CNT: Semi-Automatic Translation from CWL to Nextflow for Genomic Workflows

Martin L. Putra*, In Kee Kim†, Haryadi S. Gunawi*, and Robert L. Grossman*
*Department of Computer Science, University of Chicago, {martinluttap, haryadi, rgrossman1}@uchicago.edu
†School of Computing, University of Georgia, inkee.kim@uga.edu

***Abstract*—With the rise of advanced workflow languages for scientific computations, Nextflow has gained increased attention from the bioinformatics community. Nextflow offers native support for advanced parallelism, which can greatly enhance resource utilization and throughput. Still, a significant portion of bioinformatics workflows are developed with the Common Workflow Language (CWL). Transitioning from CWL to Nextflow poses a significant challenge due to the differences in programming models, scripting language compatibilities, and the prerequisite for in-depth knowledge in both languages.**

To address this challenge, we present CNT, a novel, semi-automated translator converting CWL workflows into Nextflow ones. At its core, CNT uses an automated translation mechanism that converts the CommandLineTool, the most basic unit of CWL, into Nextflow's Process class. This component integrates tool-level conversion, graph dependency analysis, and correctness checks to provide highly automated translation coverage, significantly reducing the development time while satisfying language-specific requirements like building a proper dataflow model when creating workflows. Furthermore, CNT incorporates a module for aiding manual translation. Specifically, it can identify three common JavaScript patterns in CWL workflows, offering further guidance for developers during the translation phase. We evaluated CNT with production-grade workflows and found that it can cover up to 81% of the original workflows, substantially reducing development time. Additionally, transitioning from a cwltool-based system to Nextflow with CNT can result in a 72% speedup and 85% increased CPU utilization.

***Index Terms*—Genomic Workflows; CWL; Nextflow, Workflow Translation; Bioinformatics;**

I. Introduction

Workflow languages are widely used when production systems analyze biomedical data since they provide consistent processing of batches of data, reproducible processing of a particular dataset, restart and reentry, data provenance, and other desirable properties, such as the ability to easily share bioinformatics pipelines with others. The popularity of particular workflow languages and workflow managers varies by subfield, data type, computing environment, and over time. For this reason, a wide variety of workflow languages and workflow managers are used in bioinformatics. For example, the survey [1] mentions over 150 workflow managers in use or in production.

Given the variety of workflow languages, it is desirable to have reliable tools that translate a workflow from one workflow language to another. For example, the Genomic Data Commons (GDC) [2] has developed 11 production CWL workflows [3] that are used for processing all the data submitted to the GDC. Given the popularity of the GDC and the amount of data it processes, others are interested in using the same workflows so that the data may be processed uniformly [3]. On the other hand, different groups may want to use different workflow managers.

	DNA-Seq	RNA-Seq
Speedup over cwltool [5]	21-39%	33-72%
CPU Utilization Increase	18-33%	45-85%
Translation Coverage	73%	81%

TABLE I. **Summary of evaluation results (§I).**

With the growing popularity of Nextflow [4], there is significant interest within the bioinformatics and cancer genomics community in having a tool that translates CWL into Nextflow, as they have developed numerous CWL pipelines. While converting a CWL workflow to Nextflow manually is possible, this approach is exhaustively time-consuming and demands in-depth domain-specific expertise. For example, based on our experience, translating a production-grade pipeline, such as the GDC DNA-Seq workflow from CWL to Nextflow, takes approximately 160 person-hours. This estimate does not include testing and integration time.

Translating a CWL workflow into Nextflow presents several significant challenges. First, Nextflow follows a dataflow programming model, whereas CWL is declarative. This requires translation efforts to ensure the correctness of the dataflow model in Nextflow using various Nextflow operators. In contrast, CWL often relies on the workflow engine to support dataflow correctness, resulting in the explicit dataflow being commonly overlooked in workflow descriptions. Second, existing CWL workflows heavily utilize JavaScript, which must be translated into Nextflow's scripting counterpart (*e.g.*, Groovy). This implies that the workflow translation involves not only moving from CWL to Nextflow but also transitioning from JavaScript to another language. Finally, to achieve the desired performance, developers must be familiar with advanced task parallelism supported in Nextflow and apply the appropriate Nextflow operators.

