Electronic Communications of the EASST
Volume 078 (2019)

8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation

-

Doctoral Symposium and Industry Day, 2018

Workflow Discovery with Semantic Constraints: The SAT-Based Implementation of APE

Vedran Kasalica, Anna-Lena Lamprecht

25 pages

Guest Editors: Falk Howar, Anna-Lena Lamprecht
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Workflow Discovery with Semantic Constraints: The SAT-Based Implementation of APE

Vedran Kasalica, Anna-Lena Lamprecht

Department of Information and Computing Sciences
Utrecht University, 3584 CC Utrecht, Netherlands

Abstract: Science today is increasingly computational, and many researchers regularly face the need of creating purpose-specific computational pipelines for their specific data analysis problems. The manual composition and implementation of such workflows regularly costs valuable research time. Hence, many scientists wish for a system that would only require an abstract description of their intended data analysis process, and from there automatically compose and implement suitable workflows.

In this paper we describe APE (the Automated Pipeline Explorer), a new implementation of a synthesis-based workflow discovery framework that aims to accomplish such automated composition. The framework captures the required technical domain knowledge in the form of tool and type taxonomies and functional tool annotations. Based on this semantic domain model, the framework allows users to specify their intents about workflows at an abstract, conceptual level in the form of natural-language templates. Internally, APE maps them to a temporal logic and translates them into a propositional logic instance of the problem that can be solved by an off-the-shelf SAT solver. From the solutions provided by the solver, APE then constructs executable workflow implementations.

First applications of APE on realistic scientific workflow scenarios have shown that it is able to efficiently synthesize meaningful workflows. We use an example from the geospatial application domain as a running example in this paper.

Keywords: scientific workflows, computational pipelines, automated workflow composition, program synthesis, workflow synthesis, semantic domain modeling, temporal logics, SAT solving

1 Introduction

Contemporary science across all disciplines is increasingly computational, and many scientists regularly face the need of producing software themselves to become able to solve their specific data analysis problems. Many of these programs are essentially *computational pipelines*, that is, sequences of calls to existing computational components, where the new program is mainly responsible for the coordination of the flow of data between them. Scientific workflow management systems (popular examples include Apache Taverna [WHF+13], Kepler [I+04] and Pegasus [DSS+05], but several more exist and are being developed [LPV+15, AGMT17, SWS]) support researchers in assembling computational components into complex scientific workflows, and facilitate their execution and monitoring directly within the same framework. However, they

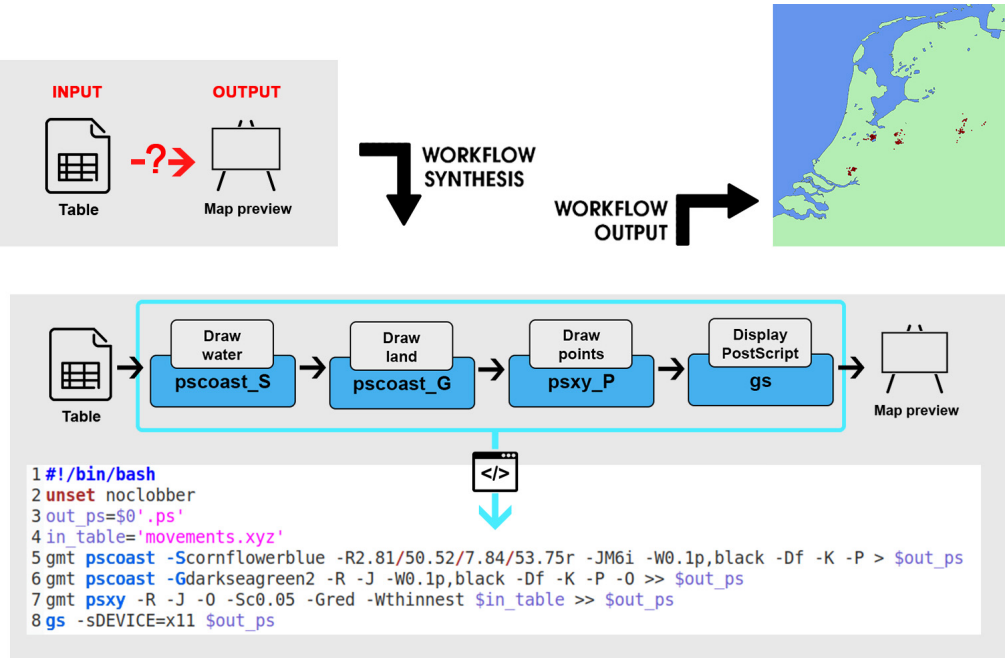

Figure 1: Synthesis-based discovery and composition of a geospatial analysis workflow.

typically require the users to know which components to use and connect to solve the specific problems, which connections are possible with regard to the compatibility of input and output data types and formats, and other kinds of technicalities.

Semantics-based automated workflow composition techniques strive to assist users in the discovery and composition of purpose-specific workflows [CSG+03, MPM+05, KBBG08, LMS09]. Ideally, users would only need to state their intents about the workflow at an abstract, conceptual level (e.g. by providing information about the available inputs and intended output data, or particular kinds of operations to use or avoid), and the workflow environment would automatically translate the specification into a concrete executable workflow (as illustrated in Figure 1). This is essentially a case of *program synthesis* (considered to be one of the central problems in theory of programming [PR89]), which, in the general formulation, aims to find a program that meets a given specification. In fact, with the PROPHETS framework for loose programming [NLS12, LNMS10] previous work has already demonstrated how program synthesis techniques, concretely the process synthesis approach of Steffen et al. [SMF93, FSMZ95, SMB97, MS07], can be used for semantics-based, constraint-driven automated composition of scientific workflows [PLIS18, ALM16, Lam13].

PROPHETS is a plugin to the jABC modeling framework for eXtreme model-driven development (XMDD) [SMN+07, MS12]. It allows workflow developers to mark connections between workflow building blocks as "loosely specified" and run the synthesizer to turn the loose specification into a fully specified and executable workflow part. Users can formulate additional constraints for the loose specifications that the synthesizer takes into account. Therefore PROPHETS provides a constraint editor with natural-language constraint templates, which the

users can easily fill with terms from a domain-specific controlled vocabulary. This allows users to very flexibly influence the synthesis results on a domain-specific, semantic level of abstraction.

As the development of jABC has been discontinued and superseded by the work on the Cinco SCCE Meta-Tooling Suite [NLKS17, NNMS16], we needed a replacement for PROPHETS to continue our work on synthesis-based automated discovery and composition of scientific workflows. Inspired by PROPHETS and based on the lessons learned with its application to scientific workflows, we started to develop APE[1] (the Automated Pipeline Explorer) as a workflow synthesizer tailored to scientific applications. Two observations were central: 1) PROPHETS' loose programming approach supports the development of workflows with complex control-flows. However, the computational workflows created in scientific applications are typically pipelines that do not require these complex dependencies. In addition, the pipelines generally do not involve concepts such as branching or loops. Therefore the ability to synthesize linear workflows is sufficient in most cases, and we decided to focus on that. As we will discuss further below, this simplifies specification and is advantageous for the runtime performance of the synthesizer. 2) The close integration with the full jABC framework made it difficult to connect PROPHETS to the software ecosystem of the eScience community. Thus, in order to facilitate uptake by practitioners, the new implementation aims to simplify import of semantic domain knowledge and export of synthesized workflows in formats that are commonly used in the eScience community, and provides its functionality as both and API and command line interface.

In this paper, we describe the SAT-based implementation of workflow synthesis in APE. We provide background information and survey related work about program synthesis and automated composition of scientific workflows in Section 2. Then Section 3 describes the modeling of domain knowledge and user intent in APE, which together form the workflow specifications. Section 4 explains how the specification is encoded as a SAT problem that can be given to an off-the-shelf solver. Section 5 describes and discusses a realistic application in the field of geovisualization [KL18, KL19], before Section 6 concludes the paper.

We use the geovisualization application as a running example throughout the paper. The goal in that application is to create a topographic map depicting bird movement patterns in the Netherlands, by combining existing geovisualization tools that perform the required operations. We use the tools from the popular GMT (Generic Mapping Tools) [WS91] collection of command line tools, which perform various geospatial operations. With a semantic description of the available input data, the desired output and additional workflow constraints, APE discovers and composes workflows from the tools that perform the given task. Figure 1 shows an example of a possible workflow automatically created by the synthesizer.

2 Background and Related Work

Unlike typical compilers that translate well-defined high-level languages to machine code, using sets of syntactical rules, program synthesis is typically accomplished by performing some type of search over the search space of programs consistent with the specification, usually resulting in more than one possible solution. Two major challenges in program synthesis are the state explosion of the search space [Val98, LV13], caused by the combinatorial nature of the problem,

[1] https://github.com/sanctuuary/APE

and the correct interpretation of the user intent. Both are non-trivial problems that were tackled from different angles throughout the years. This resulted in development of various synthesis techniques [BJ13a, GPS17], as well as application to many different domains. According to Gulwani [Gul10], each synthesis approach can be characterized by three essential dimensions: 1) the way the *user intent* is being provided, 2) the *search space* of the candidate programs that it searches and 3) the *algorithm* used to perform the search.

