluates the resulting constraints. Each template collapses to a fixed-signature integrity constraint with at most one `#count` aggregate, though the aggregate's tuple shape varies per template: `max_uses` becomes a cardinality bound keyed on a single time variable in `constraint_selected_time`, `mutex_tools` forbids simultaneous presence of two selectors and uses no aggregate, `connected_op` aggregates over a (T_1, T_2) pair of output-flow edges `output_used_by`, and `operation_input` aggregates over a $(T,\ Tool,\ WF)$ tuple of input bindings. This keeps the selection logic unchanged when constraints are added or removed and confines the per-horizon variation to the integrity constraints that are grounded.

Horizon Filtering and Deferred Goal Validation

Between grounding a horizon and running the full `solve()`, the runner applies a two-stage filter before the expensive deferred grounding is activated. First, it runs a translator-side structural feasibility check against the precomputed smart-expansion metadata in F_{opt}.`backend_stats` [18]; this metadata aggregates per-horizon goal-support counts, supportable-candidate counts, missing-goal sets, unsupported-input counts, forced associations, and must-run coverage, so a horizon whose precomputed structural state already proves it infeasible is rejected without invoking the solver. Combined with the precomputed `dynamic_candidate_allowed_at_step` index, this prunes candidates whose feasible window starts after `t` or whose goal distance exceeds the remaining budget, and is the main reason `step(t)` grounding remains usable on large domain models. If the horizon passes this structural screen, the runner grounds `certificate_check(t)` and solves it as an exact-horizon certificate that asks whether all goals *and* globally required must-run tools and candidates are already covered at the current horizon, rejecting models that would only become valid later. If the certificate fails, the deferred `step_query(t)`, `check(t)`, and `check_usage(t)` grounding is skipped. When it succeeds, deferred goal validation is grounded in compressed, profile-based form: each goal is represented by a small set of `(category, profile)` pairs, and an artifact satisfies the goal iff its dimension values are accepted by the required profiles through `dynamic_profile_accepts`. Because these profiles are precomputed by grouping equivalent terminal descendants, the grounder sees only a small number of (*profile*, *value*) pairs per goal instead of the full ontology cross product. On large domain models, these deferred checks dominate the per-horizon atom count, so activating them only after certificate success is necessary to stay below the grounder 32-bit identifier ceiling. After this two-stage filter, the remaining bindability cost is dominated by the $\Theta(B \times D \times A)$ term, where B is the number of bindable

producer-consumer port pairs, D the number of dimensions, and A the per-profile accepted-value count. The K-cluster and 1:N variants add further per-horizon cost surfaces beyond this term: selective-split sub-port facts, check-relevant output-category and profile-class facts, and the corresponding growth in `bindable_pairs`, support classes, and association classes scaled by the per-port multiplicity and the number of decompressed sub-ports.

```
certificate_goal_satisfied(t, GoalID) :- certificate_query(t),
    goal_id(GoalID), certificate_goal_satisfied_by(t, GoalID, _).
:- certificate_query(t), goal_id(GoalID),
    not certificate_goal_satisfied(t, GoalID).
```

Listing 6: Exact-horizon certificate (`exact_certificate.lp`)

Runtime Enumeration at the Query Horizon

Algorithm 1 summarizes how the fact bundle produced by the compressed-candidate translation is injected into the optimized incremental solving loop. At a fixed horizon, each accepted model is followed by a blocking constraint over its selected-candidate sequence so that subsequent iterations enumerate a different workflow structure. Enumeration stops once the configured solution cap is reached, no further model exists at the current horizon, or the global timeout has been reached. For the benchmarks reported in this paper we configure clingo's parallel mode to use two threads in split configuration (`2,split`), which was the setting we found best on our development hardware; this is a configuration choice rather than a property of the optimized backend itself. For input-usage constraints (`enable_all_inputs_used`, `enable_some_input_used`, `enable_inputs_consumed`), it derives usage history from the stepwise workflow-input-to-artifact mapping rather than constructing the full produced-artifact ancestry closure.

