Automated Constraint Specification for Production Scheduling by Regulating Large Language Model with Domain-Specific Representation

Yu-Zhe Shi, Qiao Xu, Yanjia Li, Mingchen Liu, Huamin Qu,
Lecheng Ruan and Qining Wang, *Senior Member, IEEE*

***Abstract*—The integration of Advanced Planning and Scheduling (APS) system in smart manufacturing environments requires precise constraint specification to effectively utilize manufacturing resources. While Large Language Models (LLMs) show promise in automating constraint specification from heterogeneous raw manufacturing data, their direct application faces challenges due to natural language ambiguity, non-deterministic outputs, and limited domain knowledge. This paper presents a constraint-centric architecture that regulates LLMs to perform reliable automated constraint specification for production scheduling. The architecture defines a hierarchical structural space organized across three levels, implemented through domain-specific representation to ensure precision and reliability while maintaining flexibility. Furthermore, an automated adaptation algorithm is introduced to efficiently customize the architecture for specific manufacturing configurations. Experimental results demonstrate that our approach successfully balances the generative capabilities of LLMs with the reliability requirements of manufacturing systems, outperforming pure LLM-based approaches in constraint specification tasks.**

***Note to Practitioners*—This paper presents a practical solution for automating the conversion of raw manufacturing data into job scheduling specifications, addressing a common challenge in implementing APS systems. The proposed architecture can process diverse manufacturing documentation formats, from structured route sheets to natural language instructions, while ensuring reliability through domain-specific representations. Manufacturing practitioners can use this system to reduce the manual effort in formalizing production rules, particularly beneficial for facilities with frequent requirement changes or small-batch, multi-variety production. The system's ability to automatically adapt to different manufacturing scenarios makes it accessible without requiring extensive programming expertise, offering a practical balance between automation and accuracy in production planning.**

I. Introduction

Smart manufacturing has become a cornerstone of industrial development, promising enhanced productivity and adaptability through digital transformation [1]. This manufacturing paradigm represents a fundamental shift from traditional production systems, incorporating advanced technologies to create more intelligent and responsive operations [2], [3], [4]. As manufacturers face increasing pressure to handle product variety and shorter delivery times, the conventional rigid production models are giving way to Flexible Manufacturing System (FMS) [5]. These systems must efficiently process multiple jobs with different specifications, processing requirements, and priorities—all while sharing limited manufacturing resources. Accordingly, Advanced Planning and Scheduling (APS) system [6] emerges as a crucial decision-making process in smart manufacturing, as it determines how effectively manufacturing resources are utilized to meet production objectives.

The implementation of an APS system comprehends two major phases: (i) the *specification* of real-world manufacturing constraints, such as order requirements and factory resource availability, into a mathematical Job Scheduling Problem (JSP) formulation [7], [8]; and (ii) the *solution* for the formulated JSP [9], [10]. In the previous research paradigm, manufacturing constraints are typically abstracted and standardized to emphasize the scheduling fundamentals while deliberately excluding factory-specific variables such as material characteristics, machine specifications, and product requirements [11], [12]. Under this setting, diverse solvers have been successfully developed for specific problem characteristics and instances [13], [14].

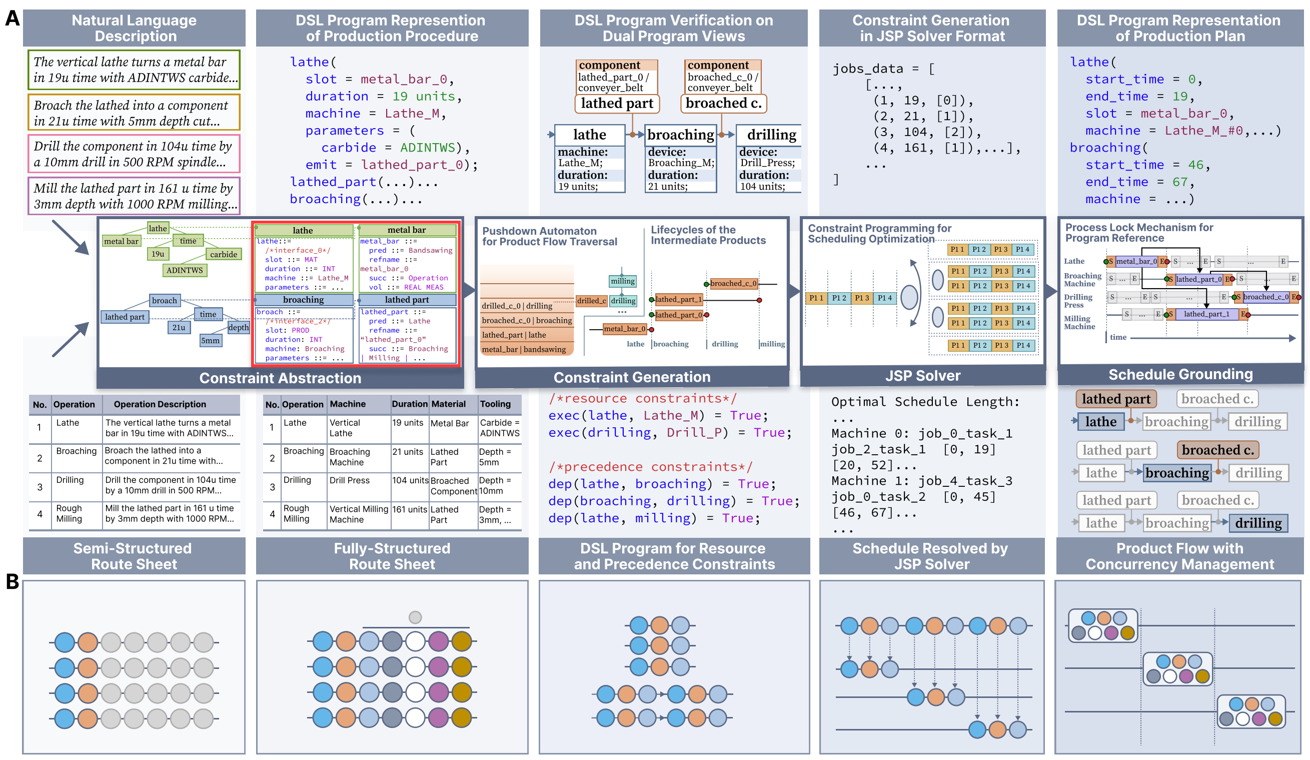
However, abstracted constraints for a JSP solver do not emerge spontaneously from real-world manufacturing scenarios; they require meticulous development and formalization by experienced manufacturing experts [15]. This manual specification process was historically manageable in traditional manufacturing settings with stable production scopes, as it was essentially a one-time effort. However, the emergence of smart manufacturing has fundamentally transformed this landscape through several challenges: (i) the shift toward small-batch, multi-variety production demands extensive constraint specifications to accommodate diverse processing requirements [16]; (ii) the dynamic nature of modern production involves continuous order arrivals and frequent disruptions requiring constant updates of new constraints [17]; and (iii) manufacturing knowledge exists in heterogeneous formats across different sources, from structured or semi-structured route sheets to Natural Language (NL) instructions [18], [19]. Furthermore, solver results must be translated back into interpretable pro-

This work was partially supported by the National Natural Science Foundation of China under Grant 52475001. Yu-Zhe Shi and Qiao Xu contributed equally. (Corresponding authors: Lecheng Ruan and Qining Wang)

Yu-Zhe Shi, Qiao Xu, Lecheng Ruan, and Qining Wang are with the Department of Advanced Manufacturing and Robotics, College of Engineering, Peking University, Beijing 100871, China.

Yu-Zhe Shi, Yanjia Li, and Huamin Qu are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong SAR.

Mingchen Liu is with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.


A
Natural Language Description
DSL Program Represention of Production Procedure
DSL Program Verification on Dual Program Views
Constraint Generation in JSP Solver Format
DSL Program Representation of Production Plan
Constraint Abstraction
Constraint Generation
JSP Solver
Schedule Grounding
Pushdown Automaton for Product Flow Traversal
Lifecycles of the Intermediate Products
Constraint Programming for Scheduling Optimization
Process Lock Mechanism for Program Reference
Semi-Structured Route Sheet
Fully-Structured Route Sheet
DSL Program for Resource and Precedence Constraints
Schedule Resolved by JSP Solver
Product Flow with Concurrency Management
B
component
lathed_part_0 / conveyer_belt
lathed part
broached_c_0 / conveyer_belt
broached c.
lathe
machine: Lathe_M;
duration: 19 units;
broaching
device: Broaching_M;
duration: 21 units;
drilling
device: Drill_Press;
duration: 104 units;
metal bar
time
19u
carbide
ADINTWS
broach
lathed part
21u
depth
5mm
drilled_c_0 | drilling
broached_c_0 | broaching
lathed_part | lathe
metal_bar | bandsawing
milling
drilled_c
lathed_part_1
lathed_part_0
metal_bar_0
broached_c_0
P1 1
P1 2
P1 3
P1 4
Lathe
Broaching Machine
Drilling Press
Milling Machine
S
E
time
drilling

The vertical lathe turns a metal bar in 19u time with ADINTWS carbide...

Broach the lathed into a component in 21u time with 5mm depth cut...

Drill the component in 104u time by a 10mm drill in 500 RPM spindle...

Mill the lathed part in 161 u time by 3mm depth with 1000 RPM milling...

```
lathe(
  slot = metal_bar_0,
  duration = 19 units,
  machine = Lathe_M,
  parameters = (
    carbide = ADINTWS),
  emit = lathed_part_0);
lathed_part(...)...
broaching(...)...
```

```
jobs_data = [
    [...,
     (1, 19, [0]),
     (2, 21, [1]),
     (3, 104, [2]),
     (4, 161, [1]),...],
    ...
]
```

```
lathe(
  start_time = 0,
  end_time = 19,
  slot = metal_bar_0,
  machine = Lathe_M_#0,...)
broaching(
  start_time = 46,
  end_time = 67,
  machine = ...)
```

```
lathe::=
  /*interface_0*/
  slot ::= MAT
  duration ::= INT
  machine ::= Lathe_M
  parameters ::= ...
```

```
metal_bar ::=
  pred ::= Bandsawing
  refname ::=
metal_bar_0
  succ ::= Operation
  vol ::= REAL MEAS
```

```
broach ::=
  /*interface_2*/
  slot: PROD
  duration: INT
  machine: Broaching
  parameters ::= ...
```

```
lathed_part ::=
  pred ::= Lathe
  refname ::=
"lathed_part_0"
  succ ::= Broaching
| Milling | ...
```

No.	Operation	Operation Description
1	Lathe	The vertical lathe turns a metal bar in 19u time with ADINTWS...
2	Broaching	Broach the lathed into a component in 21u time with...
3	Drilling	Drill the component in 104u time by a 10mm drill in 500 RPM...
4	Rough Milling	Mill the lathed part in 161 u time by 3mm depth with 1000 RPM...

No.	Operation	Machine	Duration	Material	Tooling
1	Lathe	Vertical Lathe	19 units	Metal Bar	Carbide = ADINTWS
2	Broaching	Broaching Machine	21 units	Lathed Part	Depth = 5mm
3	Drilling	Drill Press	104 units	Broached Component	Depth = 10mm
4	Rough Milling	Vertical Milling Machine	161 units	Lathed Part	Depth = 3mm, ...

```
/*resource constraints*/
exec(lathe, Lathe_M) = True;
exec(drilling, Drill_P) = True;

/*precedence constraints*/
dep(lathe, broaching) = True;
dep(broaching, drilling) = True;
dep(lathe, milling) = True;
```

```
Optimal Schedule Length:
...
Machine 0: job_0_task_1
job_2_task_1  [0, 19]
[20, 52]...
Machine 1: job_4_task_3
job_0_task_2  [0, 45]
[46, 67]...
...
```

Fig. 1. **Illustration of the proposed constraint-centric architecture. (A)** This panel presents a running example that illustrates the complete information flow, transitioning from two formats of orders to a grounded production plan (depicted in the top and bottom rows). It also outlines the three modules involved: constraint abstraction, constraint generation, and schedule grounding, along with their corresponding core working mechanisms (shown in the middle row). Additionally, the DSL specified for the manufacturing scenario is highlighted as the primary driving force of the architecture. **(B)** This panel provides an intuitive visualization of the procedural transmission of information throughout the architecture's process. Progressing from left to right, the sequence of states includes the original order, fully-structured route sheet, generated constraint, JSP-solver-generated schedule, and ultimately, the production plan. The information is color-coded according to its type, such as operation name, machine, duration, and unspecified information, to enhance clarity and understanding.

duction plans. While researchers focus on improving JSP solver efficiency, the manual constraint specification process has become a critical bottleneck in APS implementation, potentially restricting smart manufacturing's full potential. Consequently, there is a growing need to automate the entire workflow from abstracting heterogeneous constraint rules to generating solver specifications and translating the resulting schedules back to interpretable production plans, thereby reducing the reliance on intensive human expert efforts.

