Comparing SLTL^x-based Synthesis and ASP for the Automated Composition of Data Analysis Workflows (Extended Abstract)

Nikolas Bertrand[0000−0001−7259−2111], Mario Frank[0000−0001−8888−7475], and Anna-Lena Lamprecht[0000−0003−1953−5606]

University of Potsdam, 14476 Potsdam, Germany
`nikolas.bertrand@uni-potsdam.de`, `mario.frank@uni-potsdam.de`, `anna-lena.lamprecht@uni-potsdam.de`

Problem Description Many scientific disciplines are generating increasingly larger datasets that need analysis to extract meaningful insights. Data Analysis Workflows (DAWs) are often used for orchestrating computational tools to process and analyze this data [4]. As the tools available become more numerous and complex, manually constructing valid workflows is increasingly challenging. The Automated Pipeline Explorer (APE) [8] addresses this challenge by automatically composing workflow candidates using a semantic domain model and user configuration. This model includes the inputs, outputs, and data-type relationships of tools, derived from an ontology. APE transforms a given workflow specification (containing the initial data inputs, eventual outputs, and constraints) by finding valid configurations, leveraging temporal constraints in the form of SLTL^x [7], an instance-aware extension of Semantic Linear Temporal Logic (SLTL) [12] to distinguish data instances and facilitate executable workflow construction. APE has successfully been used for workflow composition in different scientific domains [10,9]. However, this has also identified some open challenges. For example, APE's performance is fairly weak when workflow candidates become larger. This motivates the question whether an approach using Answer Set Programming (ASP) and Clingo [3,11] as a replacement for APE's encoding and synthesis implementations [7] could be more efficient.

ASP-based problem modeling To evaluate its base performance, we model workflow synthesis using ASP, framing the problem as a search for a stable model representing a valid execution trace. The system's domain knowledge is encoded in a static program base, including taxonomies defining type hierarchies, such as $taxonomy(csv\_file, file)$, and multi-dimensional data types describing tool I/O signatures. Semantic propagation of types is implemented via deductive rules computing the transitive closure of the taxonomy and is achieved through forward and backward propagation. Forward propagation,
$out(Parent) \leftarrow out(Child)$, generalizes outputs: Producing a csv_file (Child) counts as producing a $file$ (Parent). Backward propagation,
$in(Child) \leftarrow in(Parent)$, relaxes input requirements: A tool requiring a $file$

(Parent) can also accept a csv_file (Child). Tool outputs (and inputs resp.) are represented as $output\_mode(tool\_id, mode\_id, Ix, Dim1, Dim2)$, where $tool\_id$ refers to the unique tool, $mode\_id$ specifies an operational mode, Ix is the index for a specific input or output within that mode, and $Dim1$ and $Dim2$ represent different dimensions for each tool. At least one $mode_id$ must be chosen by the solver, and all its associated (Ix) must hold. With this, one can represent instances where either single or multiple inputs or outputs are required. A binding between two tools, using the output at step $t-1$ and the input at step t, is valid if their propagated type sets intersect. Integrity constraints prune any model where an input cannot be satisfied, ensuring data-flow consistency via semantic unification, which determines compatibility based on type hierarchy. Workflow candidates are constructed incrementally up to a set maximum length. Using the choice rules shown in Listing 1.2, at each step t, valid next steps are generated by choosing a tool, identifying forward and backward propagation options, and resolving possible inputs and outputs for the selected tool. The workflow is composed step by step, ensuring semantic and type consistency throughout.

Preliminary Results We used a problem domain that creates workflows based on the steps needed for baking a pizza (Listing 1.1), which has a complexity comparable to the Proteomics domain from the bio.tools problem set [5], but on a smaller and controllable scale and without using temporal constraints. One of the composed workflows can be seen in Figure 1. The tests were run on an Apple M4 Pro with 48Gb of RAM. The results shown in Table 1 indicate a significant speed up and improved RAM usage for the problem encoding and synthesis for the ASP variant. For ASP, the worst and best case were tracked since the runtime differed significantly depending on the path chosen by the solver, while APE runtimes showed very little variance.