3. Evaluation

To evaluate `clindaws` we compared it against APE across the benchmark domains listed in Table 2. Each domain is run in a no-constraints (NC) and a with-constraints (WC) variant; a notation such as $5 \times \{\text{NC, WC}\}$ for GeoGMT means five distinct workflow specifications each run in both variants. The MassSpectrometry benchmarks span three tool-count tiers that together stress the large-scale regime.

All runs used a MacBook Pro with an Apple M5 Pro chip and 64 GB RAM. Both frameworks received identical inputs and used the same configuration, including solution caps and timeouts of 900 s for small and medium cases and 6000 s for the large MassSpectrometry benchmarks. We record peak memory usage, grounding time, and solving time; for APE the *problem setup* column covers CNF construction and building the paths before sending them to MiniSAT for validation. All measurements are cold-start single-query runs and therefore do not measure cross-query reuse after editing constraints or goals. For `clindaws`, grounding and solving are measured after Python-side translation and precomputation; this translation phase took less than one second on small and medium test cases and about 14 seconds on average for the large full `bio.tools` cases. Table 3 summarizes the resulting peak memory, grounding time, and solving time for the three optimized variants and APE.

3.1. Benchmarking Results

The earlier single-shot and plain multi-shot encodings were also run on the same suite, but are not included in Table 3 because they timed out or hit a `clasp` memory error on most medium- and large-scale examples. Their representative total ASP runtimes show the effect of the encoding generations: on GeoGMT 3-WC, runtime drops from 17.46 s in the single-shot encoding to 6.81 s in the plain multi-shot encoding and 1.45 s in the optimized default backend; on MassSpec 1-extended, from 114.14 s to 4.35 s and 0.81 s; on Metabolomics NC, from 0.64 s to 0.32 s and 0.09 s. This is a coarse generation-level ablation rather than an isolation of each individual optimization, but it shows that both incremental

grounding and the compressed-candidate backend contribute to the final performance. Across all small and medium benchmarks, the three `clindaws` variants generally require less memory and have shorter grounding times than APE. For GeoGMT 2-NC, 2-WC, 4-NC, and 4-WC, their grounding times are comparable to or longer than APE's, although they still use less memory. The largest memory savings occur in ontology-heavy benchmarks, where the annotated tools constitute only a subset of the full ontology. In Metabolomics (3,421 terms and 37 tools), `clindaws` uses approximately 133 MB, compared with 4,100–4,600 MB for APE, a reduction of 96.8–97.1%. In the original MassSpec benchmark, memory usage decreases from 7,800–12,700 MB to 154–242 MB. In the extended MassSpec benchmark, it decreases from 5,400–11,300 MB to 1,900–4,700 MB. For most benchmarks, `clindaws` completes grounding in under 3 s, whereas APE requires tens to hundreds of seconds. For example, on GeoGMT 3-WC, APE requires 626 s, while all `clindaws` variants finish in under 1 s. At 1,678 tools, all three `MassSpectrometry` full-`bio.tools` variants consume 47–52 GB, compared with 12–14 GB for APE, because the precomputed bindability graph itself is large. Grounding is still faster (197–266 s vs 314–818 s) and total runtime is lower on cases 1, 2, and 4. Case 3-full is the exception: APE finishes in 1,714 s total while *default* needs 3,600 s and *k-cluster* 4,800 s; The *1:N* variant exceeds `clasp`'s 32-bit identifier range and crashes. Among the optimized variants, *default* consistently shows the highest peak memory; *k-cluster* and *1:N* are comparable and lower. Solving time depends on the domain. On metaCascabel NC, *default* takes 104 s while both decomposition variants finish in 7 s. On GeoGMT 2-WC, the roles reverse (*default* 1 s, *k-cluster* 92 s) because decomposition widens the search space on that configuration. The starred rows mark cases that require further explanation. MassSpec 2-full (*): *default* and *k-cluster* find 65 workflows at horizon 3, APE and *1:N* find 77; all runs reach 1,000 candidates and terminate at horizon 4. Metabolomics (*) yields only one valid workflow, so its timing reflects finding a single candidate at $t = 1$ and then running until $t = 10$. The MassSpec 3-ext. k-cluster peak-memory monitor produced an implausible value; we mark that cell as unavailable and do not use it in the discussion.

Figure 2 plots peak memory by workflow horizon for representative full MassSpectrometry configurations. It shows that memory for the optimized ASP variants grows sharply once dense horizons are reached, whereas APE follows a slower but still increasing profile.