User intent is an obvious choice for a key characteristic of synthesis approaches. It defines the interaction between the user and the synthesis framework, essential for the applicability of the approach. The main goals of modeling the user intent are to have an intuitive way of providing the specification (what is considered intuitive typically depends on the targeted users) and to remove ambiguities in the specification.

Early synthesis approaches relied on the existence of a complete and formal specification of the program. Some approaches used theorem provers to construct a proof of the user specification, and the logical program itself [Gre69, MW71], while others used program transformations over abstract program specifications to produce the desired low-level programs [MW75]. However, providing the initial specification proved to be as complex as writing the program itself. This led to a focus shift [Smi75, SSG75, Sum86] from deductive program specification to inductive specifications, such as input-output examples, partial specification, etc. It has also become common practice to have an interactive loop between the user and the synthesis algorithm, where the user, based on the provided candidate solutions in the previous step, can provide additional examples or specification constraints in order to resolve ambiguities.

Many current approaches use *formal logic descriptions* to provide the user intent [SGF10]. It is used to describe the logical relation between the program input and the program output. However, these types of specifications are usually quite hard to construct, as they require the user to be familiar with the underlying logic. Although the work presented in this paper falls under this category, similarly to PROPHETS, it uses natural-language templates for providing the specification, and so accounts for users not trained in logics or formal specification. This type of user intent would, according to [Gul10], be categorized between logical and natural language specification.

Although the formal specification allows for a quite accurate description of the user intent, end users might not find it intuitive and straightforward. To solve this issue, some approaches focus on an *example-based specification* format [Gul16, Gul11] to model the user intent. This type of problem specification allows users to provide examples of desired outputs based on given inputs.

Some approaches allow for a *partial description* of the program as part of the specification of the user intent. This approach is also called *sketching* [SL08]. Loose programming as in PROPHETS is another example of the same underlying idea. As an example from the eScience community, the WINGS (Workflow Instance Generation and Selection) framework [GRKo11] employs the idea of providing a workflow template, where individual steps can be left out and are automatically included based on the context when the template is instantiated.

Finally, programmers might sometimes consider a programming language as the best means for specifying their intent. This is applied in superoptimization of code [PTBo16, GJTV11] and in the synthesis of program inverses [Dij79], such as compression/decompression, encryption/decryption, etc.

The search space is defined by the structures that can be provided as synthesis output, as well as by the restrictions made on the problem implementation. Furthermore, its size and complexity is crucial for the computational complexity of the synthesis problem. The search space should keep balance between the expressiveness of the framework and efficiency of a search over it. In other words, it should be comprehensive enough to support a large set of candidate programs, and at the same time restrictive enough to support efficient search mechanisms. Thus, synthesis approaches tend to limit the search space in some way in order to improve their runtime performance.

In practice, the search space can vary from programs in general programming languages to such in domain-specific formalisms. It is defined by the supported *operators* and *control structures*. Our approach targets programs that restrict control structure to linear/sequential programs, also referred as *loop-free programs*. Another such approach is the previously mentioned superoptimization approach [GJTV11], implemented using SMT solvers. Loop-free programs can express a wide range of computations, such as text-editing programs [MW92, LDW00], API call sequences [MXBK05], and unbounded data type manipulations [KMP+10].

Other approaches allow the user to provide a skeleton (grammar) of the space of possible programs in addition to the specification [ABJ+13]. As the grammar provides structure for the hypothesis space, these approaches can yield more efficient search procedures. Additionally, a strict grammar ensures better interpretability of the candidate solutions. Examples of such approaches include the aforementioned *Sketch* [SL08] and WINGS systems, the looping templates described in [SGF10], etc.

The search technique can be based on enumerative search, deduction, constraint solving, statistical techniques, or a combination of these. Our approach uses constraint solving techniques, also categorized as *logical reasoning based techniques*. The main idea is to reduce the synthesis problem to a SAT problem, and then use an off-the-shelf SAT solver to explore the search space. The reduction typically involves two steps: *constraint generation* and *constraint solving*. The constraint generation procedure involves the generation of the logical constraints, such as the logical relation between inputs and outputs, whereas resolving of the constraints yields the desired program. The latter involves the translation of the generated logical constraints into the corresponding SAT constraints and the usage of the SAT solver as the synthesis reasoner. Counterexample-guided inductive synthesis (CEGIS), as used in the Sketch system [SL08], is another popular constraint solving method.

3 Domain Knowledge and User Intent

As discussed in the previous section, one of the two main challenges in program synthesis is accurately capturing the *user intent*, that is, the specification of the desired program. We follow the modeling framework introduced by Steffen et al. [SMF93, FSMZ95, SMB97, MS07] for this purpose, as it has proven to be suitable and effective in practice for similar applications [PLIS18, Lam13, ALM16]. It comprises two main parts:

- **Domain Knowledge**: *Taxonomies* of types and tools as controlled domain vocabulary, and the semantic *annotation of tools* with them (see Sections 3.1 and 3.2).

- **User Intent**: Logical constraints that describe intended workflow(s) using terms from the domain model (see Section 3.3).

The idea is that experts from the application domain are in charge of modeling the domain knowledge, and that this modelling allows the end users (workflow developers) to simply use the domain knowledge as vocabulary to express their intents.

3.1 Modeling Domain Knowledge: Taxonomies

To properly capture user intent, it is essential to use clear and precise terminology. It should be abstract enough not to require users to be familiar with the actual tools used in the domain, and still concrete enough to accurately capture the desired goal. In order to support different levels of abstraction, terms for domain tools and data types are organized in *taxonomies*, tree-like structures composed of semantic *tools* and *data types*, respectively.

A taxonomy $\mathcal{T} = (C, A, \rightarrow)$ is an acyclic directed graph, where C represents a set of concrete elements from the domain (a concrete tool, a concrete data format, etc.), A represents a set of conceptual elements, or classes, used to provide abstraction over the concrete elements, and $\rightarrow$ is a relation *is_a* over the two sets, specifically, over concrete and conceptual elements $c \rightarrow a$, where $c \in C, a \in A$, or pairs of conceptual elements $a_1 \rightarrow a_2$, where $a_1, a_2 \in A$.