Table 1. Performance comparison between APE and an implementation using incremental solving in ASP for a workflow of maximum length 15. All times are in seconds.

Workflow Length 15	**APE**	**ASP Worst**	**ASP Best**
Peak Memory used:	25.4 GB	303 MB	281 MB
Total encoding - synthesize / grounding time:	192.77s	0.42s	0.42s
Total solving time:	4.70s	43.90s	1.22s
Total runtime:	211.20s	44.32s	1.64s

Conclusion In our preliminary results, Clingo vastly outperforms APE. It remains to be evaluated, however, how the results change with increased complexity through multi-dimensional domains and additional constraints. Therefore, we plan to include temporal constraints [1] through a meta-programming based approach [6] or using Telingo [2] for further comparison.

Acknowledgments. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 414984028 – SFB 1404 FONDA;
We would like to acknowledge the help in proofreading and smaller corrections of the manuscript by Sebastian Müller.

References

1. Bosser, A.G., Cabalar, P., Diéguez, M., Schaub, T.: Introducing Temporal Stable Models for Linear Dynamic Logic. In: KR. pp. 12–21 (2018)
2. Cabalar, P., Kaminski, R., Morkisch, P., Schaub, T.: telingo= ASP+ time. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 256–269. Springer (2019). https://doi.org/10.1007/978-3-030-20528-7_19
3. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. CoRR **abs/1705.09811** (2017). https://doi.org/10.1017/S1471068418000054
4. Hilbrich, M., Müller, S., Kulagina, S., Lazik, C., De Mecquenem, N., Grunske, L.: A Consolidated View on Specification Languages for Data Analysis Workflows. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering. pp. 201–215. Springer Nature Switzerland, Cham (2022). https://doi.org/10.1007/978-3-031-19756-7_12
5. Ison, J., Ienasescu, H., Chmura, P., Rydza, E., Ménager, H., Kalaš, M., Schwämmle, V., Grüning, B., Beard, N., Lopez, R., et al.: The bio. tools registry of software tools and data resources for the life sciences. Genome biology **20**(1), 164 (2019). https://doi.org/10.1186/s13059-019-1772-6
6. Kaminski, R., Romero, J., Schaub, T., Wanko, P.: How to build your own ASP-based system?! Theory and Practice of Logic Programming **23**(1), 299–361 (2023). https://doi.org/10.1017/S1471068421000508
7. Kasalica, V., Lamprecht, A.L.: Workflow discovery with semantic constraints: The SAT-based implementation of APE. Electronic Communications of the EASST **78** (2019). https://doi.org/10.14279/tuj.eceasst.78.1092
8. Kasalica, V., Lamprecht, A.L.: APE: A command-line tool and API for automated workflow composition . In: International Conference on Computational Science. pp. 464–476. Springer (2020). https://doi.org/10.1007/978-3-030-50436-6_34
9. Kasalica, V., Schwammle, V., Palmblad, M., Ison, J., Lamprecht, A.L.: APE in the wild: Automated exploration of proteomics workflows in the bio.tools registry . https://doi.org/10.1021/acs.jproteome.0c00983
10. Kruiger, J.F., Kasalica, V., Meerlo, R., Lamprecht, A.L., Nyamsuren, E., Scheider, S.: Loose programming of GIS workflows with geo-analytical concepts **25**(1), 424–449. https://doi.org/10.1111/tgis.12692
11. Lifschitz, V.: Answer set programming, vol. 3. Springer Cham (2019). https://doi.org/10.1007/978-3-030-24658-7
12. Steffen, B., Margaria, T., Informatik, L., Freitag, B.: Module Configuration by Minimal Model Construction (02 1998)