With the rapid advancement of Artificial Intelligence Generated Content (AIGC) techniques, leveraging Large Language Models (LLMs) to automate constraint specification based on heterogenous manufacturing data and knowledge appears promising. Recent studies have demonstrated LLMs' capability to solve abstract Operations Research (OR) problems from NL descriptions, suggesting their potential applicability for specifying the constraints [20], [21]. However, despite LLMs' strengths in information extraction and NL understanding, their direct application to manufacturing constraint specification faces several fundamental challenges. (i) the intrinsic ambiguity of NL representation [22]; (ii) the non-deterministic nature of generative models like LLMs [23]; and (iii) the lack of fine-grained domain-specific knowledge regarding the specific production domains and factories in the training data of LLMs [24]. While such characteristics might be acceptable or even beneficial in creative applications like artwork composition, manufacturing systems demand absolute precision, reliability, and detailed domain knowledge [25]. Specifically, generated constraints must strictly align with factory resources and order-specified production processes, while production plans must precisely define execution configurations. This fundamental mismatch between manufacturing's reliability requirements on constraints and the capabilities of pure LLM-based solutions suggests the necessity of an external architecture atop the LLM-based workflow to regulate the LLM performance in the manufacturing scenarios.

To address these challenges, this paper proposes a *constraint-centric architecture* that regulates the LLM to perform automated reliable constraint specification from raw manufacturing knowledge and data. The architecture defines a hierarchical structural space that systematically organizes manufacturing constraints across three distinct levels [26]. The top level captures global operation dependencies and resource relationships. The middle level handles the context-specific execution configurations, while the bottom level manages detailed scheduling parameters and production specifications. The space is gradually constructed upon a representation with DSLs [27] for the natural alignment with manufacturing constraints [28], [29], the domain-specific feature to avoid redundancy [30], the support of flexible constraint combination, and the maintenance of reliability [31], [32]. To further adapt the proposed architecture across different manufacturing scenarios and factory configurations [33], and avoid the labor-intensive, case-by-case, and costly DSL crafting [34], we present an automated algorithm to efficiently adapts the architecture to specific manufacturing configurations. Comprehensive experiments are conducted to verify our advantages over pure LLM-based approaches.

In this work, we address the automatic, entire-workflow constraint specification for production scheduling, based on a domain-specific representation to regulate LLM outputs, to achieve a balance between the AIGC power and the guardrail of reliability. Our contributions are three-fold: (i) we introduce and formulate the constraint-centric architecture for reliable constraint specification (Sec. II); (ii) we develop an automatic adapter for customizing the proposed architecture across various production domains (Sec. III); (iii) we integrate the architecture into the AIGC pipeline for constraint specification, evaluate the pipeline on diverse manufacturing scenarios (Sec. IV), and demonstrate the usability and scalability of our proposed architecture (Sec. V).

II. The Constraint-Centric Architecture

In this section, we introduce the constraint-centric architecture for entire-life-cycle constraint specification (see Fig. 1). We start from an overview of the architecture (Sec. II-A). Afterwards, we describe the three modules, including abstraction from the orders (Sec. II-B), specification for JSP (Sec. II-C), and instantiation for executable production plan (Sec. II-D).

A. Architecture Overview

The primary utility of the architecture lies in its ability to take new-coming orders as input and generate a corresponding scheduled production plan as output, tailored to the specific context of a concrete factory within a manufacturing domain. The overall input to the architecture can be classified into two types: (i) NL-based order descriptions, which specify the target production procedure step-by-step in textual form; or (ii) semi-structured manufacturing route sheets, which contain extracted columns detailing the operation name and NL-based operation descriptions. According to the Standard Operating Procedure (SOP) in manufacturing, production procedure descriptions must be transformed into fully-structured manufacturing route sheets, which serve as the foundation for scheduling. These route sheets typically include columns detailing machine names and operation durations. Therefore, to accurately capture such information within the orders, the first module compiles the unstructured or semi-structured procedure descriptions into programs of the corresponding DSL of the specific manufacturing scenario.

Subsequently, the second module works on the fine-grained route sheet and specifies the constraints for JSP. This is implemented as verification over the DSL programs, generating resource constraint programs and precedence constraint programs. These programs are then transformed into a format compatible with off-the-shelf JSP solvers.

Finally, the third module grounds the resulting schedule, output by the JSP solvers, into production plans that are ready for further interpretation and execution by the Computer Numerical Control (CNC) systems within the factory environment. This step completes the concrete semantics of the operations and their corresponding execution configurations, transforming the semanticless schedule produced by the JSP solvers. This is achieved by referencing the DSL programs.

B. Constraint Abstraction from the Orders

The first module takes the order as input and outputs a fully structured representation of the target production procedures, specifically the complete manufacturing route sheet. This desired route sheet must accurately capture the operation name, required machine, standard duration, and execution configurations for each step. Given the intrinsic ambiguity of NL [22], achieving this objective necessitates the precise parsing of NL-based descriptions and the fine-grained representation of procedural knowledge. In precision-demanding scenarios like manufacturing, any deviation from the provided orders or the factory's conditions is inadmissible. In this context, we opt to utilize DSLs as the representation for production procedures, ensuring both preciseness and usability.

The working mechanism of this module can be formally expressed as CAP $= (\Upsilon \mid \Gamma, \mathcal{L})$, where $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_{|\Gamma|}\}$ represents the given *order*, which outlines the procedures for producing specific products γ_k; $\mathcal{L}$ denotes the DSL tailored to the manufacturing domain and the factory. The outcome $\Upsilon = \{\mathcal{J}, \mathcal{M}\}$ represents the set of fully structured manufacturing route sheets, where $\mathcal{J} = \{J_1, J_2, \ldots, J_{|\mathcal{J}|}\}$ indicates the *production procedures* represented as DSL programs according to the order, and $\mathcal{M} = \{M_1, M_2, \ldots, M_{|\mathcal{M}|}\}$ denotes the *factory*, equipped with a concrete set of machines.

The corresponding DSL $\mathcal{L} = \{\mathcal{L}_o, \mathcal{L}_p\}$ is indicated by a pair of dual program views: the operation-centric program view $\mathcal{L}_o$ and the product-flow-centric program view $\mathcal{L}_p$. This design choice arises from the understanding that both operations and product flows are critical elements in production procedures, yet they cannot be tracked simultaneously because they are intricately intertwined with each other. When focusing on operations, the context becomes the input and output products; conversely, when focusing on products, the context shifts to the operations that yield and consume them, respectively. By adopting a dual representation of these two views, we are able to track the detailed execution of operations and the detailed transition of product flows in parallel, resulting in a precise and reliable model of the production procedure.

The operation-centric program view focuses on the execution context of an operation. It captures the availability of necessary materials as the precondition, the required machines, and their corresponding execution configurations as the program body, along with the expected output as the postcondition. This view serves as an *interface* between the semantic identifier and the grounded instances of the operations, interpreting the purpose of the operation indicated by the order in terms of a specific execution context. For example, a milling operation can be performed on either a vertical or a horizontal milling machine, depending on the scale of the input material and the dimensions of the product.

In contrast, the product-flow-centric program view emphasizes the states of the product as it progresses through the various operations, effectively modeling the product flow. Each flow unit is produced by a predecessor operation and consumed by a successor operation. A critical property of the product flow is its spatial-temporal continuity, whereby the transitions between the states of the product are largely driven by specific operations. This view acts as a *pipe*, passing products along the temporal dimension and tracking the invariance of the entire procedure. For instance, to produce a part incorporating two different types of metal materials, the raw material input to the production line must consist of the two required materials; these materials cannot appear from nowhere but must be passed down the product flows.

The operation-centric program view, denoted as $\mathcal{L}_o = \{\mathcal{S}_o, \Lambda_o\}$, is characterized by the syntactic language feature set $\mathcal{S}_o$ and the semantic language feature set λ_o. The syntax $\mathcal{S}_o = (\varphi, \phi, \varphi^{\text{prec}}, \varphi^{\text{post}}, \varphi^{\text{exec}})$ defines a structural space for encapsulating the precondition φ^{prec}, postcondition φ^{post}, and execution φ^{exec} within the interface structure of the operation. Utilizing this syntax, an operation with the semantic identifier φ is referenced through an interface ϕ to a set of execution contexts, represented as

$$\langle \varphi \mapsto \phi \mapsto \{(\varphi^{\text{prec}}, \varphi^{\text{post}}, \varphi^{\text{exec}})\}\rangle. \tag{1}$$

The operation φ can be grounded to a corresponding instance in any compatible execution context, echoing the idea of *modular design* [35]. The semantics $\Lambda_o = (\Phi, \Phi^{\text{prec}}, \Phi^{\text{post}}, \Phi^{\text{exec}})$ specifies the permissible assignments of the fields and their corresponding values within the structural space defined by $\mathcal{S}_o$, where $\varphi \in \Phi$, $\varphi^{\text{prec}} \in \Phi^{\text{prec}}$, $\varphi^{\text{post}} \in \Phi^{\text{post}}$, and $\varphi^{\text{exec}} \in \Phi^{\text{exec}}$. Please refer to Sec. III for the automated design of such DSLs.

The product-flow-centric program view, denoted as $\mathcal{L}_p = \{\mathcal{S}_p, \Lambda_p\}$, is characterized by the syntactic language feature set $\mathcal{S}_p$ and the semantic language feature set λ_p. The syntax $\mathcal{S}_p = (\omega, \omega^{\text{pred}}, \omega^{\text{succ}}, \omega^{\text{prop}}, \psi\langle\omega^{\text{pred}}, \omega^{\text{succ}}\rangle)$ defines a structural space for encapsulating a selected set of key properties of the product ω^{prop}, the predecessor ω^{pred}, and the successor ω^{succ} within the interface structure of the product, which is inherited from the operation-centric program view. Using this syntax, a product with the semantic identifier ω is represented as $\langle \omega \mapsto (\omega^{\text{pred}}, \omega^{\text{succ}}, \omega^{\text{prop}})\rangle$. Additionally, there is a special syntactic feature $\psi\langle\omega_t^{\text{pred}}, \omega_t^{\text{succ}}\rangle$ that captures the pipe structure $\psi\langle\cdot,\cdot\rangle$ of the product flow at the unit level ω_t, indicating potential N-predecessors-to-M-successors relationships within the product flow. If the product is transferred directly from the predecessor to the successor, the pipe forms a linear structure. In cases where two products are produced by two different predecessors and consumed by one successor, the pipe forms a *"Y-shaped"* structure. This syntax can express any *"N-to-M-intersection-shaped"* relationships along the product flow, *i.e.*, N products produced by N different predecessors and consumed by M different successors. Consequently, the pipe structure syntax provides a guardrail for accurately modeling product flows of varying complexity. The semantics $\Lambda_p = (\Omega, \Omega^{\text{pred}}, \Omega^{\text{succ}}, \Omega^{\text{prop}}, \Psi)$ specifies the permissible assignments of the fields and their corresponding values within the structural space defined by $\mathcal{S}_p$, where $\omega \in \Omega$, $\omega^{\text{pred}} \in \Omega^{\text{pred}}$, $\omega^{\text{succ}} \in \Omega^{\text{succ}}$, $\omega^{\text{prop}} \in \Omega^{\text{prop}}$, and $\psi \in \Psi$. Please refer to Sec. III for the automated design of such DSLs.