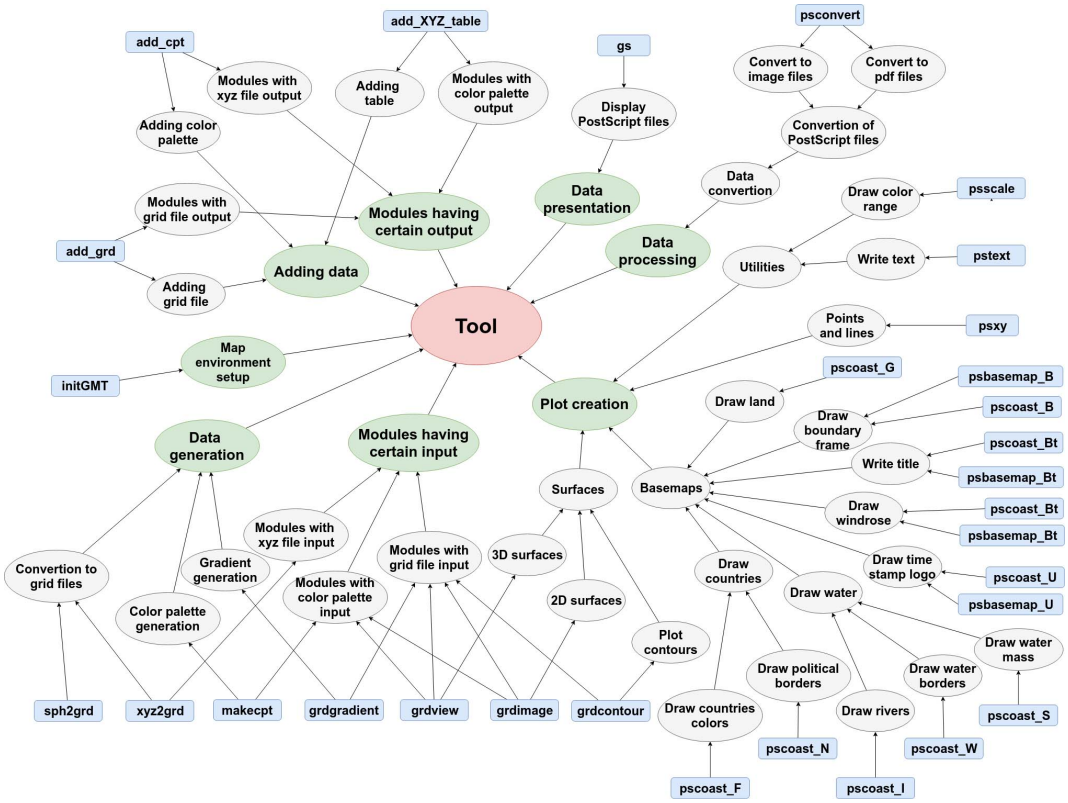

Figure 2: Tool Taxonomy for the Geovisualization Use Case Scenario

The tool taxonomy $\mathcal{T}_M = (C_M, A_M, \rightarrow_M)$ shown in Figure 2 shows a tool classification from the geovisualization case study, where C_M is a set of concrete tool implementations from the domain (rectangles) and A_M is a set of abstract/conceptual tools from the domain (ellipses) that represent groups of concrete tools that share common properties. Edges in the figure represent the $\rightarrow_M$ (*is_a*) relations. The concrete tools (leaves in the presented taxonomy) correspond to the most specific operations, technically the executable units in the domain. For example, the tool *pscoast_s* in the lower right corner of the tool taxonomy is a command line operation used to automatically color water surfaces on a map. The *is_a* relation abstracts from the actual tool and allows the user to refer to it as a conceptual tool that *Draws water mass* (we also say that *pscoast_s* is a *subtool* of *Draws water mass*). Note that there is not necessarily a one-to-one correspondence between concrete tools in the taxonomy and real-world tools that implement them. In some cases, several tools in the taxonomy refer to the same executable tool (*polymorphism*). For example *pscoast_B*, *pscoast_Bt*, *pscoast_U* and *pscoast_Td* all call the *pscoast* command line tool, but with different parameters, causing it to perform different operations.

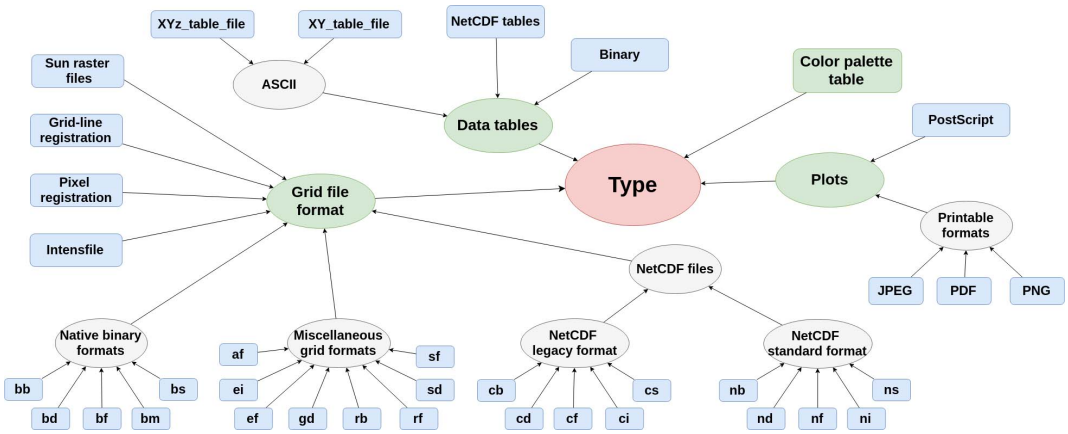

Figure 3: Type Taxonomy for the Geovisualization Use Case Scenario

The type Taxonomy $\mathcal{T}_T = (C_T, A_T, \rightarrow_T)$ shown in Figure 3 is defined in the same way, describing the data types in the geovisualization case study. C_T is a set of concrete types that directly correspond to data instances that are used by domain tools. A_T is a set of abstract types that represent groups of concrete types. The term *subtype* corresponds to the definition of *subtool*.

Note that in some domains, data are classified according to multiple disjoint criteria. We refer to them as *dimensions*. For example, in an application from the bioinformatics domain [PLIS18], inputs and outputs are described by two dimensions of data, namely types and formats. The data type describes semantic properties of a data instance, while the data format is about the syntactic representation. APE supports annotation of such domains using disjoint dimension taxonomies, but for simplicity we focus on a simple, one-dimensional type taxonomy in this paper.

Technically, we use a subset of the W3C Web Ontology Language (OWL) [AH04] to represent the taxonomies. OWL is a well-known Semantic Web language designed to represent ontologies, which has become the de facto standard for ontologies in many domains. Major domain ontologies, such as the EDAM data and methods ontology in bioinformatics [IKJ+13], are provided in OWL and can thus directly be used in our framework. OWL can be used to describe complex

relations between classes, but our framework only uses the concepts and concept inclusions (i.e., only the taxonomy part of the OWL file) to define and classify the taxonomy elements.

3.2 Modeling Domain Knowledge: Tool Annotations

In order to be able to combine concrete tools, the synthesizer needs to know how these tools operate over data types. In practice, tools can perform complex operations over data types and implement various transformations on them. However, at the semantic level, we abstract these relations into two basic functions, known from data-flow analysis:

- $use(\cdot) : C_M \to \mathcal{P}(C_T)$ – in order for a tool x to be executed, elements of the set of types $use(x)$ must be available
- $gen(\cdot) : C_M \to \mathcal{P}(C_T)$ – after execution of a tool x, elements of the set of type $gen(x)$ are available

The two functions can also be referred to as $input(\cdot)$ and $output(\cdot)$ respectively, which is more natural terminology for computational tools. Table 1 lists a selection of concrete tools from the geovisualization case study, each with its name, function description and its (possibly empty) sets of input and output types. In practice it has proven to be reasonable to only use the "payload" inputs/outputs in the annotation, and not all parameters that a tool might have. The latter tends to blow up the search space without actually being helpful to find new meaningful solutions.

Name	Description	Type in (uses)	Type out (gen)
add_grd	Provide a grid file		NetCDF
grdgradient	Compute directional gradient	NetCDF	Intensfile
makecpt	Make color palette tables	cpt_file	cpt_file
...			
grdview	3-D imaging of 2-D gridded data	NetCDF, cpt_file	PostScript
pscoast_W	Draw water borders	PostScript	PostScript
initGMT	Setup the GMT map environment		PostScript
gs	Tool used to display graphical files	Plots	

Table 1: Annotation of concrete tools in Geovisualization Use Case Scenario

Technically, we use a JSON representation for the tool annotations that follows the JSON representation of tool function applied in the bio.tools registry [IRMo16]. For each tool function we annotate its name and ID, the operation(s) that it performs, a set of inputs, a set of outputs and a command that corresponds to the tool execution. The last information is crucial for automated implementation of the workflow. The current version of APE supports simple shell commands only, resulting in a shell script workflow implementation. However, we plan to extend it to support other commonly used languages, such as CWL [ACT+16], NextFlow [DCFo17], etc.

3.3 Modeling User Intent: Temporal Constraints

We use *Semantic Linear Time Logic* (SLTL) [SMF93, FSMZ95], an extension of the well-known Linear Time Logic (LTL), as a flexible and intuitive language for modeling the user intent.

However, in order to allow non-experts to proficiently use the formalism, we provide a natural-language templates to complement the formal representation.

Definition 1 The syntax of SLTL can be described by the following BNF:

$$\Phi ::= true \mid type(t_c) \mid \neg\Phi \mid \Phi \wedge \Phi \mid \langle m_c \rangle \Phi \mid \mathbf{G}\Phi \mid \Phi \mathbf{U} \Phi$$

where t_c and m_c represent constraints over types and tools, respectively.

In our implementation, type and tool constraints are limited to using propositions that correspond to concepts from the type and tool taxonomies. SLTL formulas are interpreted over all legal paths, that is, as we explain further in Section 4, in our case type-correct alternating sequences of types and tools. The sequences start and end with types. A formal definition of SLTL is available in [SMF93, FSMZ95]. Intuitively, the constructs mean the following:

- $true$ is satisfied by every path.
- $type(t_c)$ is satisfied by every path whose first element (a type) satisfies the t_c type constraint.
- $\neg$ (negation) and $\wedge$ (conjunction) are interpreted in the usual fashion.
- $\langle m_c \rangle \Phi$ is satisfied by paths whose second element (i.e., the first tool in the sequence) satisfies the tool constraint m_c and whose *first suffix subpath*[2] satisfies Φ. This modal operator represents a parameterized version of the *next-time* operator of LTL. Thus, we use $\mathbf{X}\Phi$ to denote $\langle \texttt{true} \rangle \Phi$, i.e. paths whose first suffix path satisfies Φ, disregarding the initial tool in the sequence.
- $\mathbf{G}\Phi$ is satisfied if Φ is satisfied globally, i.e. for every suffix subpath of the path.
- $\Phi \mathbf{U} \Omega$ is satisfied if property Φ holds in the first n suffix subpaths, where $n \in \mathbb{N}$, until a suffix subpath is reached, that satisfies the property Ω. The until operator is interpreted as *strong until*, thus it guarantees that the property Ω holds eventually. The operator *finally* ($\mathbf{F}\Omega$) is used to simplify a commonly used notion $\texttt{true}\,\mathbf{U}\,\Omega$.