A Domain Model - 2-Dimensional Pizza Baking

Listing 1.1. Partial Representation of the Domain Model translated into ASP facts.

```
% ############# TAXONOMY DEFINITIONS #############
% —— Dimension Roots ——
taxonomy_type_root("Type").
taxonomy_type_root("State").

% —— State Hierarchy ——
taxonomy("raw", "State").
taxonomy("mixed", "State").
taxonomy("kneaded", "State").
...
% —— Type Hierarchy (all are direct children of Type) ——
taxonomy("Flour", "Type").
taxonomy("Salt", "Type").
taxonomy("Yeast", "Type").
taxonomy("Water", "Type").
...

% ############# WORKFLOW I/O #############
% —— Initial Inputs ——
out((0,1), ("Type", "Flour"), ("State", "raw")).
out((0,2), ("Type", "Salt"), ("State", "raw")).
out((0,3), ("Type", "Yeast"), ("State", "raw")).
out((0,4), ("Type", "Water"), ("State", "raw")).
out((0,5), ("Type", "Tomatoes"), ("State", "raw")).
...
% —— Final Output ——
in((-1,1), ("Type", "EatenPizza_1"), ("State", "consumed")).

%############# TOOL ANNOTATIONS #############
annotated("mix_dry_ingredients").
tool_mode("mix_dry_ingredients", 1).
mode_input("mix_dry_ingredients",1,1, ("Type","Flour"),
                                      ("State","raw")).
mode_input("mix_dry_ingredients",1,2, ("Type","Salt"),
                                      ("State","raw")).
mode_input("mix_dry_ingredients",1,3, ("Type","Yeast"),
                                      ("State","raw")).
mode_output("mix_dry_ingredients",1,1,("Type","DryMix"),
                                      ("State","mixed")).
annotated("add_water_and_knead").
tool_mode("add_water_and_knead", 1).
mode_input("add_water_and_knead",1,1, ("Type","DryMix"),
                                      ("State","mixed")).
```

```
mode_input("add_water_and_knead",1,2, ("Type","Water"),
                                      ("State","raw")).
mode_output("add_water_and_knead",1,1,("Type","KneadedDough"),
                                      ("State","kneaded")).
annotated("let_dough_rise").
tool_mode("let_dough_rise", 1).
mode_input("let_dough_rise",1,1,("Type","KneadedDough"),
                                ("State","kneaded")).
mode_output("let_dough_rise",1,1,("Type","RisenDough"),
                                 ("State","risen")).

annotated("roll_dough").
tool_mode("roll_dough", 1).
mode_input("roll_dough",1,1,     ("Type", "RisenDough"),
                                 ("State","risen")).
mode_output("roll_dough",1,1,    ("Type", "PizzaBase"),
                                 ("State","rolled")).

annotated("add_cheese").
tool_mode("add_cheese", 1).
mode_input("add_cheese", 1, 1,   ("Type", "SaucedPizzaBase"),
                                 ("State", "base_sauced")).
mode_input("add_cheese", 1, 2,   ("Type", "Cheese"),
                                 ("State", "raw")).

mode_output("add_cheese", 1, 1, ("Type", "CheesedPizza"),
                                 ("State", "cheesed")).

annotated("eat_pizza").
tool_mode("eat_pizza", 1).
mode_input("eat_pizza", 1, 1,    ("Type", "ServedPizza"),
                                 ("State", "served")).
mode_output("eat_pizza", 1, 1,   ("Type", "EatenPizza_1"),
                                 ("State", "consumed")).

...
```