We implement the translation from the original orders Γ to the fully structured route sheets Υ inspired by the practice of Shi *et al.* [36]. Given an original description of production procedure $\gamma_k \in \Gamma$ for translation, we first parse the NL sentences by an off-the-shelf tool and extract the actions accordingly [37]. Then, the extracted actions are matched with the operation set Φ of the DSL, according to both exact match score and semantic similarity. Afterwards, we extract the arrays of entities related to the extracted action $\mathcal{E}$ by an off-the-shelf LLM-based tool [38], where we regard the output labels to the entities and relations as *pseudo-labels* because they can possibly be noisy. On this basis, we can formulate the objective of this DSL program synthesis tasks as

$$\begin{aligned} &\arg\min_{^*\mathcal{L}(\Gamma), \mathcal{E}} \quad D(\Gamma \parallel \Upsilon) \\ &s.t. \quad \mathcal{L} = \{\mathcal{S}_o, \mathcal{S}_p, \Lambda_o, \Lambda_p\}, \end{aligned} \tag{2}$$

where $^*\mathcal{L}(\Gamma) = \{\mathcal{L}(\Gamma) \mid \Gamma \Rightarrow^* \mathcal{S}_o, \Gamma \Rightarrow^* \mathcal{S}_p, \mathcal{L}(\Gamma) \in \Lambda_o \cup \Lambda_p\}$ denotes the set of all possible DSL program patterns generated by the procedures described by Γ. The divergence function $D(\cdot \parallel \cdot)$ possesses three indicators: (i) the selected program patterns should be as close as possible to the text span; (ii) the selected program pattern should be as similar as possible with the extracted subject-verb-object structure parsed from NL description; and (iii) as many pseudo-labeled entities as possible should be mapped to the semantics space.

C. Constraint Generation for JSP

The second module processes fully structured route sheets, derived from the original order, and produces a specified set of constraints that formulates the JSP. This specification is subsequently converted into a format compatible with off-the-shelf JSP solvers, which are then utilized to determine the optimal schedule for the order. The constraint generation necessitates two types of constraints: (i) resource constraints, which specify *"which operation must be conducted on which machine"*; and (ii) precedence constraints, which dictate *"which operation must be conducted before which operation"*. Ensuring the accuracy of the constraint generation is paramount to guarantee that the schedules generated by the JSP solver are both meaningful and correct. This specification is achieved through contextualized DSL program verification, based on the DSL programs representing the route sheet.

The working mechanism of this module can be formally expressed as $\text{CGP} = (\text{Con} \mid \Upsilon)$. Here we look into the resulting route sheets $\Upsilon = \{\mathcal{J}, \mathcal{M}\}$ from the previous module. These route sheets outline the production procedure $J = \langle O_1, O_2, \ldots, O_{|J|}\rangle$ for each target product, detailing the execution sequence of involved operations. The complete set of production operations from all target products within the order, denoted as $\mathcal{O}_{\mathcal{J}} = \{O_1, O_2, \ldots, O_{|\mathcal{O}_{\mathcal{J}}|}\}$, forms a partially ordered set that indicates the inter-dependency relationships among operations. If the precondition of O_i, namely the availability of input materials required for O_i's execution, includes the postcondition of O_j, which is the output product following the execution of O_j, then O_i is *dependent* on O_j, denoted as $\text{dep}(O_i, O_j)$. Furthermore, an operation O_i must be executed on a specific machine M_j, represented by the relationship $\text{exec}(O_i, M_j)$.

Using the aforementioned notations, we can represent the underlying JSP for the production scenario with $\mathcal{J}$, $\mathcal{M}$, and $\mathcal{O}$. To solve the JSP, it is necessary to specify (i) the set of resource constraints $\mathcal{R} = \{(O_i, M_j) \mid O_i \in \mathcal{O}, M_j \in \mathcal{M}, \text{exec}(O_i, M_j) = \text{True}\}$; and (ii) the set of precedence constraints $\mathcal{P} = \{(O_i, O_j) \mid O_i, O_j \in \mathcal{O}, \text{dep}(O_i, O_j) = \text{True}\}$. The objectives of the JSP may include maximizing throughput, minimizing response time, or balancing resource utilization. As the objective is independent of constraint specification and thus falls outside the scope of this work, we exclude it from the problem definition for succinctness. In summary, we define the constraint generation problem as the task of specifying the resource and precedence constraints $\text{Con} = \{\mathcal{R}, \mathcal{P}\}$, given the description of orders and the actual condition of factories.

We specify Con from Υ through DSL program verification, adapted from the methodology of Shi *et al*. [36]. The DSL programs are verified by associating operations with product flows in a reciprocal manner. Product flow indicates the transfer of product flow units among operations, reflecting how one operation influences subsequent ones. The program verifier traverses the DSL program in execution order, utilizing the product locality revealed from the actual distribution of operations and product flow units. This process determines the *reachability* and *life cycle* of product flow units, in accordance with the theory of compilation introduced by Aho and Ullman [39]. For the implementation of the verifier, a Pushdown Automaton (PDA) with a random access memory is employed to record reachable product flow units as an operation context, defining (*i.e*., the product is produced by certain operations or the raw material is purchased) and killing (*i.e*., the product is consumed by certain operations) product flow units at each operation point along the computation. During every transition between operations, the killed products are removed from the memory, and the defined products are added to it. After a product flow unit is killed, the pair of operations that defined it and killed it is added to the set of precedence constraints $\mathcal{P}$. At each operation point, the pair of the operation and the machine specified in its execution configuration is added to the set of resource constraints $\mathcal{R}$. The accepting state of the PDA is reached if the memory is empty at the end of execution, meaning all products defined in operations are killed by other operations. Alongside the deterministic PDA-based verifier, We employ state-of-the-art LLMs to track and monitor the two actions of the PDA, kill and define, through instruction-following in-context learning [40], [41].

The specified constraints, along with the route sheets, are converted into the input format for the JSP solver, specifically using the widely adopted OR-Tools JSP solver[1]. Within the JSP solver framework, the manufacturing scheduling problem is modeled as a job shop environment comprising M machines and N jobs. Each job consists of a series of operations that may have inter-dependencies and require specific machine types, as indicated by precedence and resource constraints, respectively. Each operation must be assigned to a machine and processed within a specified execution duration, so as to fulfill the objective of optimizing the overall manufacturing process [42], [43]. With these parameters, the JSP can be effectively solved using constraint programming techniques [44], [45]. The resulting schedules specify the exact time-machine-operation arrangements for production.

[1]The JSP solver is implemented using the OR-Tools library, with documentation available at https://developers.google.com/optimization/scheduling.

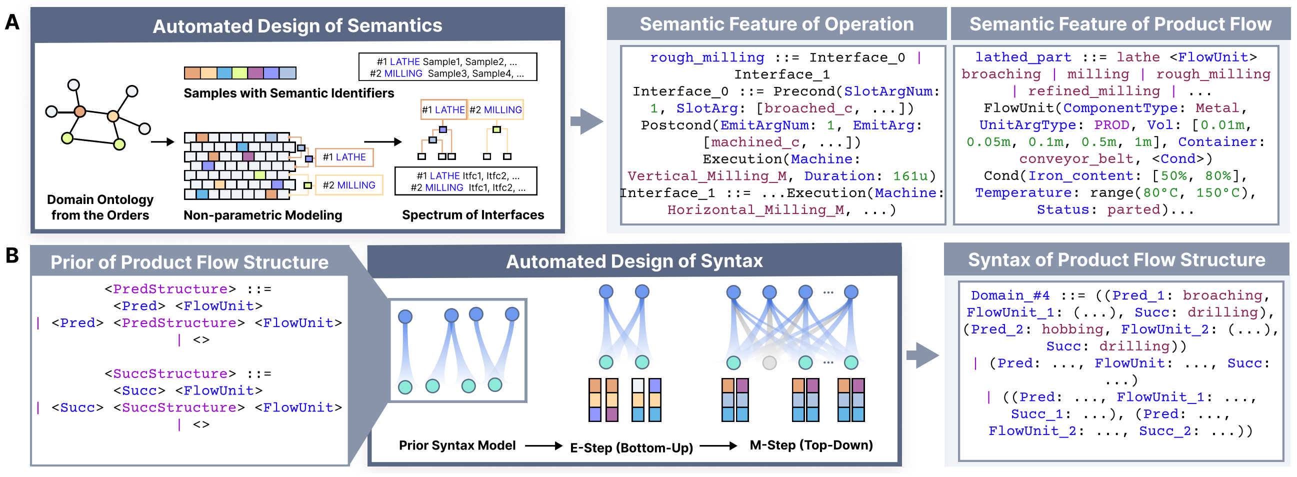
D. Schedule Grounding into Production Plans

The third module processes the schedules generated by the JSP solver, converting them into interpretable production plans within the factory environment. This module is essential for integrating off-the-shelf JSP solvers into the entire architecture, as it involves translating the physical meanings of route sheets and constraints from their DSL representations into a format compatible with the solver's input. Consequently, it is necessary to recontextualize the schedules to their physical meanings for practical application. The execution configurations in the grounded production plans must match those in the route sheets, and the execution timing must adhere precisely to the generated schedules. To ensure the reliability, this grounding is achieved through symbolic DSL program referencing.

The working mechanism of this module can be formally expressed as $\text{SGP} = (\text{Exe}(\mathcal{J}, \mathcal{M}) \mid \text{JSP}(\text{Con}, \Upsilon))$, where $\text{JSP}(\text{Con}, \Upsilon))$ denotes the output of the JSP solver. Meanwhile, $\text{Exe}(\mathcal{J}, \mathcal{M}) = \langle (O_i, M_j, t_1), \ldots, (O_k, M_l, t_{|\text{Exe}(\mathcal{J},\mathcal{M})|}) \rangle$ denotes the grounded production plans that meticulously arrange operations and machines within the temporal dimension. The incorporation of timing necessitates that the DSL program representation comprises a concurrent mechanism [46]. Specifically, an operation must wait to start until all operations producing the intermediate products required by its precondition have been completed. Given that our DSL, which adopts the product-flow-centric view, possesses the syntactic feature to model product-based transitive relationships among operations, it seamlessly facilitates the interpretation of the predecessor and successor of a product flow unit as *process locks*, serving as a guardrail for the correctness of the production plans. Furthermore, the DSL program verifier introduced in Sec. II-C functions as an additional assurance of correctness. It traverses the dependency graph to verify the absence of breakpoints in the product flow along the temporal dimension. With this dual-guardrail mechanism, we can directly reference the DSL programs using their semanticless identifiers within the JSP solver, thereby benefiting from the determinism of symbolic representations. Subsequently, we can verify the correctness of the resulting production plans.

III. Automated Domain Adaptation

In this section, we present the automated domain adaptation of the propose constraint-centric architecture. Initially, we explore the significance of such domain adaptation within the manufacturing context (Sec. III-A). We then define the problem of adapting the desired architecture by means of DSL design (Sec. III-B; refer to Fig. 2A). Afterwards, we introduce approaches for the automated design of the DSL


A
Automated Design of Semantics
#1 LATHE Sample1, Sample2, ...
#2 MILLING Sample3, Sample4, ...
Samples with Semantic Identifiers
#1 LATHE
#2 MILLING
#1 LATHE Itfc1, Itfc2, ...
#2 MILLING Itfc1, Itfc2, ...
Domain Ontology from the Orders
Non-parametric Modeling
Spectrum of Interfaces
Semantic Feature of Operation
Semantic Feature of Product Flow
B
Prior of Product Flow Structure
Automated Design of Syntax
Prior Syntax Model
E-Step (Bottom-Up)
M-Step (Top-Down)
Syntax of Product Flow Structure

```
rough_milling ::= Interface_0 |
Interface_1
Interface_0 ::= Precond(SlotArgNum:
1, SlotArg: [broached_c, ...])
Postcond(EmitArgNum: 1, EmitArg:
[machined_c, ...])
Execution(Machine:
Vertical_Milling_M, Duration: 161u)
Interface_1 ::= ...Execution(Machine:
Horizontal_Milling_M, ...)
```

```
lathed_part ::= lathe <FlowUnit>
broaching | milling | rough_milling
| refined_milling | ...
FlowUnit(ComponentType: Metal,
UnitArgType: PROD, Vol: [0.01m,
0.05m, 0.1m, 0.5m, 1m], Container:
conveyor_belt, <Cond>)
Cond(Iron_content: [50%, 80%],
Temperature: range(80°C, 150°C),
Status: parted)...
```

```
<PredStructure> ::=
<Pred> <FlowUnit>
| <Pred> <PredStructure> <FlowUnit>
| <>

<SuccStructure> ::=
<Succ> <FlowUnit>
| <Succ> <SuccStructure> <FlowUnit>
| <>
```

```
Domain_#4 ::= ((Pred_1: broaching,
FlowUnit_1: (...), Succ: drilling),
(Pred_2: hobbing, FlowUnit_2: (...),
Succ: drilling))
| (Pred: ..., FlowUnit: ..., Succ:
...)
| ((Pred: ..., FlowUnit_1: ...,
Succ_1: ...), (Pred: ...,
FlowUnit_2: ..., Succ_2: ...))
```

Fig. 2. **Illustration of the proposed algorithms for automated domain adaptation of the architecture.** **(A)** This diagram illustrates the framework of non-parametric modeling for the automated design of semantic features within both operation-centric and product-flow-centric program view DSLs. **(B)** This diagram depicts the framework of the EM algorithm for the automated design of syntactic features within product-flow-centric program view DSL.

from the operation-centric program view (Sec. III-C; also refer to Fig. 2A) and the product-flow-centric program view (Sec. III-D; refer to Fig. 2B) respectively.