The language is quite expressive and can accurately capture the user intent. However, to be used proficiently, it requires the user to be familiar with the underlying temporal logic. Considering that we do not expect that from our users, the framework provides natural-language templates that correspond to SLTL formulas. The user works with natural-language cloze texts, combined with the terms from the taxonomies, that will be automatically transformed into the underlying logic. Based on our previous experience with PROPHETS and its practical use cases, we have defined a set of most commonly used templates for describing the user intent (see Table 2). A simple example of such a constraint would be "Use tool *gs* in the solution." (see **T5** in Table 2) where *gs* is a tool defined in the tool taxonomy (see Figure 2).

[2] A *suffix subpath* is the part of the original path that is obtained by removing the first n pairs of types and tools (2n elements) from it, where $n \in \mathbb{N}$. First suffix subpath is a suffix subpath where $n = 1$.

ID	Constraints in natural-language	Constraints in SLTL
T1	If tool Tool_1 is used, tool Tool_2 has to be used subsequently	G(¬⟨Tool_1⟩ true | X F ⟨Tool_2⟩ true
T2	If tool Tool_1 is used, tool Tool_2 cannot be used subsequently	G(¬⟨Tool_1⟩ true | X G ¬⟨Tool_2⟩ true
T3	If tool Tool_1 is used, tool Tool_2 must have been its direct predecessor in the sequence	Not expressible in SLTL
T4	If tool Tool_1 is used, tool Tool_2 has to be used next in the sequence	G(¬⟨Tool_1⟩ true | X ⟨Tool_2⟩ true
T5	Use tool Tool_1 in the solution	F ⟨Tool_1⟩ true
T6	Do not use tool Tool_1 in the solution	G ¬⟨Tool_1⟩ true
T7	Use Tool_1 as last tool in the solution.	F ⟨Tool_1⟩ true & G(¬⟨Tool_1⟩ true | ¬XX true)
T8	Use type Type_1 in the solution	F Type_1
T9	Do not use type Type_1 in the solution	G ¬Type_1

Table 2: User Intent: Natural-language templates for SLTL formulas

We are aware of the limitations that the predefined templates present, and thus one of our future goals is to provide support for arbitrary SLTL formulas. The main target for this extension are not the end users, but rather the domain experts in charge of modeling the domain knowledge. This would allow them to define further, possibly domain specific natural-language templates and map them to the respective SLTL formulas, as also possible in PROPHETS. This way the framework would not be limited by a predefined and fixed set of constraint templates, while keeping the abstraction layer over the underlying logic for the end users.

4 Encoding as SAT Problem

This section relates to the second challenge in program synthesis, the state explosion of the search space. As it is sufficient for our workflow discovery to produce sequential computational pipelines, we reduce the search space to loop-free programs, with predefined control flow structure. Regarding the search technique dimension of the program synthesis [Gul10], the approach uses logical reasoning-based techniques. The motivation behind this type of approach is to reduce the problem to a SAT instance and benefit from state-of-the-art solving technologies.

To encode the synthesis problem as SAT instance we incorporate some well known ideas from planning as satisfiability [KS92, KS96]. Our main idea is to provide an encoding of the general workflow structure, which is further enhanced with propositional constraints that correspond to the domain model and user intent. Then, an off-the-self SAT solver can be used as a reasoning engine. This section covers each of the mentioned steps. First, we describe the encoding of the general workflow structure in propositional logic. Second, we discuss the encoding of the domain model, i.e. the encoding of the taxonomy structure and input and output dependencies. Third, we explain the propositional encoding of the temporal constraints that correspond to the user intent. Finally, we present the constraint solving and the output format of our approach.

4.1 Encoding the Workflow Structure

Bounded Model Checking [BCC+03] has proven to be very successful in practice and to make verification problems tractable. We use a similar approach and reduce Bounded Workflow Synthesis to a SAT problem. To encode our problem in propositional logic and use techniques presented in [BCC+03], our search space has to be bounded and well-structured. In other words, we need to find a propositional encoding that would be flexible enough to capture different workflow scenarios, and still structured enough to provide unambiguous solutions when mapped back to the workflow structure. Although bounding the length of the solution limits the solutions that can be found to a specific length, iteratively increasing the bound and creating an encoding for each of them allows for the exploration of workflows of any length.

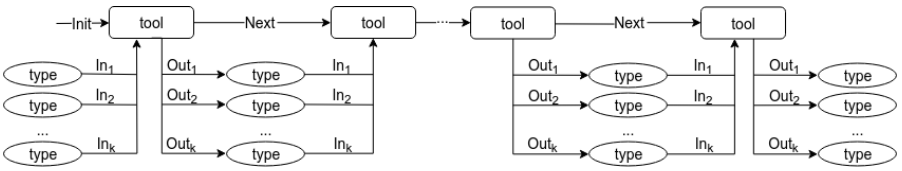

Figure 4: Message passing design

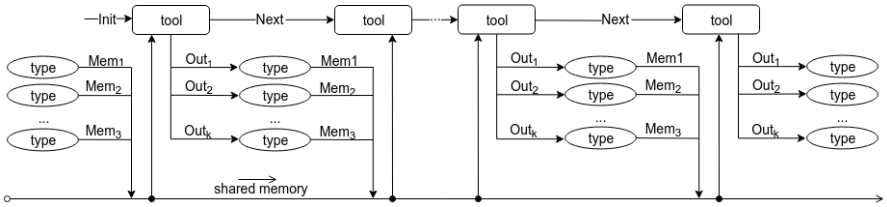

Figure 5: Shared memory design

Our focus are computational pipelines, that is, linear workflows that represent the sequential execution of tools with no explicit branching on the control-flow level. We distinguish two main workflow structure designs: 1) The *message passing* design, where data can be transferred exclusively between consecutive tools in the workflow. 2) The other structure supports the more commonly used *shared memory* design, where each tool can access the data created by any of the preceding tools. The designs follow the structure presented in Figs. 4 and 5, respectively[3]. In the rest of the paper, the encoding and examples will use the *shared memory* design (see Figure 5). The transition from one design to the other is straightforward.

As mentioned before, APE does not support loops, rather we assume that each loop can be flattened into a repetitive sequence of tools. Therefore, we restrict the framework to finite sequences of states and do not support infinite behaviour of the system. Particularly, three main points justify this decision. First, based on our experience with computational pipelines in different scientific application domains, in practice loops tend to be trivial enough to be flattened into

[3] Note that the presented structures of the program differs from the mentioned SLTL definition of the path, which consisted of alternating single type and tool elements. However, if we abstract each set of types that is created in single step (e.g. the initial k inputs) into an element, we can obtain the mentioned path structure.

sequential executions. Second, as suggested in [BCC+03], infinite behaviour could be supported through implementation of loops in our system. However, this would require a more complex encoding and ultimately increase the reasoning time. Third, trivially allowing loops in the structure would create redundant solutions for the user, such as looping over tools with no output. Solving this problem would require to properly capture the related user intent. In addition, heuristics would need to be introduced for ranking looping sequences. These are not trivial tasks.

For our initial encoding, we need a formula that captures all workflows of length n ($n \in N$), where n is a bound of our workflow. Let k be the biggest output type index among the domain tools, that is, the biggest number of outputs per tool, $Next$ and Out_x the transition relations of our system as a propositional formula, Mem_x a predicate over the type variables defining existence of an instance in the memory and $Init$ a predicate over the module variables defining the initial state of the workflow. A workflow of length n is encoded by the formula:

$$[\![W]\!]_n := Init(m_1) \wedge \bigwedge_{i=2}^{n} Next(m_{k-1}, m_k) \wedge \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} Out_j(m_i, t_i^j) \wedge \bigwedge_{i=0}^{n-1} \bigwedge_{j=1}^{k} Mem_i(t_i^j) \tag{1}$$

The *tool states* are represented as $m_1, ..., m_n$, where $Init(m_1)$ defines m_1 as a first tool in a sequence and $Next(m_{k-1}, m_k)$ defines m_{k-1} and m_k as consecutive tools in execution. The *type states* are represented as $t_0^1, ..., t_n^k$, where $Out_j(m_i, t_i^j)$ encodes type state t_i^j as output state of the tool in tool state m_i. In addition, $Mem_i(t_i^j)$ labels the type used in type state t_i^j as available in memory to be used by the tools in states m_x, where $x > i$. Furthermore, type states $t_0^1, ..., t_0^k$ represent the input provided to the workflow, while type states t_n^1, t_n^k represent outputs from the last tool, hence the desired final output of the workflow. We refer to it as final output, considering that the intermediate tools can still produce outputs that are available to the user, as long as we adhere to the shared memory design.