3.2. Discussion

The memory and grounding reductions on small and medium domains stem from moving the tool-type cross product out of the ASP grounder and into Python precomputation. APE and the direct ASP

Table 2
Benchmarking domains. NC = no constraints, WC = with constraints.

Domain	Description	Workflow Specifications	# Ontology Terms	# Tools
GeoEvents [5]	GIS spatial operations	1 NC	101	49
GeoGMT [5]	Map generation (GMT)	5 × {NC, WC}	137	34
ImageMagick*	Image manipulation	2 × {NC, WC}	40	35
Metabolomics	Metabolomics	1 NC	3,421	37
metaCascabel*	Metagenomic assembly	1 NC, WC	150	38
Defect Concentration [8]	Materials science	1 NC, WC	94	28
QuAnGIS [20]				
ArcGIS	ArcGIS subset	1 NC	75	110
No1	Hand-picked scenario	1 NC, WC	75	15
QGIS	QGIS subset	1 NC, WC	75	15
MassSpectrometry [6]				
Original (small)	Original proteomics set	4 × WC	3,421	20
Extended (medium)	Proteomics-domain set	4 × WC	3,421	277
Full Bio.Tools (large)	Proteomics	4 × WC	3,421	1,678

*The starred test cases were created in separate contexts: ImageMagick as an APE demo/test case (see https://ape-framework.readthedocs.io/en/latest/docs/demo/imagemagick/imagemagick.html) and metaCascabel through a student internship project.

Table 3
Optimized multi-shot and APE benchmark results.