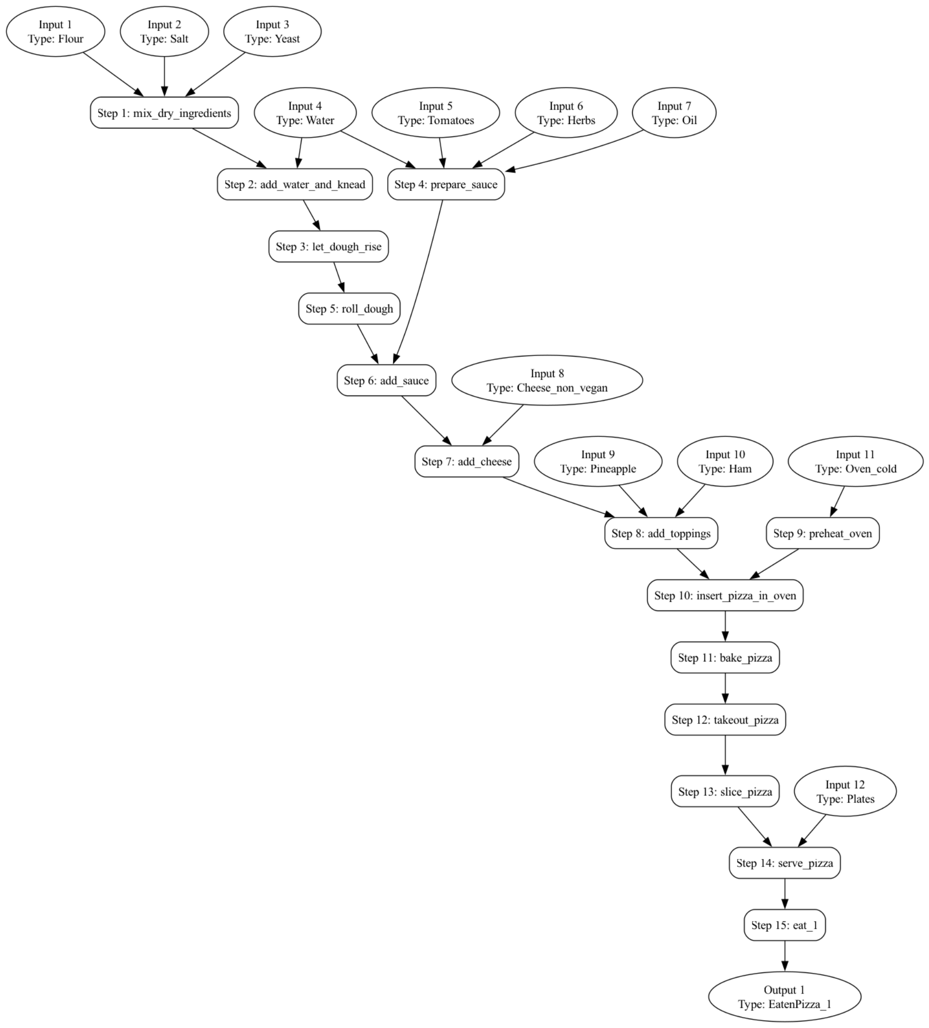

Fig. 1. One of 10 composed workflow candidates at step 15

B ASP Encodings

Listing 1.2. ASP snippet for #program step(t)

```
#program step(t).

% 1. Choose tool
1 = { use_tool(Tool, t) : term_tool(Tool) }.

% 2. Tool taxonomy propagation (upward)
use_tool(Parent, t) :- use_tool(Child, t), tool_tax(Child, Parent).

% 3. Tool taxonomy propagation (downward)
1 { use_tool(Child, t) : tool_tax(Child, Parent) } 1 :-
                use_tool(Parent, t),
                not term_tool(Parent).

% 4. Choose tool mode
1 { use_mode(Tool, Mode, t) : tool_mode(Tool, Mode) } 1 :-
                use_tool(Tool, t), term_tool(Tool).

% 5. Generate tool input
1 { in((t, Ix), D1, D2) : mode_input(Tool, Mode, Ix, D1, D2) } 1 :-
                use_mode(Tool, Mode, t),
                has_input(Tool, Mode, Ix).

% 6. Generate tool output
1 { out((t, Ix), D1, D2) : mode_output(Tool, Mode, Ix, D1, D2) } 1 :-
                use_mode(Tool, Mode, t),
                has_output(Tool, Mode, Ix).
```