Example 1 *Our goal is to synthesize a workflow whose length is bounded to $n = 2$ and the biggest domain output type index is $k = 2$. The first step of the encoding is the workflow structure, which is as follows:*

$$\begin{aligned}[\![W]\!]_2 := {} & Init(m_1) \wedge Next(m_1, m_2) \wedge Out_1(m_1, t_1^1) \wedge Out_2(m_1, t_1^2) \wedge \\ & Out_1(m_2, t_2^1) \wedge Out_2(m_2, t_2^2) \wedge Mem_0(t_0^1) \wedge Mem_0(t_0^2) \wedge \\ & Mem_1(t_1^1) \wedge Mem_1(t_1^2) \wedge Mem_2(t_2^1) \wedge Mem_2(t_2^2)\end{aligned}$$

4.2 Encoding the Domain model

Once the initial structure has been encoded, we define the rules that translate our domain knowledge into a set of propositional formulas. This includes preserving input and output types for each tool, preserving the classifications defined by the taxonomy and ensuring that none of the states in our encoded transition system violates the intended structure, that is, ensure that each state that corresponds to a tool (or type) is represented by exactly one tool (or type) predicate. These constraints ensure that our structure can be unambiguously mapped to exactly one workflow representation. The rest of this section presents the encoding of the domain model.

Preserving tool inputs. Let n be the workflow bound and k the biggest output type index. In order for our (shared memory) structure to preserve tool input relation, for each tool X and for each type $Y \in in(X)$ we define the formula:

$$[\![In(X)]\!]_n := \bigwedge_{i=1}^{n} \left(X(m_i) \implies \bigvee_{p=0}^{i-1} \bigvee_{q=0}^{k} Y(t_p^q) \right) \tag{2}$$

The formula encodes a condition where the usage of the tool X in a certain tool state m_i requires the type Y to be provided in a type state t_p^q prior to it, i.e. $Mem_p(t_p^q)$ where $p < i$). Notice that for the first tool (tool state m_1), the only prior type states are the workflow inputs encoded as $Mem_0(t_0^1) - Mem_0(t_0^k)$ in (1).

Example 2 *We extend Example 1 with the encoding of tool inputs. To simplify the example we present the encoding of a single tool input, using the tool makecpt (see Table 1), which makes a 'color palette tables' and the input type it requires is a 'cpt_file'. The encoding is as follows:*

$$\begin{aligned}[\![In(makecpt)]\!]_2 := &\left(makecpt(m_1) \implies cpt\_file(t_0^1) \vee cpt\_file(t_0^2) \right) \wedge \\ &\left(makecpt(m_2) \implies cpt\_file(t_0^1) \vee cpt\_file(t_0^2) \vee \right. \\ &\qquad \left. cpt\_file(t_1^1) \vee cpt\_file(t_1^2) \right)\end{aligned}$$

Preserving tool outputs. The following set of formulas encodes the preservation of the tool output relations. Let n be the workflow bound and k the biggest output type index. For each tool X and list of types $Y_1, .., Y_p$, where $p \leq k$ and $Y_i \in out(X)$, $\forall i \in [1, p]$ we define the formula:

$$[\![Out(X)]\!]_n := \bigwedge_{i=1}^{n} \left(X(m_i) \implies \bigwedge_{j=0}^{p} Y_j(t_i^j) \bigwedge_{j=p+1}^{k} empty(t_i^j) \right) \tag{3}$$

The formula encodes a condition where the usage of the tool X in a tool state m_i, requires the types $Y_1, .., Y_p$ to be used in the type states that follow $t_i^0 - t_i^k$, i.e. type states encoded as $Out_j(m_i, t_i^j)$, for $j \in [0, k]$ in (1). In case that p is smaller than the possible number of outputs, the rest of the type states are empty.

Example 3 *We extend Example 2 with the encoding of tool outputs. In order to simplify the example we show the encoding of a single tool input for the mentioned tool makecpt (see Table 1). The tool has one output type, which is cpt_file. The encoding is as follows:*

$$\begin{aligned}[\![Out(makecpt)]\!]_2 := &\left(makecpt(m_1) \implies cpt\_file(t_1^1) \wedge empty(t_1^2) \right) \wedge \\ &\left(makecpt(m_2) \implies cpt\_file(t_2^1) \wedge empty(t_2^2) \right)\end{aligned}$$

Preserving taxonomy classification. In order to preserve a classification provided by the taxonomy, we introduce a set of formulas that encode the dependency between tools and types and their subclasses. In other words, the encoding implies that once a conceptual tool/type has been used in a state, at least one of its subtools/subtypes needs to be used as well, and vice versa. For each conceptual element X in the tool taxonomy and list of its subtools $Y_1,..Y_p$, such that $Y_i \rightarrow_M X, \forall i \in [1,p]$ we define the following formula:

$$[\![Tax(X)]\!]_n := \bigwedge_{i=1}^{n} \left(\left(X(m_i) \implies \bigvee_{j=1}^{p} Y_j(m_i) \right) \wedge \left(\bigvee_{j=1}^{p} Y_j(m_i) \implies X(m_i) \right) \right) \tag{4}$$

The first part of the formula enforces usage of at least one of the subtools of X in a certain state, providing that X was used in that state as well. The second part of the formula enforces usage of the tool X, providing that at least one of its subtools is used in the same state. The type taxonomy classification (not shown) is encoded in exactly the same manner.

Example 4 *We extend Example 3 with the encoding of tool and type taxonomies. To keep the example simple we show the encoding of a subtaxonomy that consists of an abstract tool Write title and its two subtools pscoast_Bt and psbasemap_Bt (see the right-hand side of Figure 2). For simplicity of the notion the tools are abbreviated as WT, C_Bt and B_Bt, respectively. The encoding is as follows:*

$$\begin{aligned}
[\![Tax(WT)]\!]_2 := & \left(WT(m_1) \implies C_Bt(m_1) \vee B_Bt(m_1) \right) \wedge \\
& \left(C_Bt(m_1) \vee B_Bt(m_1) \implies WT(m_1) \right) \wedge \\
& \left(WT(m_2) \implies C_Bt(m_2) \vee B_Bt(m_2) \right) \wedge \\
& \left(C_Bt(m_2) \vee B_Bt(m_2) \implies WT(m_2) \right)
\end{aligned}$$

Enforcing state correctness. In order to ensure that each solution provided by the solver corresponds to exactly one workflow structure, we need to encode two types of rules. Let n be the workflow bound and k the biggest output type index. First, we ensure that for each state in the transition system, the correct kind of element (tool or type) is being used:

$$[\![Exist]\!]_n := \bigwedge_{j=1}^{k} type(t_0^j) \wedge \bigwedge_{i=1}^{n} \left(tool(m_i) \wedge \bigwedge_{j=1}^{k} type(t_i^j) \right) \tag{5}$$

The formula enforces that a type needs to be used in each type state t_i^j, labeled as $Mem_i(t_i^j)$ in (1) and a tool in each tool state m_i, labeled as $Init(m_i)$ or $Next(m_{i-1}, m_i)$ in (1). In this scenario it is enough to require usage of the root elements of the taxonomies, due to the previous encoding of the taxonomy. Second, we have to avoid conflicts of using two different concrete tools or types

in the same state of the structure. Let n be the workflow bound. For each pair of concrete tools X_1 and X_2, we introduce the formula

$$[\![Conf(X_1, X_2)]\!]_n := \bigwedge_{i=1}^{n} \neg X_1(m_i) \vee \neg X_2(m_i) \tag{6}$$

to eliminate conflicts regarding the usage of multiple concrete tools simultaneously. The formula forbids usage of two different concrete tools in a single tool state. A similar encoding is used to eliminate overlapping usage of concrete types.

Example 5 *We extend Example 4 with the encoding of state correctness. To simplify the example we show the encoding of mutual exclusion of just two concrete tools, makecpt and gs (see Figure 2), but omitting the mutual exclusion of data types. The encoding is as follows:*

$$\begin{aligned}
[\![Exist]\!]_2 &:= type(t_0^1) \wedge type(t_0^2) \wedge tool(m_1) \wedge type(t_1^1) \wedge type(t_1^2) \wedge \\
&\quad tool(m_2) \wedge type(t_2^1) \wedge type(t_2^2) \\
[\![Conf(makecpt, gs)]\!]_2 &:= \big(\neg makecpt(m_1) \vee \neg gs(m_1)\big) \ \wedge \ \big(\neg makecpt(m_2) \vee \neg gs(m_2)\big)
\end{aligned}$$

4.3 Encoding the Temporal Constraints

The encoding of the workflow and the domain model is already sufficient to provide valid workflow solutions to being reasoned over. However, we do not only want to be able to generate any executable workflow, but specific workflows based on a specification of a concrete user intent. Thus, the next step is to enhance the encoding with user intent, provided as SLTL formulas.