	default no decomp.			multi-shot optimized k-cluster decomp.			1:n decomp.			APE		
	peak [MB]	ground [s]	solve [s]	peak [MB]	ground [s]	solve [s]	peak [MB]	ground [s]	solve [s]	peak [MB]	problem setup [s]	solve [s]
GeoEvents												
NC	474.76	0.16	0.58	249.96	0.15	0.91	251.11	0.15	1.01	1526.84	0.26	1.06
GeoGMT												
0-NC	283.11	0.08	0.14	184.40	0.08	0.35	903.25	0.38	11.19	827.85	0.35	0.28
0-WC	1451.09	0.44	2.72	806.07	0.36	11.13	903.25	0.38	11.19	2048.00	1.33	0.84
1-NC	2944.67	1.58	0.65	1198.18	1.82	2.58	1198.43	1.80	1.77	1192.15	1.50	0.84
1-WC	2801.51	1.45	1.31	1151.76	1.63	1.32	1152.77	1.63	0.83	1388.57	2.18	1.37
2-NC	647.32	0.20	13.17	430.16	0.22	14.42	440.79	0.22	14.31	837.84	0.36	12.51
2-WC	2801.51	1.45	1.31	841.55	0.68	92.27	848.13	0.69	52.81	6894.06	9.58	21.68
3-NC	1959.15	0.94	0.33	838.24	1.06	0.28	839.23	1.06	0.28	1011.53	1.09	0.47
3-WC	1902.86	0.91	0.54	822.25	1.04	0.65	821.34	1.03	0.61	2625.79	626.19	5.53
4-NC	2915.43	1.62	0.56	1200.36	1.76	2.48	1200.08	1.79	1.94	1204.79	1.51	0.84
4-WC	4541.23	2.12	7.89	2256.91	2.11	53.63	2269.01	2.14	53.59	1690.75	3.28	4.32
ImageMagick												
1-NC	596.14	0.21	0.30	275.26	0.20	0.39	275.45	0.21	0.39	860.76	0.30	1.11
1-WC	590.72	0.20	0.31	276.76	0.20	0.43	276.53	0.20	0.43	1127.57	0.31	1.07
2-NC	1801.16	0.51	1.40	154.08	0.06	0.20	153.58	0.06	0.20	469.95	0.16	0.92
2-WC	654.51	0.21	0.36	290.48	0.19	0.42	290.82	0.20	0.41	1192.19	0.39	1.31
MassSpectrometry												
1-ext.	1074.79	0.57	0.24	528.90	0.63	0.20	522.67	0.62	0.21	5430.79	70.01	9.68
1-full	47424.66	212.47	369.49	47758.58	196.188	177.69	51849.5	266.13	312.45	13290.93	313.83	596.35
1-orig	182.44	0.08	0.30	154.20	0.08	0.30	154.18	0.08	0.30	9405.76	31.97	11.57
2-ext.	4747.18	2.98	6.76	1954.53	3.01	10.42	1930.68	3.03	10.45	10315.06	121.03	142.95
2-full*	47919.47	215.41	434.48	47688.27	197.15	387.29	52731.02	267.11	909.58	12891.41	772.59	4133.16
2-orig.*	188.61	0.07	0.29	156.03	0.07	0.28	155.91	0.07	0.29	11025.44	38.37	6.65
3-ext.	4721.63	3.03	3.70	–	3.02	4.50	1948.45	3.06	4.56	11307.04	120.81	365.87
3-full	47641.73	715.89	2882.86	48465.24	457.38	4346.29	-	-	-	14489.56	802.72	911.11
3-orig.	241.48	0.12	0.27	184.58	0.11	0.28	184.48	0.11	0.28	12665.21	37.57	12.49
4-ext.	2847.19	1.66	2.28	1219.50	1.70	2.31	1221.81	1.73	2.55	8514.17	92.47	45.44
4-full	48020.21	224.59	493.91	51947.27	201.05	407.21	51698.51	260.51	501.38	12197.47	817.74	1205.22
4-orig.	209.57	0.09	0.32	166.42	0.09	0.31	166.29	0.09	0.31	7827.25	331.77	10.35
Metabolomics												
NC*	133.18	0.08	0.01	134.37	0.08	0.01	132.86	0.08	0.01	4566.88	200.62	49.33
QuAnGIS												
ArcGIS	137.64	0.04	0.01	134.59	0.04	0.01	134.40	0.04	0.01	724.43	0.35	0.09
1-NC	100.55	0.02	0.003	100.15	0.02	0.003	100.15	0.02	0.003	751.82	0.21	0.09
1-WC	100.06	0.04	0.01	100.45	0.04	0.01	99.91	0.04	0.01	1168.53	0.42	0.16
QGIS-NC	100.81	0.02	0.002	99.75	0.02	0.002	100.06	0.02	0.002	77.42	0.14	0.10
QGIS-WC	100.57	0.02	0.003	100.22	0.02	0.003	100.12	0.02	0.003	80.28	0.14	0.09
DefectConcentration												
NC	169.17	0.05	0.72	139.18	0.05	0.73	139.73	0.05	0.72	1388.97	0.50	1.79
WC	229.08	0.10	0.84	173.00	0.10	0.86	174.98	0.10	0.85	2371.20	0.91	2.48
metaCascabel												
NC	573.68	0.21	103.69	294.16	0.22	6.74	294.15	0.22	7.43	1083.60	7.80	54.43
WC	944.28	0.39	0.23	424.63	0.41	0.21	425.23	0.41	0.21	644.35	7.98	9.67

encodings re-expand the full product of tools, type values, and ontology terms at each horizon; the optimized backend emits it once as compact identifier-indexed facts so per-step atom counts stay linear in the candidate graph. The two-stage feasibility filter (structural pre-check then certificate check) additionally avoids grounding the expensive `check(t)` rules on horizons that cannot yield solutions. The observation that reducing the number of tools considered decreases grounding and solving time as well as memory use suggests that a lazy-evaluation approach could further improve scalability, even for large tool sets. At 1,678 tools the precomputed bindability graph itself is large, and `check(t)` grounding scales linearly with output ports and goal profiles, so the optimization delivers a constant-factor improvement. Peak memory is dominated by those materialized facts held across all horizons, which is why `clindaws` exceeds APE's footprint on the full-scale cases despite being faster. Within the optimized variants, keeping ports merged (*default*) minimizes emitted facts but produces a wide search