A. Why Automated Domain Adaptation?

While it is theoretically possible to automate the entire workflow described in Sec. II for generating grounded production plans, a crucial challenge remains: full automation is contingent upon the availability of predefined constraint-centric architectures, *i.e.*, the corresponding DSLs for constraint specification. However, the origin of these DSLs poses a problem, as they are not readily available like off-the-shelf General-Purpose Languages (GPLs). In current practices, most DSLs are manually designed through the collaborative efforts of computer scientists and manufacturing experts, a process that is both time-consuming and costly. This may be acceptable for specific applications requiring only a single DSL library [47], [48], [49], as DSL design is a *once-and-for-all* endeavor there. Unfortunately, from a broader perspective of the holistic manufacturing community, DSLs for constraint specification in manufacturing encompass multiple categories of target products, diverse requirements of the Original Equipment Manufacturers (OEMs), varied production environments across factories, and an ever-expanding range of overall application scenarios. This domain specificity implies that the distributions of operations, machines, materials, intermediate products, execution configurations, and interdependency relationships vary significantly among different scenarios. Although it is conceivable that we derive a comprehensive set of language features covering all potential manufacturing scenarios, namely the so-called *one-size-fits-all general architecture*, such an endeavor would result in a system of prohibitively complexity, rendering it intractable for both machine and human end-users.

The highly varied and frequently evolving demands for DSLs are difficult to meet through human effort alone. Even if we manage to manually craft these DSLs, the advancement in automated production planning and scheduling would be compromised. This would merely shift human labor from one part of the workflow to another, even potentially increasing the overall labor required. Maintaining a joint cohort of experienced manufacturing experts and computer scientists for DSL design at a higher level than the current factory could prove more costly than simply sustaining experts who work at the existing factory level. Consequently, we find ourselves in a dilemma: GPLs, which easily accessible, are unsuitable for constraint specification due to their overwhelming complexity, whereas DSLs, which simplify specialized language features, inherently lack generalizability across different manufacturing scenarios. To address this dilemma, rather than waiting for a universally applicable GPL to emerge, a more practical solution might involve automating the design of DSLs for constraint specification. Therefore, the automated domain adaptation of our constraint-centered architecture is an essential requirement to fully unleash its potential of usability for the broader manufacturing community.

B. The Domain Adaptation Problem

We conceptualize the problem of the automated domain adaptation for the constraint-centric architecture within a specified manufacturing scenario as DAP $= (\{\mathcal{L}_o^*, \mathcal{L}_p^*\} \mid \Gamma')$. The objective is to generate a DSL with language features accommodating both the operation-centric program view $\mathcal{L}_o^*$ and the product-flow-centric program view $\mathcal{L}_p^*$. The input Γ' constitutes a generalized set of original orders, which may either be new orders from a recently established manufacturing scenario or historical order data from a long-standing manufacturing scenario. The prior knowledge of operations and product flows, represented by $p(\varphi)$ and $p(\omega)$, includes the fundamental syntax of the field-value structures and the elementary taxonomies, derived according to the general commonsense of manufacturing. Specifically, the problem essentially seeks to fit the joint distribution models $p(\varphi, \phi, \varphi^{\text{prec}}, \varphi^{\text{post}}, \varphi^{\text{exec}})$ and $p(\omega, \omega^{\text{pred}}, \omega^{\text{succ}}, \omega^{\text{prop}}, \psi)$, using Γ' as the domain-specific corpus, and leveraging the prior knowledge $p(\varphi)$ and $p(\omega)$.

C. Automated Operation-Centric DSL Design

The key challenge in the automated design of the operation-centric program view is to aggregate all possible execution contexts for an operation, and then generalize the contexts to the interface. If we keep each of the use case as one single instance of the interface, which can be in hundreds regarding one operation, the generalization is meaningless. Since there is no prior knowledge about the interface in advance, we develop the algorithm following the idea of non-parametric modeling, *i.e.*, Dirichlet Process Mixture Model (DPMM), resulting in flexible identification of interface instances.

As we must handle information coming in different granularities, from interface structures to values of parameters, we choose to model the operations in a hierarchical fashion. Compared with the flatten spectral clustering approach developed by Shi *et al.* [34], which compresses all information of an operation into a embedding vector, our modeling is competent for considering information at different levels comprehensively. We carefully adopt the prerequisite that the interface is generated subject to the operation, preconditions, postconditions, and execution configurations are generated subject to the interface, and the value of configuration parameters, denoted as $\varphi^{\text{exec-v}}$, are generated subject to their corresponding fields. Thus, we have the model

$$\begin{aligned} &p(\varphi, \phi, \varphi^{\text{prec}}, \varphi^{\text{post}}, \varphi^{\text{exec}}, \varphi^{\text{exec-v}}) \\ =&p(\varphi^{\text{exec-v}} \mid \varphi, \phi, \varphi^{\text{exec}}) p(\varphi^{\text{exec}} \mid \varphi, \phi) p(\varphi^{\text{prec}} \mid \varphi, \phi) \\ &p(\varphi^{\text{post}} \mid \varphi, \phi) p(\phi \mid \varphi) p(\varphi). \end{aligned} \tag{3}$$

Within each iteration of the DPMM process, we sample the variables level-by-level. Since the structures of preconditions, postconditions, and the selection of devices and configuration parameters are discrete, we sample them directly from the Dirichlet Process (DP). As permissible values of parameters can be discrete, *e.g.*, an array of specific values, common in acidity preparation; continuous, *e.g.*, an interval with minimum and maximum values, common in power setting; or mixed, *e.g.*, an array of specific values with random perturbations around the mean, common in tooling accuracy control, we conduct the sampling by integrating Gaussian Process (GP) with DP, obtaining

$$\varphi^{\text{exec-v}} \mid \varphi, \phi, \varphi^{\text{exec}} \sim DP(\alpha, H(\varphi^{\text{exec}}), \phi, \varphi) \times GP(m, K), \tag{4}$$

where α, H, m, and K are corresponding hyperparameters.

While clustering similar interface instances aggregates targets operations, there may remain redundant interfaces due to minor discrepancies. These discrepancies often arise from differences in parameter values or naming conventions that do not fundamentally alter the operation's functionality. To alleviate such redundancies, we implement a unification process for the interfaces. Specifically, interface instances associated with the same operation are considered equivalent if they have the same number of slots and emits and share the same fields in their execution configuration parameters. By abstracting away differences in parameter values and names, we unify these interfaces into a single, generalized interface, akin to the algorithm proposed by Martelli *et al.* [50]. Unification enhances the generality of the operation-centric program view by consolidating functionally-identical interfaces, maintaining a concise and representative set of operations.

D. Automated Product-Flow-Centric DSL Design

One of the primary challenges in the automated design of the product-flow-centric program view lies in selecting proper descriptive properties of a product flow unit component. There exists false positive cases, where properties are attributed to components with the same semantic identifier but in different phases, *e.g.*, we consider Aluminum Alloy with the property dimensions when it comes in flake and with the property volume when it comes in powder. There also exists false negative cases, where exact same components are regarded as different ones due to different reference names, *e.g.*, Aluminum Sheet, 6061 Alloy Plate, and Metal Sheet can refer to the same thing. To alleviate false positive and false negative results, we discard the design choice of the interface in the operation-centric view, which tends to cover the possibly richest context, and thereby have the non-parametric model

$$\begin{aligned} &p(\omega, \omega^{\text{pred}}, \omega^{\text{succ}}, \omega^{\text{prop}}, \omega^{\text{prop-v}}) \\ =&p(\omega^{\text{prop-v}} \mid \omega^{\text{prop}}, \omega) p(\omega^{\text{prop}} \mid \omega) \\ &p(\omega^{\text{pred}} \mid \omega) p(\omega^{\text{succ}} \mid \omega) p(\omega), \end{aligned} \tag{5}$$

where $\omega^{\text{prop-v}}$ denotes the values of property parameters. The challenges in building up this model and the corresponding strategies are similar to those in Sec. III-C.

The other primary challenge in the automated design of the product-flow-centric program view is constructing the model of ψ. This model is not captured by $p(\omega, \omega^{\text{pred}}, \omega^{\text{succ}}, \omega^{\text{prop}}, \omega^{\text{prop-v}})$, yet it remains crucial for completing the design of $\mathcal{S}_p$. Leveraging existing knowledge on programming language design, our method utilizes a bidirectional optimization strategy to formulate the most appropriate flow-structure-syntax $\psi^* \in \mathcal{S}_p^*$ of the target DSLs, ensuring that it compactly satisfies the characteristics dictated by Γ'. Inspired by the methodology introduced in Shi *et al.* [34], the algorithm utilizes an Expectation-Maximization (EM) framework, where the E-Step abstracts syntax from Γ' and the M-Step derives syntax from programming language principles.

The algorithm models latent syntactic constraint assignments $\mathcal{Z} = \{z_1, \ldots, z_{|\Gamma'|}\}$ for each procedure $\gamma \in \Gamma'$. A filter set $\Theta = \{\theta_1, \ldots, \theta_{|\mathcal{Z}|}\}$, is designed to determine if a segment of procedure description, *i.e.*, a local set of product flow units with predecessor and successor operations, aligns with the N-predecessors-to-M-successors relationships within the product flow, coming with the belief function $p(\Theta|\psi)$. The observational likelihood is computed as

$$p(\Gamma' \mid \mathcal{Z}, \Theta) = \prod_{i=1}^{\Gamma'} p(\gamma_i \mid z_i, \theta_{z_i}). \tag{6}$$

Hence, the overall joint distribution of the model is given by

$$p(\Gamma', \mathcal{Z}, \Theta \mid \psi) = p(\Gamma \mid \mathcal{Z}, \Theta) p(Z \mid \psi) p(\Theta \mid \psi). \tag{7}$$

Programming language designers leverage a general set of syntactic production rules as the prior $p(Z \mid \psi)$ for syntax specification. Following this common practice, we initialize ψ with the recursive grammar

$$\begin{aligned}\texttt{PredS} &::= \langle \omega^{\text{pred}} \rangle \ \langle \omega^{\text{prop}} \rangle \mid \langle \omega^{\text{pred}} \rangle \ \texttt{PredS} \ \langle \omega^{\text{prop}} \rangle \mid \langle \ \rangle, \\ \texttt{SuccS} &::= \langle \omega^{\text{succ}} \rangle \ \langle \omega^{\text{prop}} \rangle \mid \langle \omega^{\text{succ}} \rangle \ \texttt{SuccS} \ \langle \omega^{\text{prop}} \rangle \mid \langle \ \rangle,\end{aligned} \tag{8}$$

where $\langle \omega^{\text{pred}} \rangle$ $\texttt{PredS}$ $\langle \omega^{\text{prop}} \rangle$ and $\langle \omega^{\text{succ}} \rangle$ $\texttt{SuccS}$ $\langle \omega^{\text{prop}} \rangle$ are recursion bodies. This recursive grammar accommodates hypotheses involving arbitrary N-predecessors-to-M-successors relationships, naturally beginning with the simplest linear structure and progressively increasing in complexity. Additionally, we construct the prior belief function $p(\Theta \mid \psi)$ with a series of sliding-window-based filters $f : \Gamma' \mapsto \mathbb{R}$. This approach provides a relaxed lower bound for predicting the existence of an atomic product-flow structure.