In order to accomplish that, we need to provide a mechanism for transforming SLTL formulas into propositional logic. The transformation used in the implementation is based on the framework introduced in [BCC+03], which provides a mechanism for transforming arbitrary LTL formulas into propositional formulas. The paper distinguishes between transformations of LTL formulas that include loops in their path, and those that do not. As we are dealing with loop-free computational workflows here, we focus on the latter.

As mentioned earlier, currently our implementation is limited to the transformation of the most commonly used SLTL formulas into propositional logic, i.e. those that we also turned into constraint templates (see Section 3.3). We plan to develop a framework for the transformation of arbitrary SLTL formulas in the future, based on the framework of Biere et al. [BCC+03] for transformation of arbitrary LTL formulas.

Definition 2 Let n be the workflow bound and k the biggest output type index. The notion $[\![f]\!]_n^i$ refers to the interpretation of the SLTL formula f in the i-th state of the path of length n, where each state is a pair of type and tool elements. Translation of SLTL formulas into propositional

format is as follows:

$$[\![p]\!]_n^i := p(t_i^1) \vee \ldots \vee p(t_i^k) \qquad [\![\mathbf{G}f]\!]_n^i := [\![f]\!]_n^i \wedge [\![\mathbf{G}f]\!]_n^{i+1}$$
$$[\![\langle q\rangle true]\!]_n^i := q(m_{i+1}) \qquad [\![\mathbf{F}f]\!]_n^i := [\![f]\!]_n^i \vee [\![\mathbf{G}f]\!]_n^{i+1}$$
$$[\![\neg p]\!]_n^i := \neg p(t_i^1) \wedge \ldots \wedge \neg p(t_i^k) \qquad [\![\mathbf{X}f]\!]_n^i := [\![f]\!]_n^{i+1}$$
$$[\![\neg\langle q\rangle true]\!]_n^i := \neg q(m_{i+1})$$
$$[\![f \vee g]\!]_n^i := [\![f]\!]_n^i \vee [\![g]\!]_n^i$$
$$[\![f \wedge g]\!]_n^i := [\![f]\!]_n^i \wedge [\![g]\!]_n^i$$

Base case:

$$[\![f]\!]_n^n := 0$$

The translation rules are used to transform the user specification from the natural-language templates into propositional logic, based on their SLTL representation (see Table 1). The following example illustrates an SLTL transformation.

Example 6 *We extend Example 5 with the encoding of the SLTL formula* $\phi = \boldsymbol{G}\neg\langle grdview\rangle true$ *("Do not use tool grdview in the solution.", see* ***T6*** *in Table 2). The translation of* ϕ *to a propositional formula is as follows:*

$$[\![\boldsymbol{G}\neg\langle grdview\rangle true]\!]_2^0 := \neg grdview(m_1) \wedge [\![\boldsymbol{G}\neg\langle grdview\rangle true]\!]_2^1, \quad where$$
$$[\![\boldsymbol{G}\neg\langle grdview\rangle true]\!]_2^1 := \neg grdview(m_2) \wedge [\![\boldsymbol{G}\neg\langle grdview\rangle true]\!]_2^2, \quad where$$
$$[\![\boldsymbol{G}\neg\langle grdview\rangle true]\!]_2^2 := 0$$

This framework defines straightforward translations of the SLTL formulas. However, sometimes the encoding of commonly used formulas can be unnecessarily complex. One such example is the SLTL constraint that specifies the last tool in the solution program (see **T7** from Table 2). To express such a constraint in SLTL, e.g. "Use tool Y as the last tool in the solution", we would have to use few different modal operators, as follows:

$$[\![T_7]\!]_n := [\![\mathbf{F}\langle tool\_Y\rangle true \ \wedge\ \mathbf{G}(\mathbf{X}\,\mathbf{X} true \vee \langle tool\_Y\rangle true \vee \neg\mathbf{X} true)]\!]_n^0$$

or

$$[\![T_7]\!]_n := [\![\mathbf{G}\ \mathbf{F}\langle tool\_Y\rangle true\ ]\!]_n^0$$

However, considering that the workflow has a fixed bound, this type of constraints can be directly encoded in the structure, optimizing the encoding. Thus, APE rewrites the encoding into a simpler formula, where n is the bound of the workflow:

$$[\![T_7]\!]_n := tool\_Y(m_n)$$

Thus, an additional advantage of our current encoding is that the natural-language templates can be extended with some formulations that are not directly supported under SLTL, without

extending the formal language specification. Another such example is constraint **T3** from Table 2 that uses the temporal modal operator *previous* (inverse of *next*).

Finally, once the complete encoding is provided, it is sent to the MiniSAT solver [EEN05] to actually perform the synthesis. Based on the solutions provided by the solver, candidate workflows and their executable implementations are provided to the user. In this study, we perform the search for possible workflows until the first depths where solutions are found (the search depth is the same as the length of the solutions). Usually, the shortest solutions are also the most relevant with respect to the workflow specification, as they present the smallest number of steps necessary to satisfy it. In order to illustrate, if we look at our running example (see Examples 1 - 6) and assume that there is no initial input provided to the workflow, some of the proposed solutions would be:

- initGMT → gs
- initGMT → pscoast_W
- add_grd → makecpt

where the arrows denote the order in which the tools are being executed, i.e. the first solution suggests using tools *initGMT* and *gs* in a sequence. For the current specification, each solution is of length 2 and none of the solutions includes the tool *grdview*, due to the user intent constraint that excludes this tool (see Example 6). Additionally, each tool that is suggested as first in the sequence does not require any input, considering that we did not provide any initial workflow input. Similarly, the second tool is limited to the tools that require no input or the input that was provided as the output of the first tool. To illustrate, we will elaborate the first proposed solution. It represents a workflow that uses *initGMT* command to instantiate a GMT program and to generate an empty map, while the second command - *gs*, displays the generated map to the user. Although the workflow does not perform any notable computations, it is one of the smallest programs that satisfy the constraints presented in Examples 1 - 6. That is why an accurate specification of the program is as important as the program synthesis algorithm itself.

5 Geovisualization Application Example

In this section we describe the aforementioned geovisualization application in some more detail[4]. We discuss the quality of the provided candidate solutions and the runtime performance of the synthesis algorithm, which are key aspects for the applicability of synthesis frameworks in practice. A complete description of the scenario, where we elaborate on the context of the application and the steps taken to implement it, can be found in [KL19].

The geovisualization workflow use case is about creating a computational pipeline that produces a topographic map depicting waterbird movement patterns in the Netherlands (see **E4** in Figure 6). Wildlife tracking is an important method for biologists and scientists around the world to improve their understanding of animal behavior. The use case combines tools from the GMT

[4] https://github.com/sanctuuary/APE_UseCases/tree/master/GeoGMT

(Generic Mapping Tools) [WS91] collection with data from the Movebank [KCW+11] online database of animal tracking data.

The domain model consists of the two taxonomies described earlier (see Figs. 2 and 3) and the tool annotations as shown in Table 1). The example required five iterations of workflow specification and interactive refinement in order to produce the completely annotated topographic map depicting waterbird movement patterns in the Netherlands. The first synthesis run for the use case is performed over the waterbird movement data file as input and four simple constraints (see **E0.1** - **E0.4** in Table 3), describing the intent of plotting water mass, land, political borders and displaying the PostScript file (commonly used in the geospatial domain to generate raw maps). The synthesis tool finds first 32 candidate solutions of length 6. Evaluation of the first 3 candidate solutions resulted in choosing the output presented under label **E0** in Figure 6. Although the resulting map is not completely annotated, it provides a good base for further refinement of the specification towards the actually intended solution.

In the presented example, labels **E1.1** - **E4.4** in Table 3 describe four refinement steps and labels **E1** - **E4** in Figure 6 show the outcomes of the corresponding synthesized workflows. These refinement steps are part of the interactive loop between the user and the algorithm, which is, as mentioned before, quite common practice in program synthesis.

The quality of the results obtained by the workflow discovery framework essentially depends on the quality of the domain model. Only an accurate and detailed formalization of the relevant technical domain knowledge enables the process developer to specify the intended workflow in a simple and precise manner, and the discovery framework to provide accurate and satisfying solutions. Even if we look at the initial synthesis, the domain model provides a vocabulary capable of capturing abstract concepts such as "Use tool *Draw land*" and implementing them using appropriate GMT operations. As we have shown in the previous section, workflows can be specified using only abstract terms from the domain model, with no knowledge about the GMT being required for that step. In principle, the concrete tools could be replaced with corresponding ones from a different tool set, and without changing the specification the synthesis would return different, but semantically equivalent workflows.