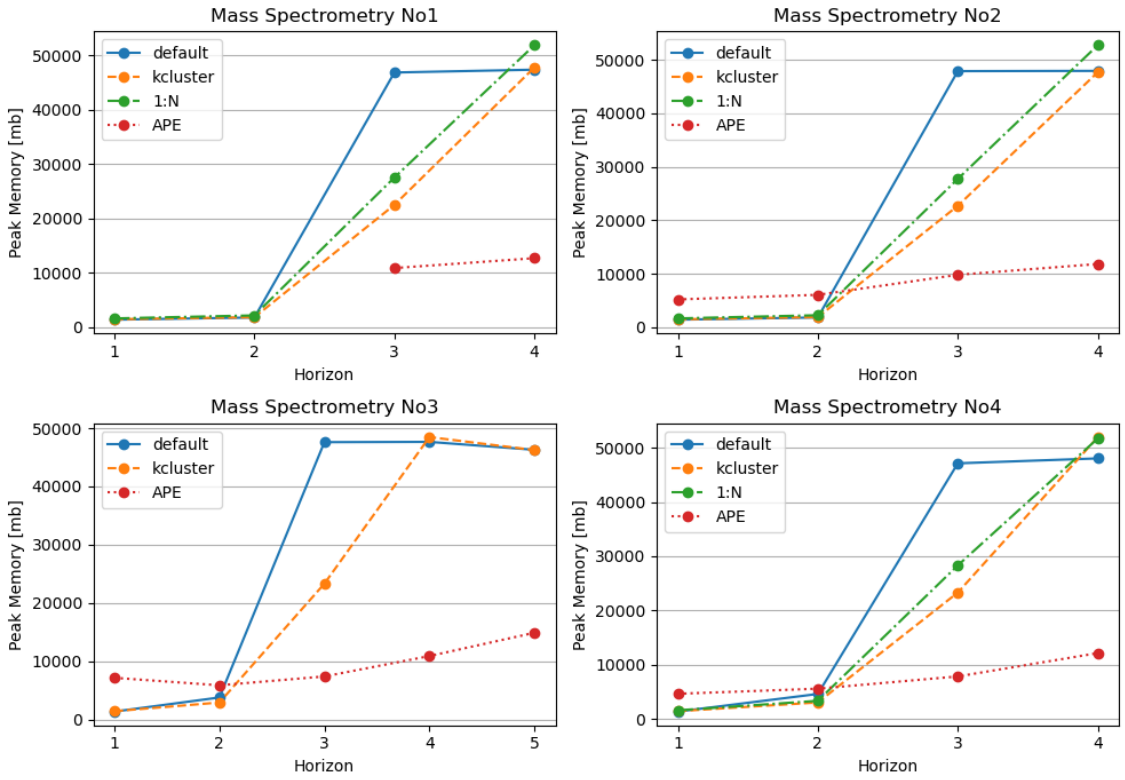


Figure 2: Peak memory by workflow horizon for representative benchmark configurations.

state. Decompressing ports (*k-cluster* or *1:N*) tightens the search at the cost of additional grounding: *k-cluster* partitions consumers into equivalence classes and emits one sub-port per class, losing the 12 MassSpec 2-full candidates that require finer allocation; *1:N* restores these allocation choices and matches APE's workflow space on the completed benchmark, but inflates fact tables proportionally to source multiplicity, pushing the densest configuration past `clasp`'s 32-bit identifier ceiling. The directly observable consequence is that too many atoms reach the grounder on dense domains, causing peak memory to blow up, triggering `clasp`'s 32-bit identifier-range error on one test case, and inflating solving time on some domain models. The 32-bit identifier range of the current clingo–clasp pipeline is a practical capacity constraint: clingo represents atoms and solver literals with 32-bit types [21]. It does not change the central engineering conclusion that the encoding must avoid generating this many ground atoms; the compressed-candidate translation is therefore necessary rather than merely an optimization. Wider 64-bit identifiers could postpone this boundary, but would not address the accompanying memory growth and are not a substitute for encoding-level compression.

4. Related Work

`clindaws` is most directly related to APE [3, 4, 5], its predecessor and comparison baseline. Both frameworks synthesize scientific workflows from tool annotations, ontology-backed types, and constraints, but `clindaws` replaces APE's per-query SAT re-encoding with incremental ASP solving in `clingo`. This matters for interactive composition, where small changes to goals or constraints should reuse prior work. This section surveys further relevant related work.