In each E-Step, we obtain the posterior of latent variables $p(\mathcal{Z} \mid \Gamma', \Theta, \psi)$ applying Bayes' theorem, which is implemented by scanning the filters over all procedure descriptions in Γ'. To note, as the spaces of prior and observation are not intractably large, we simply employ the naive version of E-Step without variational approximations.

In each M-Step, we first maximize the coverage of the sampled atomic structure ψ by maximizing

$$\mathcal{Q}(\hat{\Theta}, \Theta) = \mathbb{E}_{\mathcal{Z}|\Gamma',\Theta}\big[\log p(\Gamma', \mathcal{Z}, \hat{\Theta} \mid \psi)\big], \tag{9}$$

where $\hat{\Theta}$ is the updated Θ, resulting in the structural change of ψ. These two steps alternate iteratively until convergence, ensuring the syntactic features are aligned with the scenario.

IV. Experimental Setups

In this section, we describe the experimental setups of this study. We first introduce the datasets for experimentation (Sec. IV-A) and the baseline approaches for evaluations (Sec. IV-B). Afterwards, we describe the protocols for the five experiments, including the complete pipeline experiment (Sec. IV-C), three experiments validating the three modules of our constraint-centric architecture (Secs. IV-D to IV-F), and the domain adaptation experiment (Sec. IV-G).

A. Datasets for Experimentation

The scarcity of datasets that closely replicate real-world manufacturing environments significantly hinders the evaluation of our constraint-centric architecture. To bridge this gap, we propose the augmentation of existing datasets with synthetic data, ensuring the retention of realistic elements in the process. This approach involves the utilization of ten classical JSPs sourced from well-established OR literature [51], [52], [53], [54], [55], [56], [57], [58], [59], [60]. These JSPs, originally comprising only machine IDs and durations, serve as the foundation for our dataset enhancement.

To transform abstract JSP descriptions into comprehensive datasets, we employ a cutting edge LLM[2], which extends the sparse JSP data by generating detailed production procedure descriptions and semi-structured route sheets in NL. This transformation aligns the synthetic data with the style and complexity of realistic manufacturing orders, thereby enhancing the practical value of the augmented dataset.

The augmentation process begins with a dependency graph traversal to ascertain the global dependency set across an array of device types. This step is crucial for understanding the inter-dependency relationships and operation sequences within the manufacturing setup. Subsequently, a mapping arrangement is established between the machine IDs and the corresponding devices. Each JSP is treated as a distinct *domain* requiring a DSL for accurate constraint specification. The dependency set for each machine arrangement is meticulously configured to be a superset of the JSP's dependency set, adhering to an initial assumption of ordered and linear job dependencies. Non-monotonic dependencies, indicating circular job orders, are identified and eliminated to prevent operational conflicts.

Following the establishment of device mappings and dependency configurations, the generation of synthetic data begins. This phase involves the specification of materials, products, and the execution configurations of devices. Through this data synthesis process, we obtain a dataset that not only reflects the complexity of real-world manufacturing tasks but also serves as a controllable probe for the three respective modules within our architecture and its baseline counterparts. This supports our experimental setups, enabling a thorough evaluation of our system's performance across various scenarios. It illustrates that our approach effectively harnesses the advantages of both AIGC techniques and DSL-based structural representation, thereby achieving a balance between the capability of generation and the guardrail of reliability [61].

B. Baseline Approaches

To establish a robust evaluation framework, we implement two alternative approaches based on state-of-the-art methods for formalizing NL-described optimization problems with LLM prompt engineering techniques [20], [21]. These approaches, termed Multi-Stage-LLM (MSL) and Two-Stage-LLM (TSL), serve as baselines to benchmark the utility of our proposed constraint-centric architecture (Ours).

The MSL approach adopts a three-module-sequence that mirrors the structural alignment of our constraint-centric architecture, facilitating a direct comparative analysis. Initially, the workflow transforms NL-based descriptions or semi-structured route sheets into fully structured route sheets. Subsequently, the fully structured route sheets are converted into matrices, in the format matchable for the JSP solver. The final module integrates the schedules generated by the JSP solver with the fully structured route sheet, and then grounds these schedules into executable production plans. All of these three modules are implemented by LLM prompt engineering [3].

In contrast to the MSL, the TSL approach simplifies the workflow by reducing the number of transformation stages, potentially increasing computational efficiency but at the risk of reduced fidelity in the constraint translation process. The first stage bypasses the separate structuring step of the route sheet and directly converts the NL-based descriptions or semi-structured route sheets into JSP solver formatted matrices.

[2]We use the OpenAI `GPT-4o` model for this purpose.

[3]We implement these LLMs with the OpenAI `GPT-4o` model.

The second stage involves the integration of the JSP-generated schedules with the input order, and then grounds these integrated schedules into finalized production plans. Both of these two modules are implemented by LLM prompt engineering.

C. Protocol for the Complete Pipeline Evaluation

The major objective of this experiment is to validate the utility of our constraint-centric architecture within realistic manufacturing scenarios. This experiment is critical in demonstrating the practical utility of our approach as the primary outcome of this study. The experimental protocol involves a comparative analysis of Ours against two baseline approaches, MSL and TSL. These methods are evaluated across ten realistic manufacturing scenarios as detailed in Sec. IV-A. To ensure a fair comparison, the base LLM model for all three approaches is kept identical. The input to these pipelines consists of orders described either in NL or as semi-structured route sheets. We keep the proportion of both input forms identical throughout the experiment. The output from each system is formatted into JSON-style scripts, which align with the script formats used for CNC system execution, allowing for a standardized method of comparison.

The evaluation of these scripts is conducted using the Exact Match of Key-Value Pairs (EMKVP) and Bilingual Evaluation Understudy (BLEU) metrics, which are well-established in the literature for assessing the quality of procedural knowledge representation and information retrieval [62], [63], against the groundtruth production plans derived from the synthetic dataset under the supervision of manufacturing experts. Given that direct comparisons using cosine similarity score across entire sentences could lead to inaccuracies due to semantic discrepancies in similar-looking instructions, we adopt a more granular approach. By converting all results into a standardized JSON-style format, comparisons are made between field-value pairs rather than entire sentences using the EMKVP metric. Meanwhile, procedure-level consistencies are assessed through the BLEU metric. This hybrid method effectively alleviates concerns related to the metrics by focusing either only on the precision and accuracy of specific data elements or only on overall textual similarity. Furthermore, the results are quantitatively analyzed using three specific variants of the EMKVP metric: Precision, Recall, and F1 Score. EMKVP-Precision measures the proportion of correct field-value pairs among all specified pairs, providing insight into the accuracy of the resulting production plans. EMKVP-Recall assesses the proportion of correctly specified field-value pairs out of all pairs that should have been specified, reflecting the completeness of the information captured. Lastly, EMKVP-F1 Score combines both precision and recall to offer a balanced view of overall performance.

D. Protocol for the Constraint Abstraction Evaluation

The primary objective of this experiment is to validate the effectiveness of the constraint abstraction module (CAM) within our constraint-centric architecture. This module is crucial for ensuring that ambiguities in NL parsing are managed effectively and that the precision required for fine-grained procedural knowledge representation is maintained. By isolating this component, we aim to demonstrate the indispensable role of DSLs as mechanisms that guide and regulate the behavior of LLMs within our system. The experimental setup involves using identical input data as employed in the complete pipeline experiment. The outputs generated from this input are fully-structured route sheets, which are subsequently transformed into JSON format for consistency and ease of analysis. This standardized format allows for direct comparison between the outputs from the CAM of Ours (Ours-CAM) and those from the first module of MSL (MSL-I). Similar to Sec. IV-C, the evaluation metrics include BLEU, EMKVP-Precision, EMKVP-Recall, and EMKVP-F1. These metrics are chosen for their ability to quantitatively measure the consistency between the resulting route sheets and the groundtruth fully-structured route sheets derived from synthetic data under the supervision of manufacturing experts.

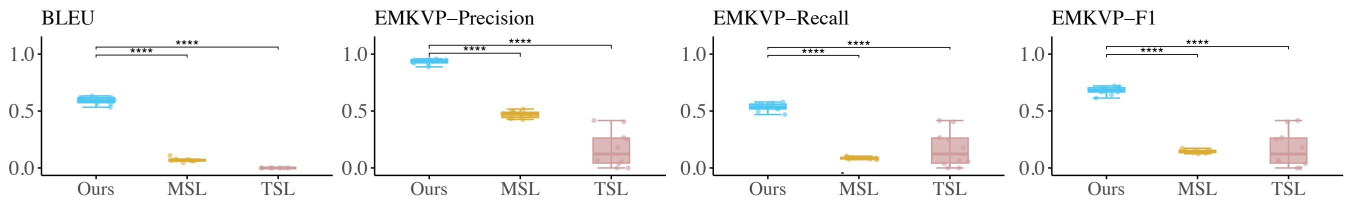
E. Protocol for the Constraint Generation Evaluation

The primary objective of this experiment is to validate the utility of the constraint generation module (CGM) within our constraint-centric architecture. This module is significant for ensuring the correctness of the constraints for JSP formulations and for converting these constraints into a format compatible with JSP solvers without any loss of information.

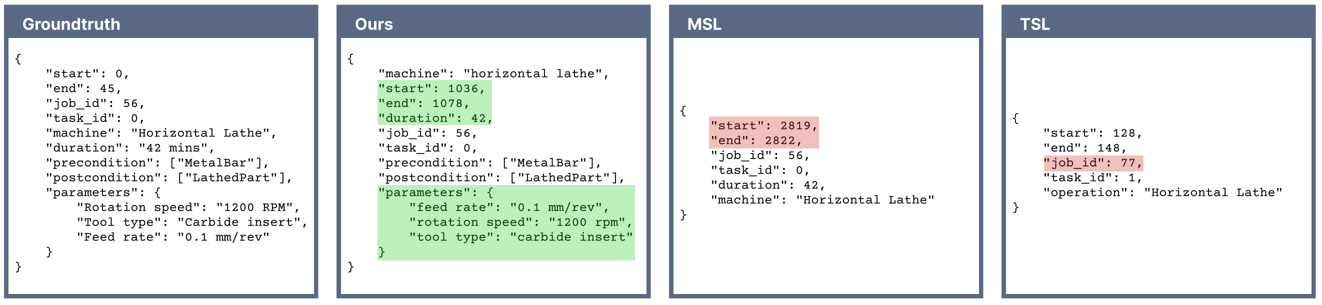
We have designed two versions of this experiment. The first version assesses the CGM in Ours (Ours-CGM) and the second module in MSL (MSL-II) to highlight the critical role of DSL program verification over the dual-program-view representation. This approach is instrumental in capturing the relationships between operations and machines, as well as among operations, in both Ours and MSL. The second version tests the integration of the first two modules, CAM & CGM, to explore the trade-offs involved in using two sequential modules with accumulated error transmission between them (*i.e.*, Ours-CAM-CGM and MSL-I-II) versus employing a single module without an explicit fully-structured route sheet as an intermediate result to work with (*i.e.*, TSL-I). For the CGM-only version, we use the groundtruth fully-structured route sheet as the input for both Ours-CGM and MSL-II to eliminate accumulated errors and truly isolate the CGM. For the CAM & CGM version, we use the same input as in the complete pipeline experiment for Ours-CAM-CGM, MSL-I-II, and TSL-I. The output for both versions is the set of constraint generation matching the input format of the JSP solver.