To address these challenges and maximize the benefits of adopting Nextflow, we developed CNT, which is, to the best of our knowledge, *the first translator* semi-automatically converting CWL workflows into Nextflow. CNT operates in three stages: a tool-level translation, graph dependency analysis, and a correctness check. In each stage, we tackle specific challenges. The tool-level translation converts CWL's

2471-7819/23/$31.00 ©2023 IEEE
DOI 10.1109/BIBE60311.2023.00012

smallest workflow components, CommandLineTool files, into Nextflow's Processes through a five-step approach based on essential CommandLineTools fields. Subsequently, the graph dependency analysis organizes these components into a complete workflow, addressing critical ordering challenges related to input/output variables in a Process, the sequence of Processes in a subworkflow, and the arrangement of subworkflows in the final workflow. Finally, CNT ensures correctness by performing typecasting that verifies the use of appropriate data types for Nextflow processes and operators. Moreover, while CNT can automatically translate a significant portion (*e.g.*, 73% – 81%) of CWL code to Nextflow, certain code segments *may* require manual intervention, such as developer-written JavaScript codes. To enhance the translation process, CNT identifies three JavaScript code patterns that guide developers during the translation.

To evaluate CNT's performance, we used CNT to translate the GDC DNA-Seq and RNA-Seq pipelines into Nextflow. Our evaluation first focuses on measuring translation coverage and correctness. The results show that CNT can automatically translate up to 81% of CWL code to Nextflow with a correctness rate of up to 71%. Subsequently, we evaluate the performance improvements achieved by using Nextflow workflows translated by CNT vs. running CWL on cwltool [5], the reference implementation of CWL. The results reveal significant speedups, with execution time reductions of 72% in RNA-Seq and 39% in DNA-Seq, along with an average increase in CPU utilization by 65% and 25.5%, respectively.

II. Background

This section provides background information on the development of a workflow translator from CWL to Nextflow.

Workflow Languages. Protocols for analyzing bioinformatics data are often represented as computational workflows. Many workflow languages have been developed to represent such workflows. These languages are designed to achieve different goals, such as portability and reproducibility (*e.g.*, CWL [6]), human-readability/writability (*e.g.*, WDL [7]), up to fine-grained control of dataflow (*e.g.*. Nextflow [8]).

Common Workflow Language (CWL). CWL uses two core classes of files to represent workflows: *CommandLineTool* and *Workflow*. The 'CommandLineTool' is the smallest building block of a workflow, describing at least a specific command-line application, its parameters, input data requirements, and the output it produces (the full list of supported information can be found in [9]). The 'Workflow' class takes a CommandLineTool file as a single computational step and composes it into a whole computational workflow. Workflows can be nested, allowing a single step within a Workflow to be another Workflow file up to arbitrary depths. Furthermore, CWL allows the use of JavaScript code to manipulate output files, such as changing names and constructing/deconstructing more complex data structures.

Nextflow. Analogous to CWL, Nextflow also has two classes which represent the smallest workflow building block and a whole computational workflow: *Process* and *Workflow*.

CommandLineTool Field	Template Placeholder
ID [E]	Process Name
Requirements [E]	Container
Inputs [C]	Input, Optional Variable
Outputs [C]	Output
baseCommand [C]	Script
Arguments [C]	Script

TABLE II. **Summary of value substitution (§III-A).** *E: elementary item that does not require processing, C: a collection that requires one before substitution.*

However, unlike CWL, Nextflow employs Groovy as its scripting language. It also introduces many dataflow operators [8] which need to be used appropriately to leverage parallelism during workflow execution.

III. Design of CNT

We present CNT, a system designed for semi-automatic translation of CWL workflows into Nextflow workflows. CNT consists of three stages:

- **Tool-level translation (§III-A).** This stage constructs the smallest workflow building block by translating a CWL's CommandLineTool into Nextflow's Process, specifically by capturing and translating essential fields defined in CWL and then arranging these building blocks into a subworkflow in the next stage.
- **Graph-dependency analysis (§III-B).** Following that, CNT addresses three important ordering challenges: the order of input/output variables in a Nextflow process, the order of processes in a subworkflow, and the order of subworkflows in the final translated workflow. CNT solves these challenges simultaneously by doing a recursive exploration technique that captures data lineage information and then builds a directed acyclic graph (DAG), which describes the type signature and order of invocations for both processes and subworkflows.
- **Correctness check (§III-C).** The last stage ensures accurate workflow translation by conducting a correctness check for both tool-level translation and graph-dependency analysis. CNT utilizes *typecasting*, a method to confirm the use of proper data types for Nextflow processes and operators.