Regarding the required time for executing the synthesis and the impact of state explosion effects, the performance of our new implementation on the geovisualization application example was very promising: It needed about 0.08 sec to detect an unsatisfiable specification, and about 1

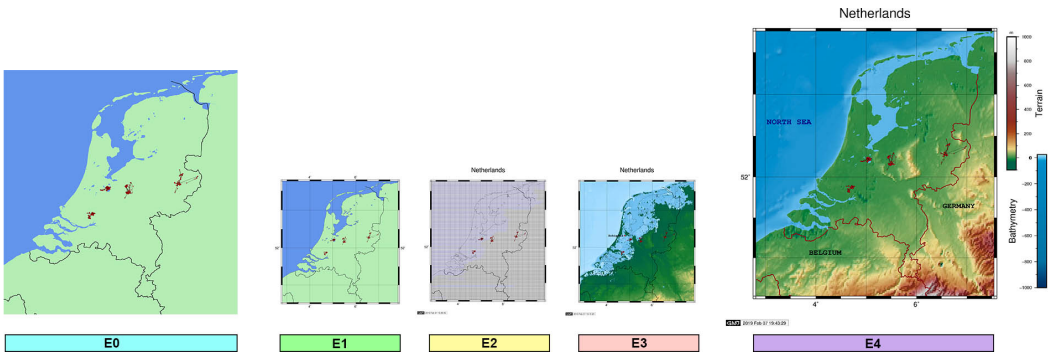

Figure 6: Output of each synthesis iteration (E0 - E4)

ID	Constraints in natural-language	Constraints in SLTL
E0.1	Use tool **Draw_water**	F ⟨**Draw_water**⟩ true
E0.2	Use tool **Draw_land**	F ⟨**Draw_land**⟩ true
E0.3	Use tool **Draw_political_borders**	F ⟨**Draw_political_borders**⟩ true
E0.4	Use **Display_PostScript** as last module in the solution	F(⟨**Display_PostScript**⟩ true G(¬⟨**Display_PostScript**⟩ true | ¬X X true))
E1.1	Use tool **Draw_boundary_frame**	F ⟨**Draw_boundary_frame**⟩ true
E1.2	Use tool **Write_title**	F ⟨**Write_title**⟩ true
E1.3	Use tool **Draw_time_stamp_logo**	F ⟨**Draw_time_stamp_logo**⟩ true
E1.4	Use tool **Draw_lines**	F ⟨**Draw_lines**⟩ true
E1.5	Use tool **Draw_points**	F ⟨**Draw_points**⟩ true
E2	Use tool **Add_table**	F ⟨**Add_table**⟩ true
E3	Use tool **2D_surfaces**	F ⟨**2D_surfaces**⟩ true
E4.1	Use tool **2D_surfaces**	F ⟨**2D_surfaces**⟩ true
4.2	Use tool **Gradient_generation**	F ⟨**Gradient_generation**⟩ true
E4.3	Use tool **Modules_with_cp_output**	F ⟨**Modules_with_cp_output**⟩ true
E4.4	Use tool **Draw_color_range**	F ⟨**Draw_color_range**⟩ true

Table 3: Synthesis constraints used in each iteration (E0 - E4)

sec to find the first 200 solutions in case of a satisfiable specification, which is hardly noticeable for a user of the system. A more systematic evaluation of APE's runtime performance is planned for the near future, when further workflow discovery scenarios will be operational. Currently we are working together with partners from the bioinformatics and geospatial domains on two new real-life applications with greater complexity.

6 Conclusion

Program synthesis techniques have been shown to be applicable to realize the idea of workflow discovery through semantics-based automated component identification and composition according to high-level specifications of intended computational pipelines. In this paper we describe APE (the Automatic Pipeline Explorer), a new implementation of a constraint-driven workflow discovery approach that allows for a formulation of user intentions on an abstract, conceptual level. This is not only useful to prune the search space in order to filter out irrelevant results. A decrease of the size of the search space usually also means a speedup of the synthesis process. It lies in the nature of synthesis problems that they suffer from state explosion effects, that is, the combinatorial blowup of the search space causing exponential runtime complexity of the algorithms. APE limits the synthesis problem to the creation of finite sequential workflows, and relies on the efficiency of SAT solving techniques for the implementation of the synthesis algorithm.

The comparison and ranking of the synthesis solutions is an open issue. The current implementation uses a simple heuristic of ranking the candidate workflows by their length. Although this is a workable approach in most cases, quite often it is not sufficient on its own to filter out

less desired options. The synthesis research community regards in particular domain-specific search heuristics (exploiting e.g. non-functional properties or additional knowledge about, for example, the preferred ordering of tools) as crucial towards efficient workflow synthesis in practice [STW+12, BJ13b]. We are therefore going to develop and include search heuristics for APE's specific use cases.

Another quite important steps towards a fully automated workflow composition and execution is data tracing. The solution workflow should provide traces of each data instance from its source to its utilization, as well as distinguish multiple instances of the same data type. Temporal logic is not well suited for handling cases of multiple instances of the same data. Thus, one of our next steps is to extend the formalism underlying APE to be able to handle such cases better.

Future research also needs to address the challenges of large-scale semantic domain modeling, and, to be adopted by the scientific community, the workflow synthesis frameworks need to be integrated with the scientists' accustomed software ecosystems. Ultimately, they should not only support the construction of computational pipelines, but also their systematic benchmarking with real input data. These are some of the problems that we are going to address with the future development of APE, our synthesis-based workflow discovery framework.

Bibliography

[ABJ+13] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. Pp. 1–8. Oct. 2013.

[ACT+16] P. Amstutz, M. R. Crusoe, N. Tijani et al. Common Workflow Language, v1.0. July 2016.

[AGMT17] M. Atkinson, S. Gesing, J. Montagnat, I. Taylor. Scientific workflows: Past, present and future. *Future Generation Computer Systems* 75:216 – 227, 2017.

[AH04] G. Antoniou, F. van Harmelen. Web Ontology Language: OWL. In Staab and Studer (eds.), *Handbook on Ontologies*. International Handbooks on Information Systems, pp. 67–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[ALM16] S. Al-Areqi, A. Lamprecht, T. Margaria. Constraints-Driven Automatic Geospatial Service Composition: Workflows for the Analysis of Sea-Level Rise Impacts. In Gervasi et al. (eds.), *Computational Science and Its Applications - ICCSA 2016 - 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part III*. Lecture Notes in Computer Science 9788, pp. 134–150. Springer, 2016.

[BCC+03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. Bounded model checking. *Advances in Computers* 58:117–148, 2003.

[BJ13a] R. Bodik, B. Jobstmann. Algorithmic program synthesis: introduction. *International Journal on Software Tools for Technology Transfer* 15(5):397–411, Oct. 2013.

[BJ13b] R. Bodik, B. Jobstmann. Algorithmic Program Synthesis: Introduction. *International Journal on Software Tools for Technology Transfer* 15(5):397–411, 2013.

[CSG+03] L. Chen, N. Shadbolt, C. Goble et al. Towards a Knowledge-Based Approach to Semantic Service Composition. In *The SemanticWeb - ISWC 2003*. P. 319334. 2003.

[DCFo17] P. Di Tommaso, M. Chatzou, E. W. Floden, others. Nextflow enables reproducible computational workflows. *Nature Biotechnology* 35:316–319, Apr. 2017.

[Dij79] E. W. Dijkstra. Program Inversion. In *Program Construction, International Summer School*. Pp. 54–57. Springer-Verlag, London, UK, UK, 1979.

[DSS+05] E. Deelman, G. Singh, M. hui Su et al. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal* 13:219–237, 2005.

[EEN05] N. EEN. MiniSat : A SAT solver with conflict-clause minimization. *Proc. SAT-05: 8th Int. Conf. on Theory and Applications of Satisfiability Testing*, pp. 502–518, 2005.

[FSMZ95] B. Freitag, B. Steffen, T. Margaria, U. Zukowski. An Approach to Intelligent Software Library Management. In *Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA)*. P. 7178. World Scientific Press, 1995.

[GJTV11] S. Gulwani, S. Jha, A. Tiwari, R. Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11, pp. 62–73. ACM, New York, NY, USA, 2011. event-place: San Jose, California, USA.

[GPS17] S. Gulwani, O. Polozov, R. Singh. Program Synthesis. *Foundations and Trends in Programming Languages* 4(1-2):1–119, July 2017.

[Gre69] C. Green. Application of Theorem Proving to Problem Solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI'69, pp. 219–239. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1969. event-place: Washington, DC.

[GRKo11] Y. Gil, V. Ratnakar, J. Kim, others. Wings: Intelligent Workflow-Based Design of Computational Experiments. *IEEE Intelligent Systems* 26(1):62–72, Jan. 2011.