Given that the scheduling optimization method used in the JSP solver is deterministic, it is unnecessary to assess the correctness of the schedules generated by the JSP solver. Therefore, we adhere to established literature on benchmarking the formalization of NL-described optimization [64], incorporating three evaluation metrics: constraint-level accuracy (Constraint-Acc), compiler error rate (Compiler-ER), and runtime error rate (Runtime-ER). Constraint-Acc is calculated using the Intersection over Union (IoU) metric between the specified constraints and the groundtruth constraints, encompassing both resource and precedence constraints. This metric

A



B



Groundtruth

```
{
    "start": 0,
    "end": 45,
    "job_id": 56,
    "task_id": 0,
    "machine": "Horizontal Lathe",
    "duration": "42 mins",
    "precondition": ["MetalBar"],
    "postcondition": ["LathedPart"],
    "parameters": {
        "Rotation speed": "1200 RPM",
        "Tool type": "Carbide insert",
        "Feed rate": "0.1 mm/rev"
    }
}
```

Ours

```
{
    "machine": "horizontal lathe",
    "start": 1036,
    "end": 1078,
    "duration": 42,
    "job_id": 56,
    "task_id": 0,
    "precondition": ["MetalBar"],
    "postcondition": ["LathedPart"],
    "parameters": {
        "feed rate": "0.1 mm/rev",
        "rotation speed": "1200 rpm",
        "tool type": "carbide insert"
    }
}
```

MSL

```
{
    "start": 2819,
    "end": 2822,
    "job_id": 56,
    "task_id": 0,
    "duration": 42,
    "machine": "Horizontal Lathe"
}
```

TSL

```
{
    "start": 128,
    "end": 148,
    "job_id": 77,
    "task_id": 1,
    "operation": "Horizontal Lathe"
}
```

Fig. 3. **Results of the complete pipeline evaluation.** **(A)** Comparison of Ours with MSL and TSL across four evaluation metrics over ten domains. **(B)** Showcases of the grounded production plans generated by Ours, MSL, and TSL, respectively.

indicates the consistency between the intended orders and the interpretations by the pipelines. Both Compiler-ER and Runtime-ER concern the interactions between the pipeline and the JSP solver. The former captures the proportion of testing procedures that fail to compile in the JSP solver, potentially due to syntactic-level issues in the formatted set of constraints provided as input. The latter measures the proportion of testing procedures encountering errors during the execution of the JSP solver, which are caused by semantic-level errors in the formatted set of constraints, such as internal logic errors, unsolvable models, or non-linear constraints.

F. *Protocol for the Schedule Grounding Evaluation*

The major objective of this experiment is to validate the usefulness of the schedule grounding module (SGM) within our constraint-centric architecture. This module is essential for ensuring the accurate recovery of fine-grained procedure knowledge from the semantically void schedules produced by the JSP solver. By isolating this component, we aim to demonstrate the pivotal role of the DSL dual program view in managing the concurrent programming nature of the grounded production plans. We input the groundtruth specified set of constraints into the JSP solver to obtain the schedule. This schedule, which remains consistent across different runs, serves as the input for the SGM in Ours (Ours-SGM), the third module of MSL (MSL-III), and the second module of TSL (TSL-II). The outputs generated from this input are grounded production plans, which are subsequently converted into JSON format for consistency and ease of analysis. This standardized format facilitates direct comparison across the three pipelines. Consistent with Sec. IV-C, the evaluation metrics include BLEU, EMKVP-Precision, EMKVP-Recall, and EMKVP-F1. These metrics are selected for quantitatively assessing the consistency between the resulting production plans and the same groundtruth used in the complete pipeline experiment.

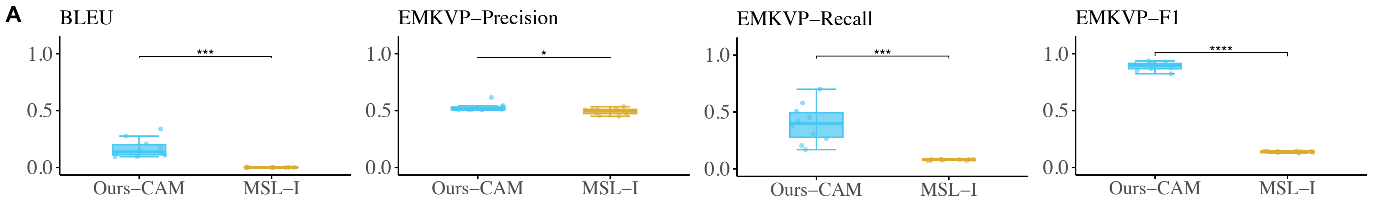
G. *Protocol for the Domain Adaptation Evaluation*

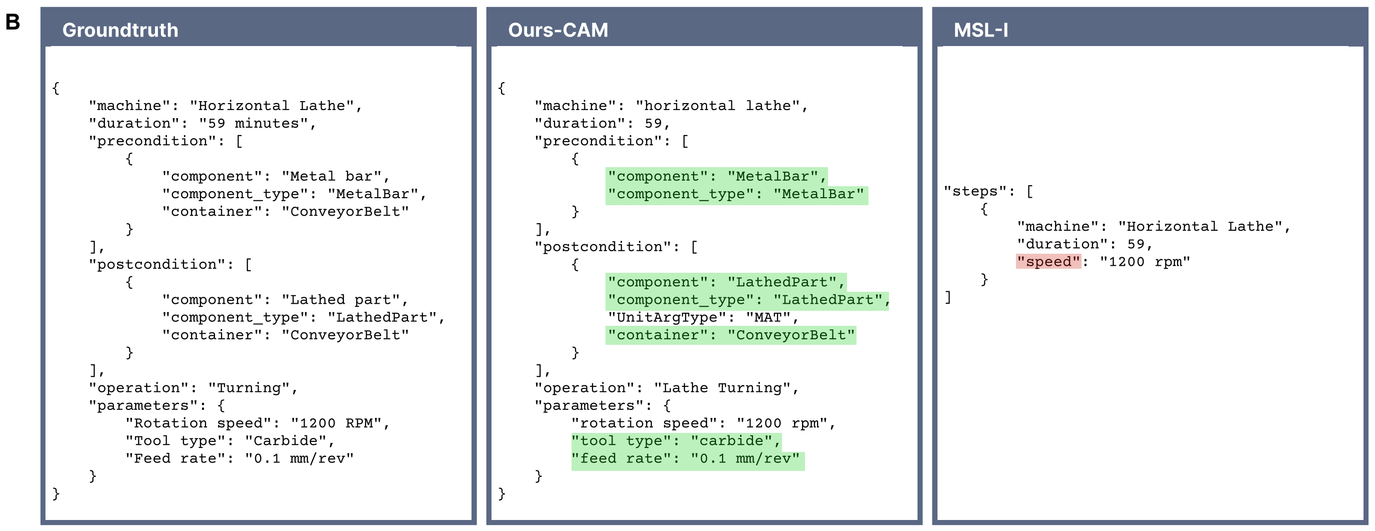
The primary objective of this meta-study-level experiment, building upon the previous four experiments, is to evaluate the scalability of our constraint-centric architecture across various manufacturing scenarios. This characteristic is central to the broader impact of our architecture on the entire manufacturing community, as it results from a trade-off between compromised generality and enhanced domain-specificity — the generalizability is thus alternatively amortized by the automated domain adaptation capability, as discussed in Sec. III-A.

This meta-study comprises two components. The first involves observing the convergence of our algorithms for automated domain adaptation across the ten selected domains. Each of the ten testing groups is initialized without any external prior knowledge, except for what is already integrated into the algorithms, such as the syntactic prior of recursion. The algorithms in these ten groups are expected to perform uniformly well, without domain-specific bias. With these automatically designed DSLs tailored for each domain, the second component involves validating the utility of the domain-adapted architecture driven by the corresponding DSL for each domain. This includes all previous experiments, such as the complete pipeline experiment and the three experiments on the individual modules of our architecture. It is expected that these experiments will not exhibit significant differences across the various domains. In addition, we utilize the Variance-to-Mean Ratio (VMR) metric, which is suitable for one-approach-multiple-domain evaluation [65], to evaluate the quantitative results of Ours, MSL, and TSL across the ten domains. This metric, where higher values are more desirable, reflects the simultaneous achievement of both outstanding and consistent performance across the different domains.

V. Results and Discussions

In this section, we analyze the results of the five experiments described in Sec. IV and discuss the insights revealed by them,




B
Groundtruth
Ours-CAM
MSL-I

Groundtruth

```
{
    "machine": "Horizontal Lathe",
    "duration": "59 minutes",
    "precondition": [
        {
            "component": "Metal bar",
            "component_type": "MetalBar",
            "container": "ConveyorBelt"
        }
    ],
    "postcondition": [
        {
            "component": "Lathed part",
            "component_type": "LathedPart",
            "container": "ConveyorBelt"
        }
    ],
    "operation": "Turning",
    "parameters": {
        "Rotation speed": "1200 RPM",
        "Tool type": "Carbide",
        "Feed rate": "0.1 mm/rev"
    }
}
```

Ours-CAM

```
{
    "machine": "horizontal lathe",
    "duration": 59,
    "precondition": [
        {
            "component": "MetalBar",
            "component_type": "MetalBar"
        }
    ],
    "postcondition": [
        {
            "component": "LathedPart",
            "component_type": "LathedPart",
            "UnitArgType": "MAT",
            "container": "ConveyorBelt"
        }
    ],
    "operation": "Lathe Turning",
    "parameters": {
        "rotation speed": "1200 rpm",
        "tool type": "carbide",
        "feed rate": "0.1 mm/rev"
    }
}
```

MSL-I

```
"steps": [
    {
        "machine": "Horizontal Lathe",
        "duration": 59,
        "speed": "1200 rpm"
    }
]
```

Fig. 4. **Results of the constraint abstraction evaluation.** **(A)** Comparison of Ours-CAM with MSL-I across four evaluation metrics over ten domains. **(B)** Showcases of the fully-structured route sheets generated by Ours-CAM and MSL-I, and TSL, respectively.

including the complete pipeline experiment (Sec. V-A), three experiments validating the utilities of the three modules of our constraint-centric architecture (Secs. V-B to V-D), and the domain adaptation experiment (Sec. V-E).

A. The Complete Pipeline Evaluation

Through paired samples t-test, we find that Ours significantly outperforms the alternative approaches MSL and TSL across the four evaluation metrics (Ours outperforms MSL, measured by BLEU: $t(18) = 47.448, \mu_d < 0, p < .0001$; measured by EMKVP-Precision: $t(18) = 39.143, \mu_d < 0, p < .0001$; measured by EMKVP-Recall: $t(18) = 39.528, \mu_d < 0, p < .0001$; measured by EMKVP-F1: $t(18) = 46.215, \mu_d < 0, p < .0001$; Ours outperforms TSL, measured by BLEU: $t(18) = 60.806, \mu_d < 0, p < .0001$; measured by EMKVP-Precision: $t(18) = 15.003, \mu_d < 0, p < .0001$; measured by EMKVP-Recall: $t(18) = 7.071, \mu_d < 0, p < .0001$; measured by EMKVP-F1: $t(18) = 9.891, \mu_d < 0, p < .0001$; see Fig. 3A). These comparisons demonstrate the suitability of our architecture for constraint specification. In addition, we find that the baseline approach equipping with an explicit fully-structured route sheet as an intermediate workspace (*i.e.*, MSL) outperforms its counterpart without such workspace (*i.e.*, TSL) (measured by BLEU: $t(18) = 13.612, \mu_d < 0, p < .0001$; measured by EMKVP-Precision: $t(18) = 5.873, \mu_d < 0, p < .0001$; see Fig. 3A). Looking into the resulting production plans, we find that Ours effectively captures fine-grained execution configurations, such as the *"feed rate"* and *"tool type"*. In contrast, MSL partially fails to capture this information, and TSL fails entirely. Among the three approaches, only Ours accurately maintains the consistency between start time, end time, and duration. The pure LLM-based counterparts even make error in this aspect, and TSL generates production plans that are *irrelevant*. These observations are illustrated by the examples presented in Fig. 3B.

B. The Constraint Abstraction Evaluation

Through paired samples t-test, we find that Ours-CAM significantly outperforms the alternative approach MSL-I across the four evaluation metrics (measured by BLEU: $t(18) = 6.487, \mu_d < 0, p < .0001$; measured by EMKVP-Precision: $t(18) = 2.481, \mu_d < 0, p < .05$; measured by EMKVP-Recall: $t(18) = 7.342, \mu_d < 0, p < .0001$; measured by EMKVP-F1: $t(18) = 28.851, \mu_d < 0, p < .0001$; see Fig. 4A). These comparisons demonstrate the capability of Ours-CAM for NL parsing and fine-grained procedural knowledge representation. Upon examining the resulting route sheets, we observe that Ours-CAM effectively captures fine-grained execution configurations, surpassing MSL-I in this regard. This achievement lays a robust foundation for subsequent processing. These observations are exemplified by the cases presented in Fig. 4B.