CNT performs all these stages using only a JSON representation of the original CWL files. The following subsections explain the above three stages in detail.

A. Tool-Level Translation

This stage consists of 5 steps: template construction, field extraction, value substitution, optional variable construction, and entrypoint management. Fig. 1 illustrates the steps involved in tool-level translation, each of which is denoted by a number from ① – ⑤.

Template generation. We observed that Nextflow process structure can accommodate most information from CWL's CommandLineTool. Moreover, each piece of information (*e.g.*, one input variable) can be separated by a new line – a property that we will exploit in later steps. These features

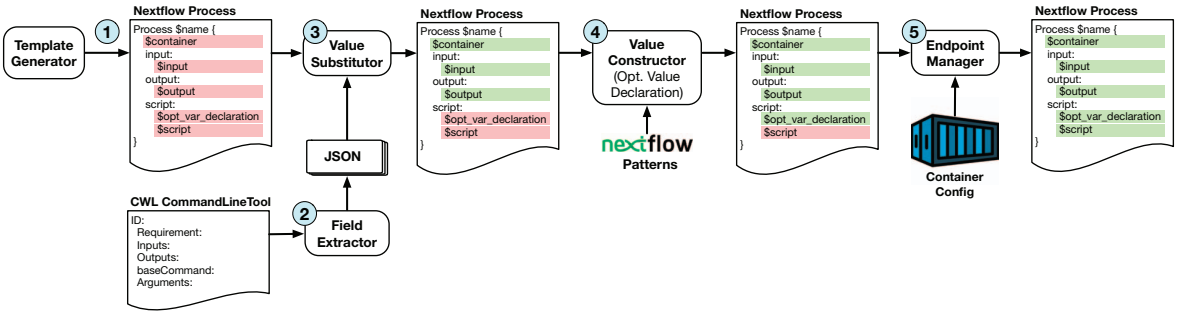

Fig. 1: **Five steps of tool-level translation (§III-A).**

facilitate the generation of a Nextflow Process template (Step ①). The template provides a placeholder for process name, container information, input/output variables, shell commands, and optional variable declarations. Throughout subsequent steps, the template will be filled with corresponding information from CommandLineTool fields, one line for each piece of information.

Fields extraction. This step extracts the information needed to fill the placeholders in the template. We identified 6 CommandLineTool fields that are essential for creating a fully functional Nextflow Process: ID, requirements, inputs, outputs, baseCommand, and arguments (Step ②). Information extraction is done by transforming the CommandLineTool source file into a JSON representation, followed by accessing the relevant key-values. `CNT` uses cwl-utils [10] to transform CWL source files into JSON. Following that, it stores all extracted information in-memory to be processed before writing them to the template placeholders.

Value substitution. Table II summarizes the CommandLineTool fields used for filling each template placeholder along with its type. The value of each field can be in the form of elementary items (*e.g.*, string or integer) or collections (*e.g.*, list or dictionary). For elementary item, `CNT` directly inserts the value to its respective placeholder (Step ③). For a collection, `CNT` first flattens any nested items. It then performs additional extractions and constructs a string using the elements from the collection.

Optional variable declaration. This step is to support the dataflow model in Nextflow, wherein a Nextflow Process requires explicit configurations of input and can only execute once all its inputs are ready. In contrast, CWL provides for optional inputs, simplifying the writing and execution of workflows (*e.g.*, allowing workflows to run without specifying every input). There was an effort from the Nextflow community to develop a pattern for addressing CWL's optional inputs [11]. `CNT` integrates this pattern to ensure that optional input channels always contain an item, whether or not they are supplied (Step ④).

Container entrypoint management. Containerized deployments for scientific workflows have been widely adopted in recent years [12], where each step can be encapsulated within a container (*e.g.*, Docker). This approach requires managing containers' entry points in order to determine which binary is executed and its corresponding path upon initialization. `CNT` addresses this issue by examining the configuration file (*e.g.*, Dockerfile) for each container involved in the workflow (Step ⑤), thus ensuring the appropriate binary path is used.

B. Graph-Dependency Analysis

The next step in `CNT` involves composing Nextflow Processes, which were created during tool-level translation, into subworkflows, and combining these subworkflows into a final workflow while addressing ordering challenges. The ordering challenges stem from the difference in how CWL and Nextflow perform invocations: named vs. positional arguments. These challenges can arise in both the Nextflow Process and the subworkflow layer. In the Process layer, the challenge is to manage the order of input/output variables. In the subworkflow layer, the challenge is to manage the order of invocations.