ASP planning and configuration. Answer set planning [22] and PDDL-to-ASP translations such as `plasp` [23] search horizon by horizon for goal-reaching action sequences. Workflow synthesis shares this incremental search pattern, but its state is a *typed artifact flow*, not a set of world fluents, so a direct PDDL-style encoding would reintroduce the type-ontology cross product that `clindaws` avoids. It is also close to ASP-based configuration, where predicate-decoupled encodings and assumption-

driven interaction scale to industrial instances [24]. The difference is that workflow synthesis must also construct the variable structure (tool positions, data dependencies, and port bindings) during the search. Closer in spirit to the temporal-constraint side of `clindaws` is Telingo [25], which extends ASP with linear-time temporal operators on finite traces and evaluates them through clingo's theory backend rather than expanding them into time-indexed integrity constraints. This is directly relevant to `clindaws`, whose constraint templates currently lower into time-keyed integrity constraints.

Compression and symmetry reduction. The optimized backend is further connected to work on equivalence compression and symmetry breaking. The k-cluster step is not clustering in the usual optimization sense; it computes exact equivalence classes of identical support sets in a single pass. In that respect it is closer to partition refinement in DFA minimization [26]. Likewise, workflow inputs with identical type profiles are interchangeable in the same sense as symmetric objects in SAT-based planning [27], and `clindaws` exploits this by canonicalizing them before encoding. When ports are decomposed, additional symmetry-breaking predicates [28] ensure that interchangeable sub-ports do not generate redundant workflow variants. Underlying these reductions is ontology-based type reasoning: subsumption over rooted-DAG taxonomies compresses signature profiles into shared identifiers without an external DL reasoner [29].

LLM-based workflow generation. Recent systems use large language models to suggest scientific workflows from natural-language requests [30, 31]. Benchmarks on frontier models show that while plausible pipeline structures can often be drafted, tool-interface constraints are routinely violated and type-incompatible chains remain common [32]. `clindaws` addresses a different requirement: it returns only workflows that satisfy the formal type and constraints and produces deterministic results from an explicit ontology. The two approaches are therefore complementary: LLM tools may help with domain-model authoring or candidate ranking, while `clindaws` provides the constraint-sound enumeration that sampling-based generation cannot guarantee.

5. Conclusion and Future Work

We have built `clindaws`, a prototype showing that clingo is a viable alternative for composing workflow candidates. Compared with APE, `clindaws` shows performance advantages on small to medium-sized domain models. Bottlenecks remain, especially peak memory and grounding time on large domain models and the per-step re-grounding cost of time-keyed integrity constraints. Optimizing resource efficiency for local and remote deployments is our immediate priority.

Future work will proceed along several paths. We will integrate `clindaws` into APE as a second solver choice and use the multi-shot interface to support step-by-step manual construction of workflow candidates. This would be useful with workflow engines like pyiron, Nextflow, and Galaxy, where APE can act as a recommender system. We also plan to use weak constraints to rank candidates by user preferences. To address the per-horizon re-grounding cost of time-keyed integrity constraints, we plan to explore temporal-logic ASP along the lines of Telingo [25]. This also opens a path to representing domain models with Basic Formal Ontology (BFO) [33] in ASP, where occurrents and continuants are distinguished through explicit temporal information. This is useful for multi-scale computational materials-science workflows [8], which may run in parallel and interact across scales. For large domain models, caching grounded atoms and partial workflow candidates may further reduce runtime.

Another focus of our future work will also lie in the exploration of a neuro-symbolic based approach to address scaling, uncertainty, and manual modeling bottlenecks. Because workflow synthesis resembles complex SAT and ASP configuration problems, we plan to use neural networks, such as Graph Neural Networks, to predict variable assignments. This may let us prune the domain model before grounding and guide the solver at runtime. We will also explore neuro-probabilistic frameworks like SLASH [34] and NeurASP [35] to combine neural networks with logical programs. Finally, we will investigate inductive learning methods like NSIL [36] to extract and refine symbolic ASP rules from experimental logs and scientific datasets.

Acknowledgments

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), Project-ID 414984028, SFB 1404 FONDA.

Declaration on Generative AI

During the preparation of this work, the author(s) used Gemini 3.1 Pro and DeepL for grammar and spell checks. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] F. Suter, T. Coleman, I. Altintas, et al., A terminology for scientific workflow systems, Future Generation Computer Systems 174 (2026) 107974. doi:10.1016/j.future.2025.107974.