C. The Constraint Generation Evaluation

In the isolated version of the experiment, through paired samples t-test, we find that Ours-CGM significantly outperforms the alternative approach MSL-II on the Constraint-Acc metric ($t(18) = 11.072, \mu_d < 0, p < .0001$; see Fig. 5A).

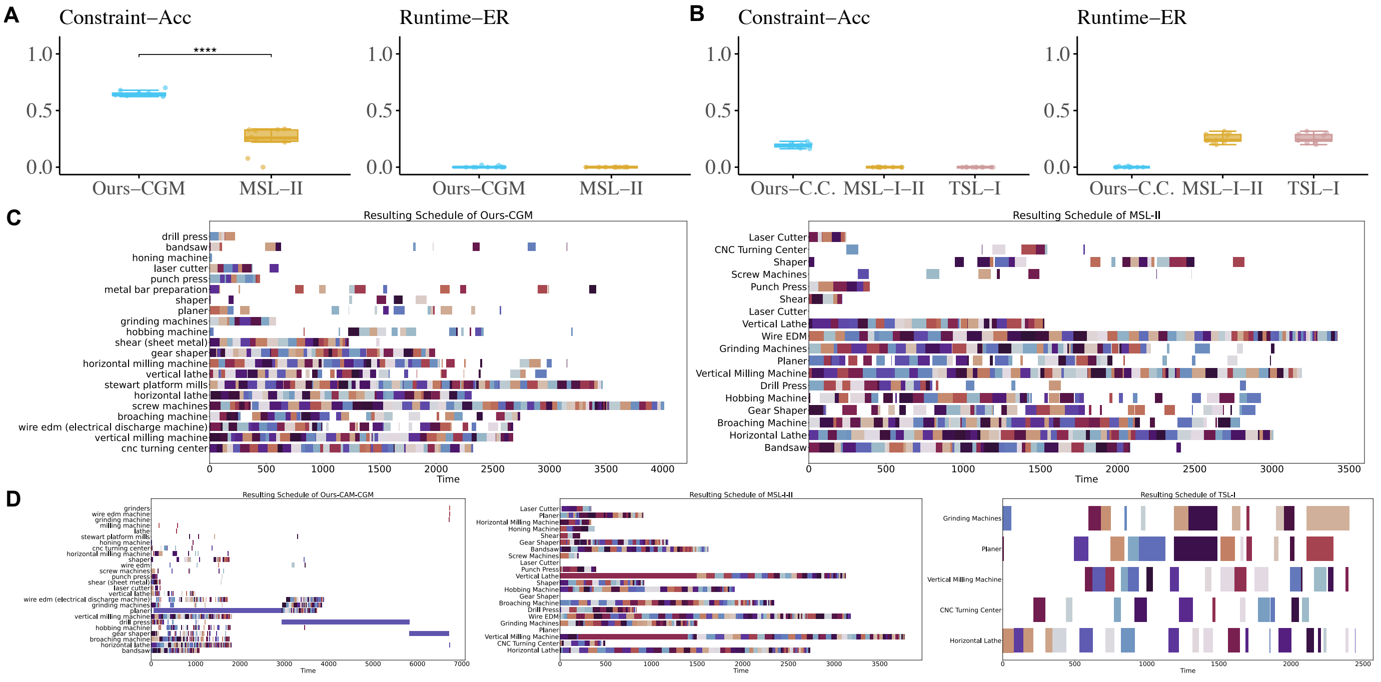


Fig. 5. **Results of the constraint generation evaluation.** **(A)** Comparison of Ours-CGM with MSL-II across the evaluation metrics Constraint-Acc and Runtime-ER over ten domains, within the isolated version of experiment. Results for Compiler-ER are not visualized because the error cases are minor and consistent across the baseline approaches. Consequently, we have omitted the corresponding plots for the sake of brevity. This design choice aligns with the discussion by Xiao *et al.* [20]. **(B)** Comparison of Ours-CAM-CGM with MSL-I-II and TSL-I across two evaluation metrics over ten domains, within the incorporated version of experiment. **(C)** Gantt chart visualizations of the JSP-solver-generated schedules, derived from the specified constraints, as generated by Ours-CGM and MSL-II, respectively. **(D)** Gantt chart visualizations of the JSP-solver-generated schedules, derived from the specified constraints, as generated by Ours-CAM-CGM, MSL-I-II, and TSL-I, respectively. The schedules derived from baseline approaches feature fewer machines, primarily due to the absence of resource constraints.

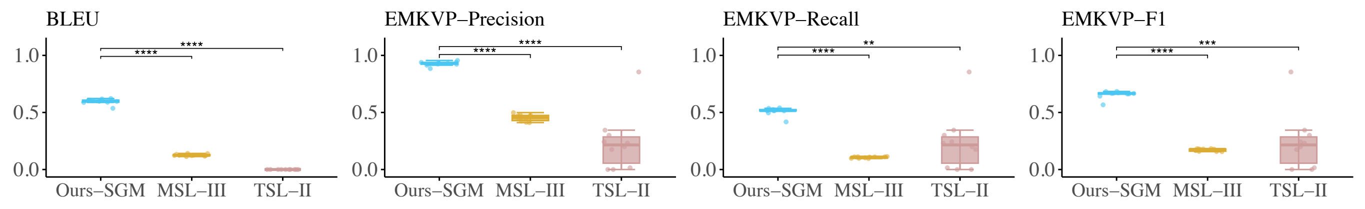


Fig. 6. **Results of the schedule grounding evaluation.** This figure presents the comparison of Ours-SGM with MSL-III and TSL-II across four evaluation metrics over ten domains.

These results support that Ours-CGM excels constraint generation through DSL program verification over the dual program views.

In the incorporated version of the experiment, through paired samples t-test, we find that Ours-CAM-CGM significantly outperforms the alternative approaches MSL-I-II and TSL-I across the three evaluation metrics (Ours-CAM-CGM outperforms MSL-I-II, measured by Constraint-Acc: $t(18) = 30.201, \mu_d < 0, p < .0001$; measured by Runtime-ER: $t(18) = -19.562, \mu_d < 0, p < .0001$; Ours-CAM-CGM outperforms TSL-I, measured by Constraint-Acc: $t(18) = 30.201, \mu_d < 0, p < .0001$; measured by Runtime-ER: $t(18) = -19.562, \mu_d < 0, p < .0001$; see Fig. 5B). These comparisons suggest that, despite the accumulated errors transmitted between the sequential module Ours-CAM-CGM, the benefits conferred by the explicit, structural intermediate workspace architecture outweigh the drawbacks of the cumulative errors, which are mitigated by TSL-I. An examination of the schedules produced by the JSP solvers reveals that the baseline approaches fail to accurately specify both resource and precedence constraints. This inadequacy results in schedules characterized by a high rate of task loss and unreliable dependency relationships, as depicted in the Gantt charts in Fig. 5C-D. The higher Runtime-ER observed in the baseline models corroborates this issue, as evidenced by the substantial presence of *“undefined”* and *“null”* values in their formatted constraints. These findings substantiate the rationale underpinning the design of our architecture.

D. The Schedule Grounding Evaluation

Through paired samples t-test, we find that Ours-SGM significantly outperforms the alternative approaches MSL-III and TSL-II across the four evaluation metrics (Ours-SGM outperforms MSL-III, measured by BLEU: $t(18) = 58.332, \mu_d < 0, p < .0001$; measured by EMKVP-Precision:

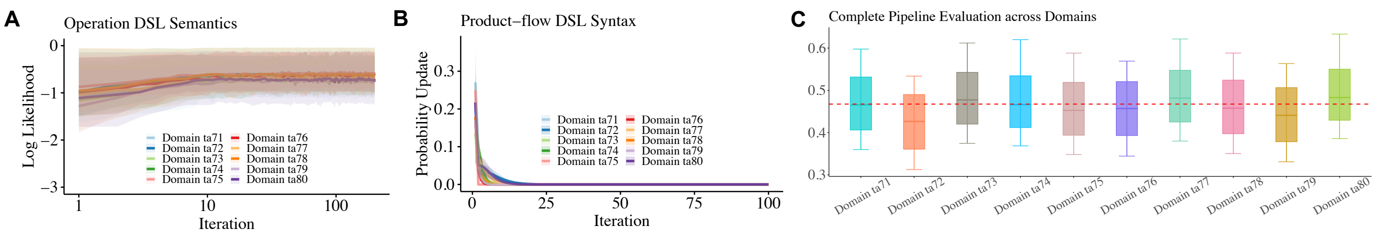


Fig. 7. **Results of the domain adaptation evaluation.** **(A)** Convergence curve of the non-parametric model for the automated design of semantic features within operation-centric program view DSLs. **(B)** Convergence curve of the EM algorithm for the automated design of syntactic features in product-flow-centric program view DSL. **(C)** Comparison of Ours with MSL and TSL on ten domains respectively integrating the four evaluation metrics.

$t(18) = 42.726, \mu_d < 0, p < .0001$; measured by EMKVP-Recall: $t(18) = 36.001, \mu_d < 0, p < .0001$; measured by EMKVP-F1: $t(18) = 43.236, \mu_d < 0, p < .0001$; Ours-SGM outperforms TSL-II, measured by BLEU: $t(18) = 79.089, \mu_d < 0, p < .0001$; measured by EMKVP-Precision: $t(18) = 8.705, \mu_d < 0, p < .0001$; measured by EMKVP-Recall: $t(18) = 3.422, \mu_d < 0, p < .01$; measured by EMKVP-F1: $t(18) = 5.273, \mu_d < 0, p < .0001$; see Fig. 6A). These comparisons highlight the effectiveness of Ours-SGM in schedule grounding, particularly through the dual program view of the DSL in managing the concurrent programming nature of grounded production plans. In contrast, even when provided with groundtruth schedules, the baseline approaches perform poorly in maintaining consistency among start time, end time, and duration, let alone their performances on addressing the issue of occupation caused by product transitions.

E. The Domain Adaptation Evaluation

In the first component of this meta-study, we present the trends observed from the behaviors of the automated design algorithms. The automated design algorithm of both operation-centric and product-flow-centric program view DSL semantics converges on ten domains respectively, as illustrated by the likelihood curve yielded by the non-parametric model in Fig. 7A. The automated design algorithm of product-flow-centric program view DSL syntax converges on ten domains respectively, as illustrated by the likelihood curve generated by the EM in Fig. 7B. Given the domain specificity, the automated design algorithms effectively tailor the resulting DSLs to align closely with the characteristics of their respective domains. These algorithms adeptly capture the unique language features inherent to each domain, preserve the common ones, and prune out those deemed unnecessary.

In the second component of this meta-study, the null hypothesis posits that the performance of Ours in uncorrelated with the choice of domain. Utilizing the Kruskal-Wallis H-Test, we determine that the null hypothesis should be accepted based on the complete pipeline experiment ($H(9) = 2.605, p = .978$; see Fig. 7E). This indicates a lack of evidence supporting a correlation between performance and domain. Furthermore, we observe that the results from Ours demonstrate a high mean and low variance, resulting in a trend where the VMR of Ours significantly surpasses that of TSL but does not significantly exceed that of MSL. This is because a substantial portion of MSL's results exhibit both low mean and low variance, whereas TSL's results display both low mean and high variance, indicating a greater degree of uncertainty. This difference of uncertainty aligns with the insights derived from the route sheet as an intermediate workspace. These results suggest that our constraint-centric architecture delivers both exceptional and consistent performance across various domains, thereby meeting the requirements of the manufacturing community. The findings further reveal the potential of our architecture to democratize the automatic, entire-workflow constraint specification for production planning and scheduling for all manufacturing practitioners, ranging from OEMs to Small and Medium-sized Enterprises (SMEs).

VI. Conclusion

This paper addresses the critical challenge of automating constraint specification from heterogeneous raw data in smart manufacturing by introducing a constraint-centric architecture that effectively regulates LLMs through domain-specific representations. Our three-level hierarchical structure successfully balances the generative capabilities of LLMs with manufacturing reliability requirements. The automated adaptation algorithm enables efficient customization across different manufacturing scenarios, making the architecture broadly applicable for various domains. Experimental results demonstrate the architecture's superiority over pure LLM-based approaches in maintaining precision while reducing reliance on human expert intervention. This work advances the practical implementation of smart manufacturing by providing a robust framework for automated constraint specification that meets the demands of modern production environments. Future research could explore the architecture's adaptation boundary to more complex manufacturing scenarios and its integration with other smart manufacturing systems.