However, to address these challenges, `CNT` requires information that is scattered across all CWL files, and collecting all the information is not trivial. For example, `CNT` first needs to collect the path of all related files. Subsequently, it has to extract all necessary information from each file. `CNT` attempts this through a two-steps approach: recursive exploration for file path collection and DAG construction for the target workflow.

Recursive exploration file path collection. `CNT` first collects the paths of all CWL files used in the workflow. In CWL, a file with class Workflow can include other CWL files as steps, while a file with CommandLineTool can not. The challenge is twofold: 1) CWL allows to use relative path when including another file as a step, and 2) an included file may, in turn, include other files, up to arbitrary depth. To address these challenges, `CNT` performs a DFS-like recursive exploration technique with the following logic. Each file is considered as the root of a tree with unknown depth. Other files included as steps are the root's children. `CNT` then examines the JSON representation of each child one by one. If the file is a CommandLineTool, `CNT` extracts its path and then backtracks to its previous operations. However, if the file is a Workflow, `CNT` treats that file as another root and performs the exact same search. This procedure is recursively executed until `CNT` traverses the whole tree.

DAG construction for target workflow. With the collected file paths, `CNT` extracts all the required information from each file and constructs a DAG for the target workflow. During DAG

construction, CNT solves the ordering challenges by ensuring correct order of: 1) I/O variables, and 2) invocations.

1) *Order of I/O variables*. When translating a language that uses named arguments (*e.g.*, CWL) to one that uses positional arguments (*e.g.*, Nextflow), reconstructing type signature requires information from two layers: the callee layer and the caller layer. This is because, with named-arguments, the input/output variables from one layer might appear in different order and aliases in another layer. A translation system can choose either order, but it must ensure consistency across both layers. CNT achieves this by maintaing a DAG data structure. For each callee, CNT creates a *vertex* that contains a map as attribute. The map describes the alias of each of callee's variable as viewed by its caller.

2) *Order of invocations.* After creating vertices, CNT analyzes the relationship between them. A directed edge from a vertex to another is created if an output from the former is used as input by the latter. This completes the DAG construction. Order of invocations is equivalent to topological order of the DAG's vertices.

C. Correctness Check

We observed that when performing translations, CNT needs to take into account Nextflow's runtime behavior related to the types of input and output variables, which affects Nextflow's runtime file staging. However, ensuring correct type translation is challenging based on our experience, as there are multiple points in a workflow where a variable may be transformed from a file to another type.

We categorize these possible locations into two layers: Nextflow's Process layer and workflow layer, each with its own characteristics. In Process layer, undesirable transformations can occur when a variable is passed as either input or output. In the workflow layer, these transformations can take place wherever a Nextflow operator handles the variables.

Typecasting. We developed a method called typecasting that ensures correct type translation across both layers.

At the Nextflow Process layer, CNT examines the type declared for each variable within input and output block. It ensures variables intended to represent file objects are correctly declared by inspecting the attributes of DAG vertices associated with the Nextflow Process (§III-B). Each Nextflow Process can be associated with at least two vertices: one representing a workflow which invokes it, and the other representing a Nextflow Process itself. As each vertex holds aliases for its respective input and output variables, CNT can determine whether a Process input was originally passed as a file object by examining these aliases. This information is used for declaring variable types.

At the workflow layer, CNT ensures the correct use of Nextflow operators to avoid undesired translations. We also ensure that a variable originating from a file is processed with Nextflow operators that preserve its type, such as using Channel.path() instead of Channel.of(). To achieve this, CNT considers data lineage by utilizing the DAG constructed in §III-B. The origin of a variable can be traced by identifying the edge representing the variable and then traversing back through nodes and directed edges in the DAG.

IV. Manual Translation

While CNT automatically translates a significant portion of CWL codes (detailed in §V), some code segments of CWL workflows require manual translation. Manual translation has two main objectives: 1) *ensuring syntax conformance* and 2) *achieving the expected performance*. Syntax conformance aims to reduce errors when converting JavaScript in CWL to Groovy in Nextflow. Achieving the expected performance ensures that the translated workflows deliver the desired performance with proper dataflow and task parallelism.