[2] U. Leser, M. Hilbrich, S. Wilkinson, R. Ferreira da Silva, Workflow systems for large-scale scientific data analysis, Berlin Universities Publishing, 2026. URL: https://depositonce.tu-berlin.de/handle/11303/25179. doi:10.14279/depositonce-24000.

[3] V. Kasalica, A.-L. Lamprecht, APE: A command-line tool and API for automated workflow composition, in: Computational Science – ICCS 2020, Lecture Notes in Computer Science, Springer, 2020, pp. 464–476. doi:10.1007/978-3-030-50436-6_34.

[4] V. Kasalica, A.-L. Lamprecht, Workflow Discovery with Semantic Constraints: The SAT- Based Implementation of APE, Electronic Communications of the EASST 78 (2020). doi:10.14279/tuj.eceasst.78.1092.

[5] V. Kasalica, Synthesis of Scientific Workflows: Theory and Practice of an Instance-Aware Approach, Ph.D. thesis, Utrecht University, 2022. URL: https://dspace.library.uu.nl/handle/1874/423894. doi:10.33540/160.

[6] V. Kasalica, V. Schwämmle, M. Palmblad, J. Ison, A.-L. Lamprecht, APE in the Wild: Automated Exploration of Proteomics Workflows in the bio. tools Registry, Journal of proteome research 20 (2021) 2157–2165. doi:10.1021/acs.jproteome.0c00983.

[7] J. F. Kruiger, V. Kasalica, R. Meerlo, A.-L. Lamprecht, E. Nyamsuren, S. Scheider, Loose programming of GIS workflows with geo-analytical concepts, Transactions in GIS 25 (2021) 424–449. doi:10.1111/tgis.12692.

[8] N. Bertrand, A.-L. Lamprecht, C. Grunert, O. Waseda, T. Hickel, Semantic workflow composition for computational materials science, 2026. URL: https://doi.org/10.5281/zenodo.18822541. doi:10.5281/zenodo.18822541.

[9] V. Kasalica, F. Morato, A. Lamprecht, M. Palmblad, Benchmarks for bioinformatics workflow bake offs, 2024. doi:10.37044/OSF.IO/RQ2DA.

[10] V. Kasalica, P. Kok, R. Marissen, M. Frank, M. Palmblad, A.-L. Lamprecht, Tackling analytical variability with workflomics, in: U. Leser, M. Hilbrich, S. Wilkinson, R. Ferreira da Silva (Eds.), Workflow Systems for Large-Scale Scientific Data Analysis, Berlin University Press, 2026, pp. 147–167. doi:10.14279/depositonce-25819.

[11] J. Ison, H. Ienasescu, P. J. Chmura, et al., The bio.tools registry of software tools and data resources for the life sciences., Genome Biology (2019). doi:10.1186/S13059-019-1772-6.

[12] V. Lifschitz, Answer set programming and plan generation, Artificial Intelligence (2002). doi:10.1016/S0004-3702(02)00186-8.

[13] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot asp solving with clingo, Theory and Practice of Logic Programming (2019). doi:10.1017/S1471068418000054.

[14] M. Gebser, B. Kaufmann, A. Neumann, T. Schaub, clasp : A conflict-driven answer set solver, International Conference on Logic Programming and Non-Monotonic Reasoning (2007). doi:10.1007/978-3-540-72200-7_23.