Acknowledgment

The authors would like to thank Jiawen Liu for the assistance in figure drawing.

References

[1] A. Kusiak, "Smart manufacturing," *International journal of production Research*, vol. 56, no. 1-2, pp. 508–517, 2018.

[2] C. J. Conti, A. S. Varde, and W. Wang, "Human-robot collaboration with commonsense reasoning in smart manufacturing contexts," *IEEE Transactions on Automation Science and Engineering*, vol. 19, no. 3, pp. 1784–1797, 2022.

[3] E. C. Balta, M. Pease, J. Moyne, K. Barton, and D. M. Tilbury, "Digital twin-based cyber-attack detection framework for cyber-physical manufacturing systems," *IEEE Transactions on Automation Science and Engineering*, vol. 21, no. 2, pp. 1695–1712, 2023.

[4] L. Chen, Z. Lu, A. Xiao, Q. Duan, J. Wu, and P. C. Hung, "A resource recommendation model for heterogeneous workloads in fog-based smart factory environment," *IEEE Transactions on Automation Science and Engineering*, vol. 19, no. 3, pp. 1731–1743, 2022.

[5] J. J. Browne, D. Dubois, K. Rathmill, S. Sethi, and K. Stecke, "Classification of flexible manufacturing systems," *The FMS magazine*, vol. 2, no. 2, pp. 114–117, 1984.

[6] H.-H. Hvolby and K. Steger-Jensen, "Technical and industrial issues of Advanced Planning and Scheduling (APS) systems," *Computers in Industry*, vol. 61, no. 9, pp. 845–851, 2010.

[7] S. Dauzère-Pérès, J. Ding, L. Shen, and K. Tamssaouet, "The flexible job shop scheduling problem: A review," *European Journal of Operational Research*, vol. 314, no. 2, pp. 409–432, 2024.

[8] H. Xiong, S. Shi, D. Ren, and J. Hu, "A survey of job shop scheduling problem: The types and models," *Computers & Operations Research*, vol. 142, p. 105731, 2022.

[9] K. Gao, Z. Cao, L. Zhang, Z. Chen, Y. Han, and Q. Pan, "A review on swarm intelligence and evolutionary algorithms for solving flexible job shop scheduling problems," *IEEE/CAA Journal of Automatica Sinica*, vol. 6, no. 4, pp. 904–916, 2019.

[10] B. Cunha, A. M. Madureira, B. Fonseca, and D. Coelho, "Deep reinforcement learning as a job shop scheduling solver: A literature review," in *Hybrid Intelligent Systems: 18th International Conference on Hybrid Intelligent Systems (HIS 2018) Held in Porto, Portugal, December 13-15, 2018 18*, pp. 350–359, Springer, 2020.

[11] Y.-J. Yao, Q.-H. Liu, X.-Y. Li, and L. Gao, "A novel milp model for job shop scheduling problem with mobile robots," *Robotics and Computer-Integrated Manufacturing*, vol. 81, p. 102506, 2023.

[12] D. B. Fontes, S. M. Homayouni, and J. C. Fernandes, "Energy-efficient job shop scheduling problem with transport resources considering speed adjustable resources," *International Journal of Production Research*, vol. 62, no. 3, pp. 867–890, 2024.

[13] D. B. Fontes, S. M. Homayouni, and J. F. Gonçalves, "A hybrid particle swarm optimization and simulated annealing algorithm for the job shop scheduling problem with transport resources," *European Journal of Operational Research*, vol. 306, no. 3, pp. 1140–1157, 2023.

[14] F. Zhang, Y. Mei, S. Nguyen, and M. Zhang, "Survey on genetic programming and machine learning techniques for heuristic design in job shop scheduling," *IEEE Transactions on Evolutionary Computation*, vol. 28, no. 1, pp. 147–167, 2023.

[15] J. Li, X. Li, L. Gao, and Q. Liu, "A flexible job shop scheduling problem considering on-site machining fixtures: A case study from customized manufacturing enterprise," *IEEE Transactions on Automation Science and Engineering*, 2024.

[16] Z. Tian, X. Jiang, W. Liu, and Z. Li, "Dynamic energy-efficient scheduling of multi-variety and small batch flexible job-shop: A case study for the aerospace industry," *Computers & Industrial Engineering*, vol. 178, p. 109111, 2023.

[17] L. Zhao, J. Fan, C. Zhang, W. Shen, and J. Zhuang, "A drl-based reactive scheduling policy for flexible job shops with random job arrivals," *IEEE Transactions on Automation Science and Engineering*, 2023.

[18] T. Wang, J. Zhao, Q. Xu, W. Pedrycz, and W. Wang, "A dynamic scheduling framework for byproduct gas system combining expert knowledge and production plan," *IEEE Transactions on Automation Science and Engineering*, vol. 20, no. 1, pp. 541–552, 2022.

[19] H. Zhang and U. Roy, "A semantics-based dispatching rule selection approach for job shop scheduling," *Journal of Intelligent Manufacturing*, vol. 30, no. 7, pp. 2759–2779, 2019.

[20] Z. Xiao, D. Zhang, Y. Wu, L. Xu, Y. J. Wang, X. Han, X. Fu, T. Zhong, J. Zeng, M. Song, *et al.*, "Chain-of-experts: When llms meet complex operations research problems," in *International Conference on Learning Representations*, 2023.

[21] A. Ahmaditeshnizi, W. Gao, and M. Udell, "Optimus: Scalable optimization modeling with (mi) lp solvers and large language models," in *International Conference on Machine Learning*, 2024.

[22] B. Russell, "Vagueness," *The Australasian Journal of Psychology and Philosophy*, vol. 1, no. 2, pp. 84–92, 1923.

[23] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of chatgpt in code generation," *ACM Transactions on Software Engineering and Methodology*, 2024.

[24] V. G. Cannas, M. P. Ciano, M. Saltalamacchia, and R. Secchi, "Artificial intelligence in supply chain and operations management: a multiple case study research," *International Journal of Production Research*, vol. 62, no. 9, pp. 3333–3360, 2024.

[25] Y.-Z. Shi, M. Xu, J. E. Hopcroft, K. He, J. B. Tenenbaum, S.-C. Zhu, Y. N. Wu, W. Han, and Y. Zhu, "On the complexity of Bayesian generalization," in *International Conference on Machine Learning*, 2023.

[26] Y.-Z. Shi, Q. Xu, F. Meng, L. Ruan, and Q. Wang, "Abstract Hardware Grounding towards the Automated Design of Automation Systems," in *International Conference on Intelligent Robotics and Applications*, 2024.

[27] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[28] A. Tarski, *Introduction to Logic and to the Methodology of Deductive Sciences*. Dover Publications, 1946.

[29] S. J. Russell and P. Norvig, *Artificial intelligence a modern approach*. Prentice Hall Press, 2010.

[30] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, "Design guidelines for domain specific languages," in *OOPSLA Workshop on Domain-Specific Modeling (DSM' 09)*, 2009.

[31] N. Chomsky, *Syntactic Structures*. Mouton de Gruyter, 1957.

[32] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[33] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[34] Y.-Z. Shi, H. Hou, Z. Bi, F. Meng, X. Wei, L. Ruan, and Q. Wang, "AutoDSL: Automated domain-specific language design for structural representation of procedures with constraints," in *Annual Meeting of the Association for Computational Linguistics*, 2024.

[35] H. Abelson and G. J. Sussman, *Structure and interpretation of computer programs*. The MIT Press, 1996.

[36] Y.-Z. Shi, F. Meng, H. Hou, Z. Bi, Q. Xu, L. Ruan, and Q. Wang, "Expert-level protocol translation for self-driving labs," in *Advances in Neural Information Processing Systems*, 2024.

[37] M. Honnibal and M. Johnson, "An improved non-monotonic transition system for dependency parsing," in *Annual Conference on Empirical Methods in Natural Language Processing*, 2015.

[38] T. Xie, Q. Li, Y. Zhang, Z. Liu, and H. Wang, "Self-improving for zero-shot named entity recognition with large language models," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics*, 2024.

[39] A. V. Aho and J. D. Ullman, *The theory of parsing, translation, and compiling*. Prentice-Hall Englewood Cliffs, NJ, 1972.

[40] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, 2020.

[41] J. Wei, M. Bosma, V. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," in *International Conference on Learning Representations*, 2022.

[42] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[43] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," in *Annals of discrete mathematics*, Elsevier, 1979.

[44] P. Baptiste, C. Le Pape, and W. Nuijten, *Constraint-based scheduling: applying constraint programming to scheduling problems*. Springer Science & Business Media, 2001.

[45] P. Baptiste, P. Laborie, C. Le Pape, and W. Nuijten, "Constraint-based scheduling and planning," in *Foundations of artificial intelligence*, vol. 2, pp. 761–799, Elsevier, 2006.

[46] K. M. Chandy, "Parallel program design," in *Opportunities and Constraints of Parallel Computing*, Springer, 1989.

[47] M. Voelter and E. Visser, "Product line engineering using domain-specific languages," in *2011 15th IEEE International Software Product Line Conference*, 2011.

[48] K. Meixner, F. Rinker, H. Marcher, J. Decker, and S. Biffl, "A domain-specific language for product-process-resource modeling," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021.

[49] M. Hofmann, L. Albaugh, T. Wang, J. Mankoff, and S. E. Hudson, "Knitscript: A domain-specific scripting language for advanced machine knitting," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023.

[50] A. Martelli and U. Montanari, "An efficient unification algorithm," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 2, pp. 258–282, 1982.

[51] E. Taillard, "Benchmarks for basic scheduling problems," *European Journal of Operational Research*, vol. 64, no. 2, pp. 278–285, 1993.

[52] J. Adams, E. Balas, and D. Zawack, "The shifting bottleneck procedure for job shop scheduling," *Management Science*, vol. 34, no. 3, pp. 391–401, 1988.

[53] D. Applegate and W. Cook, "A computational study of the job-shop scheduling problem," *ORSA Journal on Computing*, vol. 3, no. 2, pp. 149–156, 1991.

[54] C. Bierwirth, "A generalized permutation approach to job shop scheduling with genetic algorithms," *Operations Research Spektrum*, vol. 17, no. 2, pp. 87–92, 1995.

[55] E. Balas and A. Vazacopoulos, "Guided local search with shifting bottleneck for job shop scheduling," *Management Science*, vol. 44, no. 2, pp. 262–275, 1998.

[56] J. Carlier and É. Pinson, "An algorithm for solving the job-shop problem," *Management Science*, vol. 35, no. 2, pp. 164–176, 1989.

[57] A. S. Jain and S. Meeran, "Deterministic job-shop scheduling: Past, present and future," *European Journal of Operational Research*, vol. 113, no. 2, pp. 390–434, 1999.

[58] E. Nowicki and C. Smutnicki, "A fast taboo search algorithm for the job shop problem," *Management Science*, vol. 42, no. 6, pp. 797–813, 1996.

[59] I. Sabuncuoglu and M. Bayiz, "Job shop scheduling with beam search," *European Journal of Operational Research*, vol. 118, no. 2, pp. 390–412, 1999.

[60] R. H. Storer, S. D. Wu, and R. Vaccari, "New search spaces for sequencing problems with application to job shop scheduling," *Management Science*, vol. 38, no. 10, pp. 1495–1509, 1992.

[61] Y.-Z. Shi, H. Li, L. Ruan, and H. Qu, "Constraint representation towards precise data-driven storytelling," in *IEEE Visualization and Visual Analytics Gen4DS Workshop*, 2024.

[62] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008.

[63] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Annual Meeting of the Association for Computational Linguistics*, 2002.

[64] R. Ramamonjison, T. Yu, R. Li, H. Li, G. Carenini, B. Ghaddar, S. He, M. Mostajabdaveh, A. Banitalebi-Dehkordi, Z. Zhou, *et al.*, "Nl4opt competition: Formulating optimization problems based on their natural language descriptions," in *NeurIPS 2022 Competition Track*, 2023.

[65] Y.-Z. Shi, S. Li, X. Niu, Q. Xu, J. Liu, Y. Xu, S. Gu, B. He, X. Li, X. Zhao, *et al.*, "PersLEARN: Research Training through the Lens of Perspective Cultivation," in *Annual Meeting of the Association for Computational Linguistics*, 2023.