A. Ensuring Syntax Conformance

Formal methods [13] for translating JavaScript code into Groovy require constructing a grammar for each language. Since CWL's adaptation of JavaScript uses a slightly altered grammar (*e.g.*, enclosing inline expressions with '$()' or prefixing with '|'), for this work we translated JavaScript into Groovy based on a code-level analysis of our CWL workflows. We found that JavaScript expressions within these workflows can be categorized into three common patterns: 1) object attribute access, used to extract values from collection inputs, 2) object method calls, handling simple modifications on literal values, such as string pre/suffix changes and upper/lower case transformations, and 3) 'actual' code block, containing tasks ranging from constructing collections and sorting to calculating and managing workflow executions.

Based on these findings, we further enhanced CNT with a three-step approach to assist developers in semi-automatically addressing each pattern. This approach utilizes our carefully designed regex rules to accurately identify each pattern. Subsequently, CNT performs the following three steps.

Handling object attribute access. This pattern typically appears as two or three JavaScript tokens separated by dots. As Groovy's way of accessing object attributes is identical to that of JavaScript's, CNT can directly apply this expression in the translation. A challenge for CNT, however, is addressing the many strings appearing before, between, or after successive patterns. Our regex rules effectively tackle this issue.

Replacing object method call. The challenge in this step arises from the similarity of this pattern to the previous one. The key difference is that the last token in this pattern represents a method call. To differentiate attributes from method call tokens, we identified frequently used method names, treating them as 'flags' for this pattern. For example, in our target workflows, we identified four common method calls: *nameroot*, *tostring*, *basename*, and *dirname*. CNT then maps each method call to its corresponding Groovy equivalent.

Isolating 'actual' code block. We recognize that the methods applied to the previous patterns are not sufficient for addressing 'actual' code blocks due to the extensive variability in JavaScript codes. However, CNT can correctly detect and isolate these code blocks. In its current version, CNT replaces them with a placeholder designed to trigger a syntax error.

	# Files	MD5 Simil.	O	OV	S
RNA	31	71%	0%	22%	100%
DNA	46	63%	55%	22%	44%

TABLE III. **Translation MD5 similarity for automated-translation of CNT (§V-A).** *Column names "O", "OV", and "S" are shortnames for template placeholders described in §III-A. O: Output, OV: Optional Variable, S: Script.*

This placeholder is based on the principle that a clear failure is often better than introducing transient errors. Furthermore, this placeholder acts as a signal for developers, indicating the need for manual translation of the code block into Groovy.

Although resolving this challenge still requires manual intervention by developers, our experience indicates that CNT can significantly reduce the development time by roughly 75%.

V. Evaluation

We built CNT in Python, leveraging the `cwl-utils` [10] that facilitates parsing CWL into a JSON representation. We evaluated CNT's performance across two dimensions: translation MD5 similarity and coverage by employing two widely-used genomic workflows, namely RNA-Seq and DNA-Seq. We then measured the performance gain of adopting CNT by comparing the execution speedup and CPU utilization increase against a popular CWL execution engine.

A. Translation MD5 Similarity

CNT successfully performs tool-level translation using the 'MD5 similarity' metric, where the **upper bound for perfect similarity is 74%** for RNA-Seq (because JavaScript code with an unhandled pattern will cause an execution error). **CNT is able to achieve 71% similarity** for RNA-Seq.

We define 'MD5 Similarity' as having an identical MD5 value between the outputs of CWL and its Nextflow translation. We focused on evaluating the correctness of workflow steps translated by the tool-level translation (§III-A), because workflow-level translation often involves manual intervention which leads to different MD5 values. Furthermore, some outputs may contain execution timestamps, resulting in a unique MD5 value for each execution. To address this, we removed the timestamp from these outputs before calculating their MD5 value, or exclude them if removal is not possible (*e.g.*, binary output format).

For any translation errors, we categorize the cause into three groups based on the template placeholder that causes it: output (O), optional variable (OV), and script (S). A single translation mistake can involve multiple placeholders, in which case we increment counts for each affected placeholder. We omitted input and container placeholders from the results since they did not exhibit any translation errors.

Table III reports the translation MD5 similarity for two workflows. The '# Files' column represents the number of CommandLineTools files invoked during execution, while the 'MD5 Simil.' column indicates the percentage of files producing correct outputs. Template placeholder columns (*e.g.*, O, OV, S) show the percentage of translation errors attributed to each placeholder. For example, for DNA-Seq workflows, O=55% means 55% of incorrect translations have mistakes in the output placeholder.