[15] N. Bertrand, Workflomics/clindaws: Prototype version 1, 2026. URL: https://doi.org/10.5281/zenodo.20113712. doi:10.5281/zenodo.20113712.
[16] N. Sörensson, N. Eén, Minisat v1.13 - a sat solver with conflict-clause minimization, SAT 2005 Competition (2005). URL: https://minisat.se/downloads/MiniSat_v1.13_short.pdf.
[17] E. Böhl, S. Ellmauthaler, S. A. Gaggl, Winning snake: Design choices in multi-shot asp, Theory and Practice of Logic Programming 24 (2024) 772–789. doi:10.1017/S1471068424000267.
[18] L. Balážová, R. Comploi-Taupe, S. Hahn, N. Rühling, G. Schenner, Smart expansion techniques for ASP-based interactive configuration, Theory and Practice of Logic Programming 25 (2025) 473–488. doi:10.1017/S147106842510029X.
[19] A. Beiser, M. Gebser, M. Hecher, S. Woltran, FastFound: Easing the ASP bottleneck via predicate-decoupled grounding, in: Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning (KR 2025), 2025, pp. 100–109. doi:10.24963/kr.2025/10.
[20] V. Kasalica, A.-L. Lamprecht, Workflow discovery through semantic constraints: A geovisualization case study, in: Computational Science and Its Applications (ICCSA 2019), Part III, Lecture Notes in Computer Science, Springer, 2019, pp. 473–488. doi:10.1007/978-3-030-24302-9_34.
[21] Potassco, clingo C API: Basic data types and functions, 2026. URL: https://potassco.org/clingo/c-api/current/group__BasicTypes.html.
[22] T. C. Son, E. Pontelli, M. Balduccini, T. Schaub, Answer set planning: A survey, Theory and Practice of Logic Programming 23 (2023) 226–298. doi:10.1017/S1471068422000072.
[23] Y. Dimopoulos, M. Gebser, P. Lühne, J. Romero, T. Schaub, plasp 3: Towards effective ASP planning, Theory and Practice of Logic Programming 19 (2019) 477–504. doi:10.1017/S1471068418000583.
[24] J. Baumeister, K. Herud, M. Ostrowski, et al., Towards industrial-scale product configuration, in: Proceedings of the 17th Int. Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR 2024), volume 15245 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 71–84. doi:10.1007/978-3-031-74209-5_6.
[25] P. Cabalar, R. Kaminski, P. Morkisch, T. Schaub, telingo = ASP + Time, in: Logic Programming and Nonmonotonic Reasoning (LPNMR 2019), Lecture Notes in Computer Science, Springer, 2019, pp. 256–269. doi:10.1007/978-3-030-20528-7_19.
[26] J. E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Z. Kohavi, A. Paz (Eds.), Theory of Machines and Computations, Academic Press, New York, 1971, pp. 189–196. doi:10.1016/B978-0-12-417750-5.50022-1.
[27] J. Rintanen, Symmetry reduction for sat representations of transition systems, in: Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS 2003), AAAI Press, 2003, pp. 32–41. URL: https://dblp.org/rec/conf/aips/Rintanen03.
[28] J. M. Crawford, M. L. Ginsberg, E. M. Luks, A. Roy, Symmetry-breaking predicates for search problems, in: Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR 1996), 1996, pp. 148–159. URL: https://dblp.org/rec/conf/kr/CrawfordGLR96.
[29] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003.
[30] M. Esmailoghli, M. Weidlich, FlowPilot: A suggestion system for designing scientific workflows, Proceedings of the ACM on Management of Data 4 (2026) 1–27. doi:10.1145/3786653.
[31] J. Xu, W. Du, X. Liu, X. Li, LLM4Workflow: An LLM-based automated workflow model generation tool, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024), Tool Demonstrations, Association for Computing Machinery, 2024, pp. 2394–2398. doi:10.1145/3691620.3695360.
[32] K. Alam, B. Roy, From prompt to pipeline: Large language models for scientific workflow development in bioinformatics, Research Square (2025). doi:10.21203/rs.3.rs-7642675/v1, preprint.
[33] J. N. Otte, J. Beverley, A. Ruttenberg, Bfo: Basic formal ontology, Applied Ontology 17 (2022)

17–43. doi:10.3233/AO-220262.
[34] A. Skryagin, W. Stammer, D. Ochs, D. S. Dhami, K. Kersting, Neural-probabilistic answer set programming, International Conference on Principles of Knowledge Representation and Reasoning (2022). doi:10.24963/KR.2022/48.
[35] Z. Yang, A. Ishay, J. Lee, Neurasp: Embracing neural networks into answer set programming, International Joint Conference on Artificial Intelligence (2020). doi:10.24963/IJCAI.2020/243.
[36] D. Cunnington, M. E. Law, J. Lobo, A. Russo, Neuro-symbolic learning of answer set programs from raw data, International Joint Conference on Artificial Intelligence (2023). doi:10.24963/IJCAI.2023/399.