[Gul10] S. Gulwani. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. PPDP '10, pp. 13–24. ACM, New York, NY, USA, 2010. event-place: Hagenberg, Austria.

[Gul11] S. Gulwani. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages. POPL '11, pp. 317–330. ACM, New York, NY, USA, 2011. event-place: Austin, Texas, USA.

[Gul16] S. Gulwani. Programming by examples (and its applications in data wrangling). Pp. 137–158. Apr. 2016.

[I+04] E. J. Ilkay Altintas, Chad Berkley et al. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *Proceedings of SSDBM 2004*. P. 2123. IEEE Computer Society, June 2004.

[IKJ+13] J. Ison, M. Kala, I. Jonassen et al. EDAM: an ontology of bioinformatics operations, types of data and identifiers, topics and formats. *Bioinformatics*, 2013.

[IRMo16] J. Ison, K. Rapacki, H. Mnager, others. Tools and data services registry: a community effort to document bioinformatics resources. *Nucleic Acids Research* 44(D1):D38–47, Jan. 2016.

[KBBG08] S. Kona, A. Bansal, M. Blake, G. Gupta. Generalized Semantics-Based Service Composition. In *ICWS 2008*. Pp. 219–227. IEEE Computer Society, Sept. 2008.

[KCW+11] B. Kranstauber, A. Cameron, R. Weinzerl, T. Fountain, S. Tilak, M. Wikelski, R. Kays. The Movebank data model for animal tracking. *Environmental Modelling & Software* 26(6):834–835, June 2011.

[KL18] V. Kasalica, A.-L. Lamprecht. Automated Composition of Scientific Workflows: A Case Study on Geographic Data Manipulation. Pp. 362–363. 10 2018.

[KL19] V. Kasalica, A.-L. Lamprecht. Workflow Discovery Through Semantic Constraints: A Geovisualization Case Study. In Misra et al. (eds.), *Computational Science and Its Applications ICCSA 2019*. Lecture Notes in Computer Science, pp. 473–488. Springer International Publishing, Cham, 2019.

[KMP+10] V. Kuncak, M. Mayer, R. Piskac, P. Suter, V. Kuncak, M. Mayer, R. Piskac, P. Suter. Complete functional synthesis. *ACM SIGPLAN Notices* 45(6):316–329, June 2010.

[KS92] H. Kautz, B. Selman. Planning as satisfiability. In *Proceedings of the 10th European conference on Artificial intelligence*. ECAI '92, pp. 359–363. John Wiley & Sons, Inc., Vienna, Austria, Aug. 1992.

[KS96] H. Kautz, B. Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 2*. AAAI'96, pp. 1194–1201. AAAI Press, Portland, Oregon, Aug. 1996.

[Lam13] A.-L. Lamprecht. *User-Level Workflow Design - A Bioinformatics Perspective*. Lecture Notes in Computer Science 8311. Springer, 2013.

[LDW00] T. A. Lau, P. Domingos, D. S. Weld. Version Space Algebra and Its Application to Programming by Demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*. ICML '00, pp. 527–534. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[LMS09] A.-L. Lamprecht, T. Margaria, B. Steffen. Bio-jETI: a framework for semantics-based service composition. *BMC Bioinformatics* 10 Suppl 10:S8, 2009.

[LNMS10] A.-L. Lamprecht, S. Naujokat, T. Margaria, B. Steffen. Synthesis-Based Loose Programming. In *Proc. of the 7th Int. Conf. on the Quality of Information and Communications Technology (QUATIC 2010), Porto, Portugal*. Pp. 262–267. IEEE, Sept. 2010.

[LPV+15] J. Liu, E. Pacitti, P. Valduriez et al. A Survey of Data-Intensive Scientific Workflow Management. *Journal of Grid Computing* 13(4):457–493, Dec 2015.

[LV13] Y. Lustig, M. Y. Vardi. Synthesis from Component Libraries. *International Journal on Software Tools for Technology Transfer* 15(5):603–618, 2013.

[MPM+05] D. Martin, M. Paolucci, S. McIlraith et al. *Bringing Semantics to Web Services: The OWL-S Approach*. Lecture Notes in Computer Science 3387, pp. 26–42. Springer Berlin / Heidelberg, 2005.

[MS07] T. Margaria, B. Steffen. LTL-Guided Planning: Revisiting Automatic Tool Composition in ETI. In *Proc. of the 31st Annual IEEE / NASA SEW 2007, Columbia, MD, USA*. Pp. 214–226. IEEE Computer Society, 2007.

[MS12] T. Margaria, B. Steffen. Service-Orientation: Conquering Complexity with XMDD. In Hinchey and Coyle (eds.), *Conquering Complexity*. Pp. 217–236. Springer London, 2012.

[MW71] Z. Manna, R. J. Waldinger. Toward Automatic Program Synthesis. *Commun. ACM* 14(3):151–165, Mar. 1971.

[MW75] Z. Manna, R. Waldinger. Knowledge and reasoning in program synthesis. *Artificial Intelligence* 6(2):175–208, June 1975.

[MW92] D. H. MO, I. H. WITTEN. Learning text editing tasks from examples: a procedural approach. *Behaviour & Information Technology* 11(1):32–45, Jan. 1992.

[MXBK05] D. Mandelin, L. Xu, R. Bodk, D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05, pp. 48–61. ACM, New York, NY, USA, 2005. event-place: Chicago, IL, USA.

[NLKS17] S. Naujokat, M. Lybecait, D. Kopetzki, B. Steffen. CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools. *Software Tools for Technology Transfer*, 2017.

[NLS12] S. Naujokat, A.-L. Lamprecht, B. Steffen. Loose Programming with PROPHETS. In Lara and Zisman (eds.), *Proc. of FASE 2012, Tallinn, Estonia*. LNCS 7212, pp. 94–98. Springer Heidelberg, 2012.

[NNMS16] S. Naujokat, J. Neubauer, T. Margaria, B. Steffen. Meta-Level Reuse for Mastering Domain Specialization. In *Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part II (ISoLA 2016)*. LNCS 9953, pp. 218–237. Springer, 2016.

[PLIS18] M. Palmblad, A.-L. Lamprecht, J. Ison, V. Schwmmle. Automated workflow composition in mass spectrometry-based proteomics. 2018.

[PR89] A. Pnueli, R. Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89, pp. 179–190. ACM, New York, NY, USA, 1989.

[PTBo16] P. M. Phothilimthana, A. Thakur, R. Bodik, others. Scaling up Superoptimization. *ACM SIGOPS Operating Systems Review* 50(2):297–310, June 2016.

[SGF10] S. Srivastava, S. Gulwani, J. S. Foster. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*. Pp. 313–326. ACM, Madrid, Spain, 2010.

[SMB97] B. Steffen, T. Margaria, V. Braun. The Electronic Tool Integration platform: concepts and design. *International Journal on Software Tools for Technology Transfer (STTT)* 1(1-2):9–30, 1997.

[SMF93] B. Steffen, T. Margaria, B. Freitag. Module Configuration by Minimal Model Construction. Technical report, Fakultt fr Mathematik und Informatik, Universitt Passau, 1993.

[Smi75] D. C. Smith. *Pygmalion: A Creative Programming Environment*. Computer Science Department, Stanford University, 1975. Google-Books-ID: mihHAAAAIAAJ.

[SMN+07] B. Steffen, T. Margaria, R. Nagel, S. Jrges, C. Kubczak. Model-Driven Development with the jABC. In Bin et al. (eds.), *Hardware and Software, Verification and Testing*. Lecture Notes in CS 4383, pp. 92–108. Springer Berlin / Heidelberg, 2007.

[SL08] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD Thesis, University of California at Berkeley, Berkeley, CA, USA, 2008.

[SSG75] D. E. Shaw, W. R. Swartout, C. C. Green. Inferring LISP Programs from Examples. Technical report CUCS-001-75, Department of Computer Science, Columbia University, 1975.

[STW+12] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, P. J. Stuckey. An Introduction to Search Combinators. In *International Symposium on Logic-Based Program Synthesis and Transformation*. Pp. 2–16. 2012.

[Sum86] P. D. Summers. A Methodology for LISP Program Construction from Examples. In Rich and Waters (eds.), *Readings in Artificial Intelligence and Software Engineering*. Pp. 309–316. Morgan Kaufmann, Jan. 1986.

[SWS] Scientific workflow system. [Online; 14 Feb 2019]. https://en.wikipedia.org/w/index.php?title=Scientific_workflow_system

[Val98] A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*. Pp. 429–528. Springer-Verlag, London, UK, 1998.

[WHF+13] K. Wolstencroft, R. Haines, D. Fellows et al. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research* 41(W1):W557–W561, 2013.

[WS91] P. Wessel, W. H. F. Smith. Free software helps map and display data. *EOS Trans. Amer. Geophys. U.* 72(41), 1991.