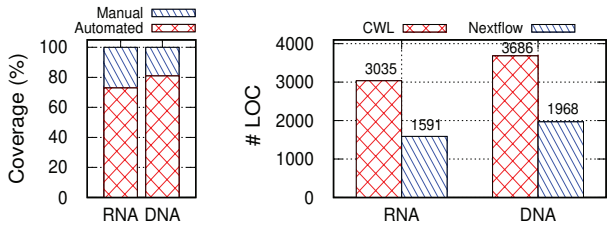

Fig. 2: **Coverage and LOC evaluation results. (§V-B)** *(a): x-axis: workflow types, y-axis: the percentage of coverage. (b): x-axis: workflow types, y-axis: the total LOC in thousands.*

The results have several interesting findings. We discuss only one due to the page limit. *We observed that no incorrect translations in RNA-Seq workflow occurred on the output (O) placeholder, while the value was around half for DNA-Seq.* This indicates DNA-Seq's way of declaring output is generally more challenging to handle by CNT compared to RNA-Seq. Our in-depth manual inspection revealed that while RNA-Seq's output declaration heavily utilized JavaScript, the majority aligned with our generalized JavaScript patterns (§IV-A). On the other hand, the JavaScript patterns in DNA-Seq are more complex, such as "nameroot.split('_').slice(0,4).join('_')".

B. Translation Coverage and Total Lines of Code (LOC)

Fig. 2a shows the translation coverage of CNT for both RNA-Seq and DNA-Seq workflows, **achieving 73% and 81% coverage**, respectively. In this figure, the lower portion represents the fraction of automated code translation, while the upper portion denotes manually written code. The automatically translated part aggregates code from tool-level translation, graph dependencies, typecasting, and partially-handled JavaScript. The manual part includes all fixes and fine-tuning needed to ensure syntax conformance and achieve the expected performance.

The results indicate that CNT had about 8% lower coverage for RNA-Seq than for DNA-Seq, leading to a larger portion of code being manually translated despite RNA-Seq's more accurate translation correctness. Our further analysis revealed that RNA-Seq has substantial volume of code from CommandLineTool files which requires manual replacements with Nextflow's operators.

Fig. 2b compares the total lines of code (LOC) of CWL workflows with their corresponding Nextflow translations, including both automated and manually translated code. We define the total LOC as the cumulative LOC across all files in a workflow. To calculate the LOC of a single file, we first took the total line count and deducted lines comprised solely of comments and whitespaces. Although our approach might count multiple complex JavaScript or Groovy statements on a single line (*e.g.*, separated by ';') as a single LOC, we found that such occurrences are exceptionally rare in practice.

C. Performance Gain by Adopting CNT

Our evaluation primarily focused on two key metrics: speedup (execution time reduction) and CPU utilization improvement when processing the same workflow using cwltool [5] and Nextflow. Cwltool, the open-source official reference for CWL, was chosen because of its wide adoption in the community. Our evaluation shows an **average speedup of 52.5% for RNA-Seq and 30% for DNA-Seq** when migrating from a cwltool-based system to a Nextflow-based system. Additionally, the Nextflow-based system showed an **average increased CPU utilization of 65% for RNA-Seq and 25.5% for DNA-Seq**, in comparison to a cwltool-based system. Detailed range of speedup and utilization increase can be found in **Table I**.

VI. Related Work

Several works employed learning techniques to achieve automatic translation between general-purpose programming and scripting languages. NGST2 [14] developed a neural architecture capable of translating sections of imperative general-purpose languages into their functional language counterparts, supporting convenient parallelization. DuoGlot [13] addresses the challenge of maintaining readable automatic translations, even when the source code has different coding styles, by incorporating user feedback. In contrast to these works, CNT does not rely on learning techniques, avoiding the need for extensive datasets which are rare yet necessary to train models effectively.

As efficiency and safety are both crucial for systems-level applications, there are several attempts to translate legacy code to memory-safe Rust. Emre *et al*. developed techniques to automatically generate safe Rust code from unsafe segments of C code [15]. Lunnikivi *et al*. semi-automatically translated Python code into low-level code for efficiency using Rust as intermediate representation [16]. These projects have a similarity to CNT as they perform automated translation followed by a manual refinement. However, CNT's primary focus lies in targeting domain-specific languages, rather than the broader general-purpose languages such as C and Python.

With the increasing use of accelerator hardware, various research domains, including bioinformatics, have increasingly adopted GPUs to handle their ever-expanding workloads [17]. To facilitate GPU support, there are attempts to translate legacy code into CUDA. GPSME [18] offers semi-automatic source-to-source translation of SME applications originally written in C and C++ to CUDA. Conversely, MocCUDA [19] automatically translates CUDA code into OpenMP to take advantage of CPU-only supercomputers.

VII. Conclusion

This paper presents CNT. To the best of our knowledge, CNT is the first semi-automatic translator that converts CWL workflows into Nextflow workflows. The core of CNT is an automatic translation module that performs tool-level translation, graph dependency analysis, and correctness checks to provide highly automated translation coverage, significantly reducing the development time. Additionally, CNT has a module to aid manual translation. In particular, CNT can accurately identify three common JavaScript patterns in CWL workflows that can further guide developers during translation. Our evaluation of CNT centered on its translation accuracy, coverage, and the performance enhancements achieved by using CNT to transition from cwltool to Nextflow. Our findings highlight that CNT not only ensures precise and extensive translations but also significantly boosts workflow execution speeds and utilization, leading to significant improvement in bioinformatics job processing throughput. In summary, our evaluation demonstrates the value of CNT when transitioning from CWL to Nextflow.

References

[1] L. Wratten, A. Wilm, and J. ke. Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers. *Nature Methods*, 2021.

[2] Allison P Heath, Vincent Ferretti, Stuti Agrawal, Maksim An, James C Angelakos, Renuka Arya, Rosita Bajari, Bilal Baqar, Justin HB Barnowski, Jeffrey Burt, and et al. The NCI Genomic Data Commons. *Nature Genetics*, 2021.

[3] Z. Zhang, K. Hernandez, J. Savage, S. Li, D. Miller, S. Agrawal, F. Ortuno, L. M. Staudt, A. Heath, and R. L. Grossman. Uniform genomic data analysis in the NCI Genomic Data Commons. *Nature Communications*, 2021.

[4] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame. Nextflow Enables Reproducible Computational Workflows. *Nature Biotechnology*, 2017.

[5] cwltool: The reference implementation of the common workflow language standards. https://github.com/common-workflow-language/cwltool.

[6] Michael R. Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojš a Tijanić, Hervé Ménager, Stian Soiland-Reyes, Bogdan Gavrilović, Carole Goble, and The CWL Community. Methods included. *Communications of the ACM*, 2022.

[7] OpenWDL - Community driven open-development workflow language. https://openwdl.org.

[8] Nextflow. https://www.nextflow.io/docs/latest/.

[9] Common Workflow Language Standards, v1.2. https://www.commonwl.org/v1.2/.

[10] cwl-utils. https://github.com/common-workflow-language/cwl-utils.

[11] Nextflow Patterns. https://nextflow-io.github.io/patterns/optional-input/.

[12] Laura Wratten, Andreas Wilm, and Jonathan Goke. Reproducible, Scalable, and Shareable Analysis Pipelines with Bioinformatics Workflow Managers. *Nature Method*, 2021.

[13] Bo Wang, Aashish Kolluri, Ivica Nikolić, Teodora Baluta, and Prateek Saxena. User-Customizable Transpilation of Scripting Languages. *Proceedings of ACM Programming Language*, 2023.

[14] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Isil Dillig. Automated transpilation of imperative to functional code using neural-guided program synthesis. *Proceedings of the ACM on Programming Languages*, 2022.

[15] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating C to Safer Rust. *Proceedings of ACM Programming Language*, 2021.

[16] Henri Lunnikivi, Kai Jylkkä, and Timo Hämäläinen. Transpiling Python to Rust for Optimized Performance. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2020.

[17] A. Taylor-Weiner, F. Aguet, N. J. Haradhvala, S. Gosai, S. Anand, J. Kim, K. Ardlie, E. M. Van Allen, and G. Getz. Scaling computational genomics to millions of individuals with GPUs. *Genome Biology*, 2019.

[18] Po Yang and *et al*. Improving Utility of GPU in Accelerating Industrial Applications With User-Centered Automatic Code Translation. *IEEE Transactions on Industrial Informatics*, 2018.

[19] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, 2023.